Extracted from:

Seven Concurrency Models in Seven Weeks

When Threads Unravel

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

Series editor: *Bruce Tate*
Development editor: *Jacquelyn Carter*

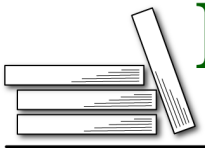
Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Bruce A. Tate (series editor)
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-65-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2014

CHAPTER 3

Functional Programming

Functional Programming is like a car powered by hydrogen fuel cells—advanced, futuristic, and not yet widely used, but it's what we'll all rely on in twenty years.

In contrast to an imperative program, which consists of a series of statements that change global state when executed, a *functional program* models computation as the evaluation of expressions. Those expressions are built from pure mathematical functions that are both first-class (can be manipulated like any other value) and side effect–free. It's particularly useful when dealing with concurrency because the lack of side effects makes reasoning about thread safety much easier. It is also the first model we'll look at that allows parallelism to be represented directly.

If It Hurts, Stop Doing It

The rules about locking that we discussed in Chapter 2, *Threads and Locks*, on page ?, apply only to data that is both shared between threads and might change—in other words *shared mutable state*. Data that doesn't change (is *immutable*) can be accessed by multiple threads without any kind of locking.

This is what makes functional programming so compelling when it comes to concurrency and parallelism—functional programs have no mutable state, so they cannot suffer from any of the problems associated with shared mutable state.

In this chapter we're going to look at functional programming in Clojure,[1] a dialect of Lisp that runs on the JVM. Clojure is dynamically typed; and if you're a Ruby or Python programmer, you'll feel right at home once you get used to the unfamiliar syntax. Clojure is not a pure functional language, but

1. http://clojure.org

in this chapter we'll be concentrating on its purely functional subset. I'll introduce the bits of Clojure that we'll be using along the way, but if you want to learn more about the language I recommend Stuart Halloway and Aaron Bedra's *Programming Clojure [HB12]*.

In day 1 we'll look at the basics of functional programming and see how it's trivial to parallelize a functional algorithm. In day 2 we'll dig deeper into Clojure's reducers framework and see how this parallelization works under the hood. Finally, in day 3, we'll switch our focus from parallelism to concurrency and create a concurrent functional web service with futures and promises.

Day 1: Programming Without Mutable State

When programmers first encounter functional programming, their reaction is often one of disbelief—that it can't be possible to write nontrivial programs without modifying variables. We'll see that it is not only possible but very often simpler and easier than creating normal imperative code.

The Perils of Mutable State

Today we're going to concentrate on parallelism. We'll construct a simple functional program and then show how, because it's functional, it's almost trivially easy to parallelize.

But first let's look at a couple of examples in Java that show why it's so helpful to avoid mutable state.

Hidden Mutable State

Here's a class that doesn't have any mutable state and should therefore be perfectly thread-safe:

FunctionalProgramming/DateFormatBug/src/main/java/com/paulbutcher/DateParser.java

```
class DateParser {
  private final DateFormat format = new SimpleDateFormat("yyyy-MM-dd");

  public Date parse(String s) throws ParseException {
    return format.parse(s);
  }
}
```

When I run a small example program that uses this class from multiple threads (you can see the source in the code that accompanies the book), I get the following:

```
Caught: java.lang.NumberFormatException: For input string: ".12012E4.12012E4"
Expected: Sun Jan 01 00:00:00 GMT 2012, got: Wed Apr 15 00:00:00 BST 2015
```

The next time I run it, I get this:

```
Caught: java.lang.ArrayIndexOutOfBoundsException: -1
```

And the next time, I get this:

```
Caught: java.lang.NumberFormatException: multiple points
Caught: java.lang.NumberFormatException: multiple points
```

Clearly the code isn't thread-safe at all, but why? It only has a single member variable, and that's immutable because it's final.

The reason is that SimpleDateFormat has mutable state buried deep within. You can argue that this is a bug,[2] but for our purposes it doesn't matter. The problem is that languages like Java make it both easy to write code with hidden mutable state like this and virtually impossible to tell when it happens—there's no way to tell from its API that SimpleDateFormat isn't thread-safe.

