Transducers

What are they?

How can you use them in JavaScript?

Pavel Forkert

Programming languages &
design patterns geek

Basics

map

```
[1, 2, 3, 4, 5].map(function(element) {
  return element * 2;
});
=> [2, 4, 6, 8, 10]
```

filter

```
[1, 2, 3, 4, 5].filter(function(element) {
  return element % 2 === 0;
});
=> [2, 4]
```

reduce

```
[1, 2, 3, 4, 5].reduce(function(accumulator, element) {
  console.log(accumulator, element);
  return accumulator + element;
}, 0);

0 1
1 2
3 3
6 4
10 5
=> 15
```

Reducers - A Library and Model for Collection Processing

Posted by Rich Hickey on May 08, 2012

I'm happy to have pushed today the beginnings of a new Clojure library for higher-order manipulation of collections, based upon *reduce* and *fold*. Of course, Clojure already has Lisp's *reduce*, which corresponds to the traditional *foldl* of functional programming. *reduce* is based upon sequences, as are many of the core functions of Clojure, like *map*, *filter* etc. So, what could be better? It's a long story, so I'll give you the ending first:

- There is a new namespace: clojure.core.reducers
- It contains new versions of *map*, *filter* etc based upon transforming reducing functions - reducers
- It contains a new function, **fold**, which is a parallel reduce+combine
- *fold* uses **fork/join** when working with (the existing!) Clojure vectors and maps
- Your new parallel code has exactly the same shape as your existing seq-based

Anatomy of a Reducer

Posted by Rich Hickey on May 15, 2012

Last time, I blogged about Clojure's new reducers library. This time I'd like to look at the details of what constitutes a reducer, as well as some background about the library.

What's a Reducing Function?

The reducers library is built around transforming reducing functions. A reducing function is simply a binary function, akin to the one you might pass to *reduce*. While the two arguments might be treated symmetrically by the function, there is an implied semantic that distinguishes the arguments: the first argument is a result or accumulator that is being built up by the reduction, while the second is some new input value from the source being reduced. While *reduce* works from the 'left', that is neither a property nor promise of the reducing function, but one of *reduce* itself. So we'll say simply that a reducing fn has the shape:

Reducers

```
[0, 1, 2, 3, 4, 5, 6, 7]
  .map(function(element) { return 1 << element; })
  .reduce(function(acc, element) {
    return acc + element;
  }, 0);
=> 255
```

Reducers

```
var powerOfTwo = function(element) { return 1 << element; };
var map = function(fn, coll) { return coll.map(fn); };
var reduce = function(step, initial, coll) {
  return coll.reduce(step, initial);
};
reduce(
  function(acc, element) { return acc + element; },
  0,
  map(powerOfTwo, [0, 1, 2, 3, 4, 5, 6, 7]));
```

Reducers

```
var map = function(fn, coll) {
  return { fn: fn, coll: coll };
};
var reduce = function(step, initial, transformResult) {
  var coll = transformResult.coll;
  var tfn = transformResult.fn;
  var newStep = function(acc, element) {
    return step(acc, tfn(element));
  };
  return coll.reduce(newStep, initial);
};
```

Reducers

```
var map = function(fn, coll) {
  return {
    reduce: function(step, initial) {
      return coll.reduce(function(acc, element) {
        return step(acc, fn(element));
      }, initial);
    }
  };
};
```

Reducers

```
var reduce = function(initial, step, coll) {
  return coll.reduce(step, initial);
};
```

Reducers

```
var filter = function(pred, coll) {
  return {
    reduce: function(step, initial) {
      return coll.reduce(function(acc, element) {
        return pred(element) ? step(acc, element) : acc;
      }, initial);
    }
  };
};
```

Reducers

```
var even = function(element) { return element % 2 === 0; };

reduce(
  function(acc, element) { return acc + element; },
  0,
  filter(even, map(powerOfTwo, [0, 1, 2, 3, 4, 5, 6, 7])));
=> 254

reduce(
  function(acc, element) { return acc + element; },
  0,
  map(powerOfTwo, filter(even, [0, 1, 2, 3, 4, 5, 6, 7])));
=> 85
```