Hidden mutable state isn't the only thing you need to be careful about, as we'll see next.

Escapologist Mutable State

Imagine that you're creating a web service that manages a tournament. Among other things, it's going to need to manage a list of players, which you might be tempted to implement along these lines:

```
public class Tournament {
  private List<Player> players = new LinkedList<Player>();

  public synchronized void addPlayer(Player p) {
    players.add(p);
  }

  public synchronized Iterator<Player> getPlayerIterator() {
    return players.iterator();
  }
}
```

At first glance, this looks like it should be thread-safe—players is private and accessed only via the addPlayer() and getPlayerIterator() methods, both of which are synchronized. Unfortunately, it is not thread-safe because the iterator returned by getPlayerIterator() still references the mutable state contained within players—if another thread calls addPlayer() while the iterator is in use, we'll see a ConcurrentModificationException or worse. The state has *escaped* from the protection provided by Tournament.

2. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4228335

Hidden and escaped state are just two of the dangers of mutable state in concurrent programs—there are plenty of others. These dangers would disappear if we could find a way to avoid mutable state entirely, which is exactly what functional programming enables us to do.

A Whirlwind Tour of Clojure

It takes only a few minutes to get the hang of Clojure's Lisp syntax.

The easiest way to experiment with Clojure is through its REPL (read-evaluate-print loop), which you can start with `lein repl` (`lein` is the standard Clojure build tool). This allows you to type code and have it evaluated immediately without having to create source files and compile them, which can be amazingly helpful when experimenting with unfamiliar code. When the REPL starts, you should see the following prompt:

```
user=>
```

Any Clojure code you type at this prompt will be evaluated immediately.

Clojure code is almost entirely constructed from parenthesized lists called *s-expressions*. A function call that in most languages would be written `max(3, 5)` is written like this:

```
user=> (max 3 5)
5
```

The same is true of mathematical operators. Here's `1 + 2 * 3`, for example:

```
user=> (+ 1 (* 2 3))
7
```

Defining a constant is achieved with `def`:

```
user=> (def meaning-of-life 42)
#'user/meaning-of-life
user=> meaning-of-life
42
```

Even control structures are *s*-expressions:

```
user=> (if (< meaning-of-life 0) "negative" "non-negative")
"non-negative"
```

Although almost everything in Clojure is an *s*-expression, there are a few exceptions. Vector (array) literals are surrounded by square brackets:

```
user=> (def droids ["Huey" "Dewey" "Louie"])
#'user/droids
user=> (count droids)
3
```

```
user=> (droids 0)
"Huey"
user=> (droids 2)
"Louie"
```

And map literals are surrounded by curly brackets:

```
user=> (def me {:name "Paul" :age 45 :sex :male})
#'user/me
user=> (:age me)
45
```

Keys in maps are often *keywords*, which start with a colon and are very similar to symbols in Ruby or interned strings in Java.

Finally, a function is defined with `defn`, with arguments specified as a vector:

```
user=> (defn percentage [x p] (* x (/ p 100.0)))
#'user/percentage
user=> (percentage 200 10)
20.0
```

That concludes our whirlwind tour of Clojure. I'll introduce other aspects of the language as we go.

Our First Functional Program

I've said that the most interesting thing about functional programming is that it avoids mutable state, but we haven't actually seen an example yet. Let's rectify that now.

Imagine that you want to find the sum of a sequence of numbers. In an imperative language like Java, you would probably write something like this:

```
public int sum(int[] numbers) {
  int accumulator = 0;
  for (int n: numbers)
    accumulator += n;
  return accumulator;
}
```

That isn't functional because `accumulator` is mutable: it changes after each iteration of the `for` loop. By contrast, this Clojure solution has no mutable variables:

FunctionalProgramming/Sum/src/sum/core.clj

```
(defn recursive-sum [numbers]
  (if (empty? numbers)
    0
    (+ (first numbers) (recursive-sum (rest numbers)))))
```

This is a *recursive* solution—recursive-sum calls itself (recurses). If numbers is empty, it simply returns zero. Otherwise, it returns the result of adding the first (head) element of numbers to the sum of the rest (tail) of the sequence.

Although our recursive solution works, we can do better. Here is a solution that's both simpler and more efficient:

FunctionalProgramming/Sum/src/sum/core.clj
```
(defn reduce-sum [numbers]
  (reduce (fn [acc x] (+ acc x)) 0 numbers))
```

This uses Clojure's reduce function, which takes three arguments—a function, an initial value, and a collection.

In this instance, we're passing it an anonymous function defined with fn that takes two arguments and returns their sum. It's called once by reduce for each element in the collection—the first time, it's passed the initial value (0 in this case) together with the first item in the collection; the second time, it's passed the result of the first invocation together with the second item in the collection; and so on.

We're not quite done yet—we can make this code better still by noticing that + is already a function that, when given two arguments, returns their sum. We can pass it directly to reduce without creating an anonymous function:

FunctionalProgramming/Sum/src/sum/core.clj
```
(defn sum [numbers]
  (reduce + numbers))
```

So we've arrived at a solution that is both simpler and more concise than the imperative one. You'll find that this is a common experience when converting imperative code to functional.

Effortless Parallelism

So we've seen some functional code, but what about parallelism? What would we need to do to convert our sum function to operate in parallel? Very little, it turns out:

FunctionalProgramming/Sum/src/sum/core.clj
```
(ns sum.core
  (:require [clojure.core.reducers :as r]))

(defn parallel-sum [numbers]
  (r/fold + numbers))
```

Joe asks:
What If We Pass an Empty Collection to reduce?

Our final version of sum doesn't pass an initial value to reduce:

```
(reduce + numbers)
```

This might make you wonder what happens if we give it an empty collection. The answer is that it does the right thing and returns zero:

```
sum.core=> (sum [])
0
```

But how does reduce know that zero (and not, say, 1 or nil) is the right thing to return? This relies on an interesting feature of many of Clojure's operators—they know what their identity values are. The + function, for example, can take any number of arguments, including zero:

```
user=> (+ 1 2)
3
user=> (+ 1 2 3 4)
10
user=> (+ 42)
42
user=> (+)
0
```

When called with no arguments, it returns the additive identity, 0.

Similarly, * knows that the multiplicative identity is 1:

```
user=> (*)
1
```

If we don't pass an initial value to reduce, it uses the result of calling the function it's given with no arguments.

Incidentally, because + can take any number of arguments, this also means that we can implement sum with apply, which takes a function together with an vector and calls the function with the vector as arguments:

FunctionalProgramming/Sum/src/sum/core.clj
```
(defn apply-sum [numbers]
  (apply + numbers))
```

But unlike the version that uses reduce, this can't easily be parallelized.

The only difference is that we're now using the fold function from the clojure.core.reducers package (which we alias to r to save typing) instead of using reduce.

Here's a REPL session that shows what this buys us in terms of performance:

```
sum.core=> (def numbers (into [] (range 0 10000000)))
#'sum.core/numbers
sum.core=> (time (sum numbers))
"Elapsed time: 1099.154 msecs"
49999995000000
sum.core=> (time (sum numbers))
"Elapsed time: 125.349 msecs"
49999995000000
sum.core=> (time (parallel-sum numbers))
"Elapsed time: 236.609 msecs"
49999995000000
sum.core=> (time (parallel-sum numbers))
"Elapsed time: 49.835 msecs"
49999995000000
```

We start by creating a vector that contains all the integers between zero and ten million by inserting the result of (range 0 10000000) into an empty vector with into. Then we use the time macro, which prints the time taken by whatever code it's given. As is often the case with code running on the JVM, we have to run more than once to give the just-in-time optimizer a chance to kick in and get a representative time.

So, on my four-core Mac, fold takes us from 125 ms to 50 ms, a 2.5x speedup. We'll see how fold achieves this tomorrow, but before then let's look at a functional version of our Wikipedia word-count example.

Counting Words Functionally

Today we'll create a sequential implementation of word count—we'll parallelize it tomorrow. We're going to need to have three things:

- A function that, given a Wikipedia XML dump, returns a sequence of the pages contained within that dump
- A function that, given a page, returns a sequence of the words on that page
- A function that, given a sequence of words, returns a map containing the frequencies of those words

We're not going to cover the first two of these in any detail—this is a book about concurrency, not string processing or XML (see the accompanying code if you're interested in the details). We will look at how to count words, however, as that's what we'll be parallelizing.

Functional Maps

Because we want to return a map of word frequencies, we'll need to understand a couple of Clojure's map functions—get and assoc:

```
user=> (def counts {"apple" 2 "orange" 1})
#'user/counts
user=> (get counts "apple" 0)
2
user=> (get counts "banana" 0)
0
user=> (assoc counts "banana" 1)
{"banana" 1, "orange" 1, "apple" 2}
user=> (assoc counts "apple" 3)
{"orange" 1, "apple" 3}
```

So get simply looks up a key in the map and either returns its value or returns a default if the key isn't in the map. And assoc takes a map together with a key and value and returns a new map with the key mapped to the value.

Frequencies

We now know enough to write a function that takes a sequence of words and returns a map in which each word is associated with the number of times it appears:

FunctionalProgramming/WordCount/src/wordcount/word_frequencies.clj

```
(defn word-frequencies [words]
  (reduce
    (fn [counts word] (assoc counts word (inc (get counts word 0))))
    {} words))
```

This time we're passing an empty map {} as the initial value to reduce. And then for each word in words, we add one more than the current count for that word. Here's an example of it in use:

```
user=> (word-frequencies ["one" "potato" "two" "potato" "three" "potato" "four"])
{"four" 1, "three" 1, "two" 1, "potato" 3, "one" 1}
```

It turns out that the Clojure standard library has beaten us to it—there's a standard function called frequencies that takes any collection and returns a map of the frequencies of its members:

```
user=> (frequencies ["one" "potato" "two" "potato" "three" "potato" "four"])
{"one" 1, "potato" 3, "two" 1, "three" 1, "four" 1}
```

Now that we can count words, all that remains is to wire things up with the XML processing.

More Sequence Functions

To see how to do that, we need to introduce a little more machinery. First, here's the map function:

```
user=> (map inc [0 1 2 3 4 5])
(1 2 3 4 5 6)
user=> (map (fn [x] (* 2 x)) [0 1 2 3 4 5])
(0 2 4 6 8 10)
```

Given a function and a sequence, map returns a new sequence that contains the result of applying the function to each element of the sequence in turn.

We can simplify the second version slightly by using partial, which takes a function together with one or more arguments and returns a partially applied function:

```
user=> (def multiply-by-2 (partial * 2))
#'user/multiply-by-2
user=> (multiply-by-2 3)
6
user=> (map (partial * 2) [0 1 2 3 4 5])
(0 2 4 6 8 10)
```

Finally, imagine that you have a function that returns a sequence, such as using a regular expression to break a string into a sequence of words:

```
user=> (defn get-words [text] (re-seq #"\w+" text))
#'user/get-words
user=> (get-words "one two three four")
("one" "two" "three" "four")
```

As you would expect, mapping this function over a sequence of strings will give you a sequence of sequences:

```
user=> (map get-words ["one two three" "four five six" "seven eight nine"])
(("one" "two" "three") ("four" "five" "six") ("seven" "eight" "nine"))
```

If you want a single sequence that consists of all the subsequences concatenated, you can use mapcat:

```
user=> (mapcat get-words ["one two three" "four five six" "seven eight nine"])
("one" "two" "three" "four" "five" "six" "seven" "eight" "nine")
```

We now have all the tools we need to create our word-counting function.

Putting It All Together

Here's count-words-sequential. Given a sequence of pages, it returns a map of the frequencies of the words on those pages:

FunctionalProgramming/WordCount/src/wordcount/core.clj
```
(defn count-words-sequential [pages]
  (frequencies (mapcat get-words pages)))
```

It starts by converting the sequence of pages into a sequence of words with (mapcat get-words pages). This sequence of words is then passed to frequencies.

It's worth comparing this to the imperative version in the code on page ?. Again, the functional solution turns out to be significantly simpler, clearer, and more concise than its imperative equivalent.