Transforming Reducing Functions

A function that transforms a reducing fn simply takes one, and returns another one:

```
(xf reducing-fn) -> reducing-fn
```

Many of the core collection operations can be expressed in terms of such a transformation. Imagine if we were to define the cores of *map*, *filter* and *mapcat* in this way:

```
(defn mapping [f]
  (fn [f1]
    (fn [result input]
      (f1 result (f input)))))

(defn filtering [pred]
  (fn [f1]
    (fn [result input]
      (if (pred input)
        (f1 result input)
        result))))

(defn mapcatting [f]
  (fn [f1]
    (fn [result input]
      (reduce f1 result (f input)))))
```

There are a few things to note:

“Using these directly is somewhat odd”

–Rich Hickey

TRANSDUCERS ARE COMING

Posted by Rich Hickey on August 6, 2014

Transducers are a powerful and composable way to build algorithmic transformations that you can reuse in many contexts, and they're coming to Clojure core and core.async.

Two years ago, in a blog post describing how reducers work, I described the *reducing function transformers* on which they were based, and provided explicit examples like '**mapping**', '**filtering**' and '**mapcatting**'. Because the reducers library intends to deliver an API with the same 'shape' as existing sequence function APIs, these transformers were never exposed a la carte, instead being encapsulated by the macrology of reducers.

In working recently on providing algorithmic combinators for core.async, I became more and more convinced of the superiority of reducing function transformers over channel->channel functions for algorithmic transformation. In fact, I think they are a better way to do many things for which we normally create bespoke replicas of map, filter etc.

So, reducing function transformers are getting a name - '**transducers**', and first-class support in Clojure core and core.async.

WHAT'S A TRANSDUCER?

To recap that earlier post:

A reducing function is just the kind of function you'd pass to **reduce** - it takes a result so far and a new input

Search

TAGS

CLOJURESCRIPT
CONTINUOUS DELIVERY
QA
SPONSORSHIP
AGILE
COMMUNITY
CULTURE
ENTERPRISE
ETSILY
LOAD TEST
SIMULATION TESTING
TESTING
TESTING TOOLS
TRAINING
COGNITECT

Why?

- Keep advantages of reducers
- Decouple transformations from inputs and outputs
- Allow transformations to work in different contexts (channels, eager collections, lazy collections, queues, event streams)
- Composability!

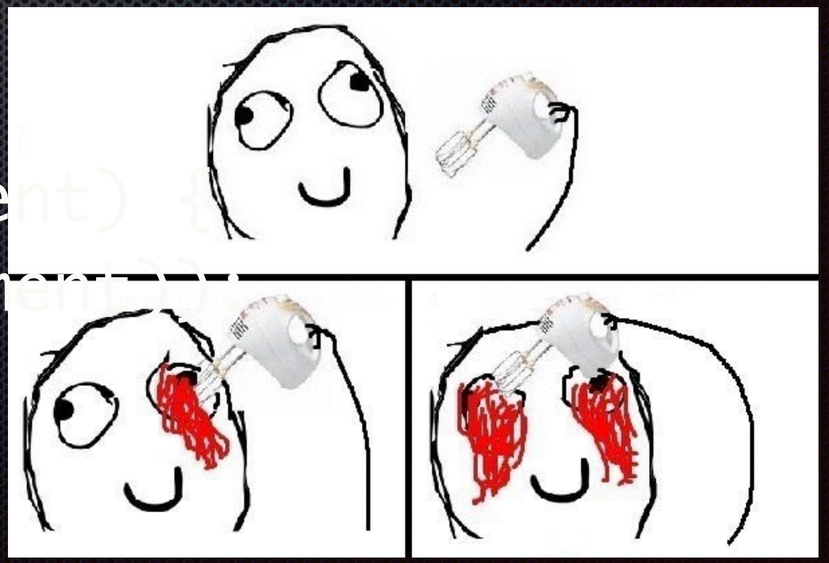
transducers

transforming reducers

Transducers

- (accumulator, element) -> accumulator
- ((accumulator, element) -> accumulator) ->
  ((accumulator, element) -> accumulator)

```
function map(fn) {
  return function(step) {
    return function(accumulator, element) {
      return step(accumulator, fn(element));
    };
  };
};
```



Reducers

```
var map = function(fn, coll) {
  return {
    reduce: function(step, initial) {
      return coll.reduce(function(acc, element) {
        return step(acc, fn(element));
      }, initial);
    }
  };
};
```

Transducers

```
var map = function(fn, coll) {
  return {
    reduce: function(step, initial) {
      return coll.reduce(function(acc, element) {
        return step(acc, fn(element));
      }, initial);
    }
  };
};
```

Reducers

```
reduce(
  function(acc, element) { return acc + element; },
  0,
  map(powerOfTwo, [0, 1, 2, 3, 4, 5, 6, 7]));
```

Transducers

```
transduce(
  map(powerOfTwo),
  function(acc, element) { return acc + element; },
  0,
  [0, 1, 2, 3, 4, 5, 6, 7]);
```

https://github.com/cognitect-labs/transducers-js

Transducers

```
reduce(
  map(powerOfTwo)(function(acc, element) { return acc +
element; }),
  0,
  [0, 1, 2, 3, 4, 5, 6, 7]);
```

Reduce as collection builder

```
var append = function(acc, element) {
  acc.push(element);
  return acc;
};

reduce(
  append,
  [],
  [0, 1, 2, 3, 4, 5, 6, 7]);
=> [0, 1, 2, 3, 4, 5, 6, 7]
```

Transducers

```
transduce(
  map(powerOfTwo),
  append,
  [],
  [0, 1, 2, 3, 4, 5, 6, 7]);

=> [1, 2, 4, 8, 16, 32, 64, 128]
```

Transducers

```
var map = function(fn, coll) {
  return transduce(
    transducers.map(fn),
    append,
    [],
    coll);
}
```

Transducers / map-into

```
var array = [];
array = transduce(map(powerOfTwo), append, array,
  [0, 1, 2, 3]);
=> [ 1, 2, 4, 8 ]
array = transduce(map(powerOfTwo), append, array,
  [4, 5, 6, 7]);
=> [1, 2, 4, 8, 16, 32, 64, 128]
```

Transducers

```
var put = function(object, pair) {
  object[pair[1]] = pair[0];
  return object;
};

transduce(
  map(function(pair) { return [pair[1], pair[0]]; }),
  put,
  {},
  { hello: 'world', transduce: 'reduce', ecmascript: 6 });

=> { 6: 'ecmascript', world: 'hello', reduce: 'transduce' }
```

Transducers / into

```
into(
  [],
  map(powerOfTwo),
  [0, 1, 2, 3, 4, 5, 6, 7]);
=> [1, 2, 4, 8, 16, 32, 64, 128]

into(
  {},
  map(swapKeysAndValues),
  { hello: 'world', transduce: 'reduce', ecmascript: 6 });
=> { 6: 'ecmascript', world: 'hello', reduce: 'transduce' }
```

Transducers

map(fn)

cat()

mapcat(fn)

filter(pred)

remove(pred)

take(number)

take-while(pred)

drop(number)

drop-while(pred)

take-nth(number)

distinct()

replace(object)

interpose(separator)

partition-by(fn)

partition-all(number)

map-indexed(fn)

keep(fn)

keep-indexed(fn)

dedupe()

Example

- Select filenames that are not directories and which size is less than 5kb.
- Extract lines that are longer than 80 chars from selected files.
- Show distributions of line sizes.

Example

```
var fs = require(‘bluebird')
  .promisifyAll(require('fs'));
var t = require('transducers-js');
var map    = t.map,
    filter = t.filter,
    mapcat = t.mapcat;
var ta = require('transduce-async');
```

https://github.com/transduce/transduce-async

Select filenames that are not directories and which size is less than 5kb

```
var filterFilenames = ta.compose(
  map(function(fileName) {
    return fs.statAsync(fileName).then(function(stat) {
      return { size: stat.size, isDirectory: stat.isDirectory(),
fileName: fileName };
    });
  }),
  filter(function(file) {
    return !file.isDirectory && file.size < 5000;
  }),
  map(function(file) {
    return file.fileName;
  })
);
```

Extract lines that are longer than 80 chars from selected files

```
var extractLines = ta.compose(
  map(function(fileName) {
    return fs.readFileAsync(fileName, 'utf8');
  }),
  mapcat(function(contents) {
    return contents.split("\n");
  }),
  filter(function(line) {
    return line.length > 80;
  })
);
```

Show distributions of line sizes

```
var sizes = ta.compose(
  filterFilenames,
  extractLines,
  map(function(line) { return line.length; }));

var putCounts = function(o, e) {
  o[e] = o[e] || 0;
  o[e]++;
  return o;
}

ta.transduce(sizes, putCounts, {}, fs.readdirSync('.'))
  .then(function(count) {
    console.log(count);
  });
```

Transducers

- Just functions… and objects
- The essence of map/filter/etc (expressed through step-functions)
- Completely decoupled from inputs and outputs
- Composable way to build algorithmic transformations

Taming the Asynchronous Beast with CSP Channels in JavaScript

September 08 2014

This is an entry in a series about rebuilding my custom blog with react, CSP, and other modern tech. Read more in the blog rebuild series.

Every piece of software deals with complex control flow mechanisms like callbacks, promises, events, and streams. Some require simple asynchronous coordination, others processing of event or stream-based data, and many deal with both. Your solution to this has a deep impact on your code.

It's not surprising that a multitude of solutions exist. Callbacks are a dumb simple way for passing single values around asynchronously, and promises are a more refined solution to the same problem. Event emitters and streams allow asynchronous handling of multiple values. FRP is a different approach which tackles streams and events more elegantly, but isn't as good at asynchronous coordination. It can be overwhelming just to know where to start in all of this.

I think things can be simplified to a single abstraction since the underlying problem to all of this is the same. I present to you CSP and the concept of *channels*. CSP has been highly influential in Go and recently Clojure embraced it as well with core.async. There's even a C version. It's safe to say that it's becoming quite popular (and validated) and I think we need to try it out in JavaScript. I'm not going to spend time comparing it with every other solution (promises, FRP) because it would take too long and only incite remarks about how I wasn't using it right. I hope my examples do a good enough job convincing you themselves.

Typically channels are useful for coordinating truly concurrent tasks that might run at the same time on separate threads. They are actually just as useful in a single-threaded environment because they solve a more general problem of coordinating anything *asynchronous*, which is everything in JavaScript.

Go-like channels

```
var ch = chan();

go(function*() {
  var val;
  while((val = yield take(ch)) !== csp.CLOSED) {
    console.log(val);
  }
});

go(function*() {
  yield put(ch, 1);
  yield take(timeout(1000));
  yield put(ch, 2);
  ch.close();
});
```

… with transducers

```
var xform = comp(
  map(function(x) {
    return x * 2;
  }),
  filter(function(x) {
    return x > 5;
  }));

var ch = chan(1, xform);
```

```
go(function*() {
  yield put(ch, 1);
  yield put(ch, 2);
  yield put(ch, 3);
  yield put(ch, 4);
});

go(function*() {
  while(!ch.closed) {
    console.log(yield take(ch));
  }
});
=> 6, 8
```

Issues

- No support for async stuff by default. transduce-async does workarounds, but something like `filter` cannot be converted to async from the outside. IE: filter-async is needed.
- No groupBy, sort, etc.

Thanks

- @fxposter
- github.com/fxposter
- blog.fxposter.org
- fxposter@gmail.com

- https://www.youtube.com/watch?v=6mTbuzafcII
- https://www.youtube.com/watch?v=4KqUvG8HPYo
- http://clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html
- http://clojure.com/blog/2012/05/15/anatomy-of-reducer.html
- http://blog.cognitect.com/blog/2014/8/6/transducers-are-coming
- http://jlongster.com/Transducers.js--A-JavaScript-Library-for-Transformation-of-Data
- http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript