Extracted from:

Functional Programming: A PragPub Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift

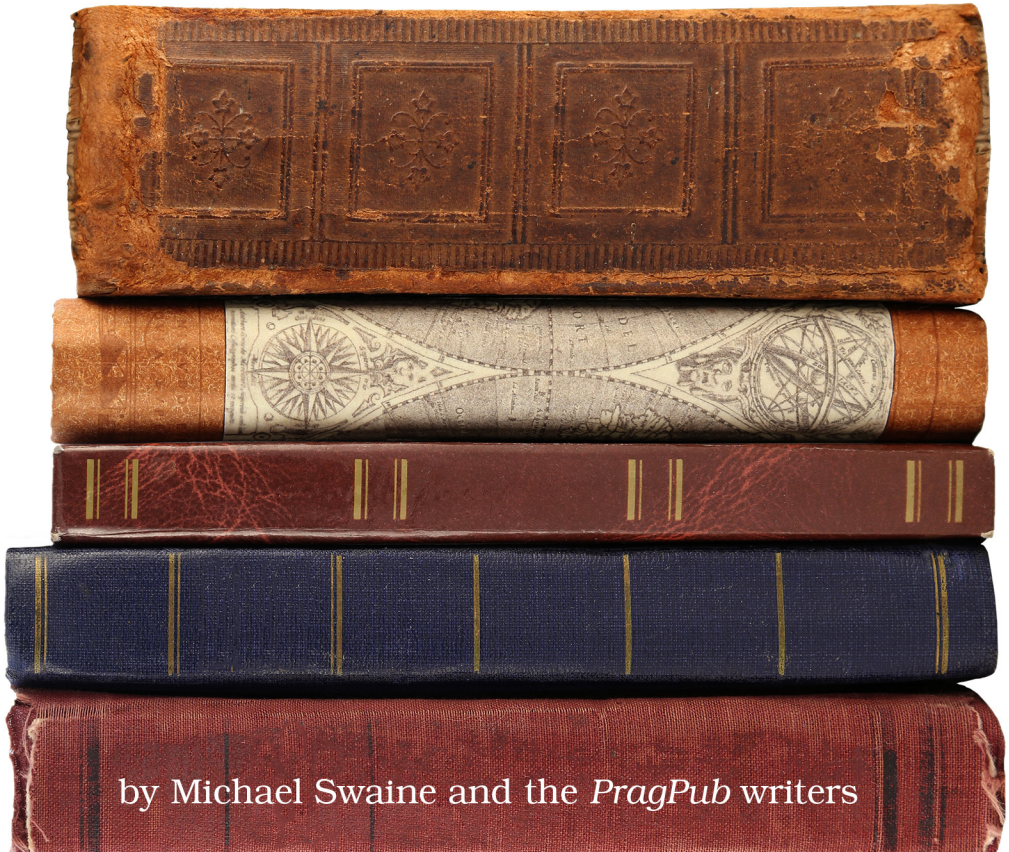
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming: A *PragPub* Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift



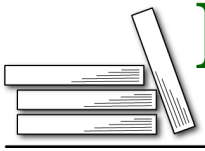
by Michael Swaine and the *PragPub* writers

Functional Programming: A PragPub Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift

Michael Swaine
and the PragPub writers

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Indexing: Potomac Indexing, LLC
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-233-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2017

CHAPTER 9

Concurrent Programming in Clojure

by Michael Bevilacqua-Linn

In the previous chapter, we learned about Clojure's take on value, state, and identity. In this chapter, we'll take things a bit further and see how Clojure's philosophy makes it ideal for concurrent programming. As we do so, we'll take a closer look at Clojure's managed references—atoms, agents, and refs—as well as the software transactional memory system they work with.

A concurrent program is one where multiple processes, or threads, may be run in parallel. The main difficulty with concurrent programming is coordinating access to shared resources. As we'll see, this can have strange effects even in seemingly simple situations.

Let's start off with a look at a simple concurrent programming problem in Java. We just want to increment counters from multiple threads running concurrently. We use two counters: the first is a plain old int, and the second is a much more impressive sounding AtomicInteger.

A "Simple" Concurrent Programming Problem

Next, we create a thread pool with five threads in it using Java's ExecutorService, and we submit 10,000 tasks to the service. Each task increments both our normal counter and our atomic counter. The full code snippet follows.

```
public class ConcurrencyExamples {

  private static int counter = 0;
  private static AtomicInteger atomicCounter =
      new AtomicInteger(0);

  public static void main(String[] args)
      throws InterruptedException {
    ExecutorService executors = Executors.newFixedThreadPool(5);
```

```
    for (int i = 0; i < 10000; i++) {
      executors.execute(new Runnable() {
        public void run() {
          counter++;
          atomicCounter.incrementAndGet();
        }
      });
    }

    // Shut down the pool and block until all tasks have executed.
    executors.shutdown();
    executors.awaitTermination(60, TimeUnit.SECONDS);

    System.out.println(
      String.format("Normal int counter: %s",
          counter));
    System.out.println(
      String.format("AtomicInteger counter: %s",
          atomicCounter));
  }
}
```

If we've handled our concurrent counter manipulation properly, both counters should have the value 10,000. As we can see from the following output, that's not the case.

```
Normal int counter: 9815
AtomicInteger counter: 10000
```

So what happened? It turns out that incrementing an integer isn't an atomic operation. It's actually broken up into multiple steps when compiled into bytecode, so the thread doing an increment can see the int it's incrementing in an inconsistent state.

To make this a bit more clear, let's take a look at a simpler example that just increments an int a single time.

```
public class JustIncAnInt {
  private static int i;

  public static void main(String[] args){
    i++;
  }
}
```

Once we compile the preceding code, we can use a command-line tool that comes with most JDKs to get a readable representation of the bytecode it's compiled to:

```
> javap -c JustIncAnInt
Compiled from "JustIncAnInt.java"
public class JustIncAnInt extends java.lang.Object{
```

```
public JustIncAnInt();
  Code:
    0:  aload_0
    1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
    4:  return

public static void main(java.lang.String[]);
  Code:
    0:  getstatic  #2; //Field i:I
    3:  iconst_1
    4:  iadd
    5:  putstatic  #2; //Field i:I
    8:  return

}
```

As we can see, that single increment is actually compiled to multiple bytecode operations. Without going too far into the weeds, the `0: getstatic #2; //Field i:I` is responsible for loading the current value of the counter. It's not until `4: iadd` that the increment is performed, and `5: putstatic #2; //Field i:I` that it's written back into the variable.

This, combined with subtleties involving when a write that one thread makes becomes visible to other threads, leaves plenty of opportunity for two threads to read and increment the variable at the same time. One solution would be to use one of Java's locking mechanisms to protect the counter variable to ensure that only one thread can access the counter at a time, and that any modifications made to it are visible when the lock is relinquished.

None of this is a surprise to any developer versed in concurrency on the JVM, but it's a nice illustration of the difficulties in concurrent programming. Programming with locks is low level and fraught with peril.

Another higher-level solution is the one that we demonstrated using `AtomicInteger`. The `AtomicInteger` class provides an `incrementAndGet`, which atomically increments an internal counter, ensuring that no thread can see it in an inconsistent state.

Two main downsides to the `AtomicInteger` solution are that it can only be used for integers, and it can only deal with coordinating changes to a single object. What if, for instance, we needed to change a map and an integer in lock step? To do so with Java, we'd probably need to fall back to locks.

Clojure's Solution

Clojure's immutable data structures, managed reference types, and software transactional memory system combine to provide a model that's both high level and much easier to use than locking concurrency. Let's dig in with a

look at an atom, which I find to be the simplest to understand of the reference types Clojure provides.

Atomic Power

As we saw in Chapter 8, *Identity, Value, and State in Clojure*, on page ?, atoms are good for managing state that's independent and to which we want to make synchronous changes. Here, independent means we can make the change without coordinating a change to any other piece of state. Synchronous means we want to block the thread that triggered the change until it's done.

We can create an atom out of any immutable data structure using `atom`, and make a change to the atom's state by passing the atom and function into the `swap!` function. The `swap!` function applies the passed in function to the value wrapped by the `atom`, swaps the old value for the new one, and returns it. Here, we create a simple atom counter, and a function to increment it.

```
(def atom-counter (atom 0))
(defn increment-atom-counter []
  (swap! atom-counter inc))
```

To de-reference an `atom`, or any of Clojure's managed references, we can use the `@`, as we show below.

```
=> @atom-counter
0
=> (increment-atom-counter)
1
```

So far, so good, but how does that help us with concurrency? When multiple threads attempt to modify an `atom` concurrently using `swap!`, Clojure first reads the current value out of the atom. Then it uses a lower-level function over atoms called `compare-and-set!`.

The `compare-and-set!` function takes an old value and a new value, and it atomically sets the `atom` to the new value only if the current value of the `atom` equals the passed in old value. If `swap!` is unable to set the value of the `atom` using `compare-and-set!`, it will continue to retry until it is successful.

Here, we've rewritten our original Java example to something similar in Clojure, which takes advantage of atoms.

```
(defn concurrent-atom-modification []
  (let [executors (Executors/newFixedThreadPool 5)
        counter (atom 0)]
    (dotimes [_ 10000]
      (.execute executors (fn [] (swap! counter inc))))
    (.shutdown executors)
```

```
    (.awaitTermination executors 60 TimeUnit/SECONDS)
    @counter))
```

Running concurrent-atom-modification sets our counter to the correct value.

```
=> (concurrent-atom-modification)
10000
```

This gives us a high-level approach to concurrency, much like the one we saw with Java's AtomicInteger, but we could wrap any immutable data structure, like a Clojure vector or map, in one so it's more general.

One consequence of the way that atoms work is that the function passed into swap! can't have any side effects, or at least no side effects that can't be repeated. This is because the function may be retried if the compare-and-set! fails the first time around.

Get an Agent

Next up, let's take a look at Clojure's agents. Like atoms, agents let us manage changes to independent state, but they're designed to do so in an asynchronous way. To modify the value referred to by an agent, we can use the send function.

Like swap!, it takes an atom to modify and a function that modifies it. Unlike swap!, it returns immediately. Operations are queued if necessary and applied to the agent serially. Here we create an atom version of our counter, and a function to increment it.

```
(def agent-counter (agent 0))
(defn increment-agent-counter []
  (send agent-counter inc))
```

Calling increment-agent-counter doesn't return the value of incrementing the counter; instead, it returns a reference to the agent. At some point in the future, the counter will be incremented. Since incrementing the counter a single time is very fast, by the time we de-reference the counter, it's already been done.

```
=> (increment-agent-counter)
#<Agent@64889c4e: 1>
=> @agent-counter
1
```

Since agents are guaranteed to be applied serially, any side effects that they execute will only execute once, unlike atoms.

Working the Ref

The final managed reference type we'll look at is the ref. These can be used when we need to make coordinated changes across more than one data structure.

In the example we'll see below, we'll take a look at a tiny system that keeps track of television series and episodes. An episode of a series is represented as a map with an id, a name, and a nested map that represents the series it's attached to. The nested series has a series id and name, as we show below.

```
{:id 42
 :name "Fragged"
 :series {:id 10 :name "Battlestar Galactica"}}
```

As episodes are added to the system, we want to populate two maps, one of episodes keyed off of episode id and one of series keyed off of series id. To add the series to our series map, we'll pick the embedded series out of an episode when it's added, and add it to the series map if it's not already there.

We'd like to ensure that we never see our data in an inconsistent state, where we've added an episode but not its corresponding series, and we'll do so using Clojure's refs and software transactional memory system.

Let's start by creating a couple of refs to hold our maps.

```
(def all-episodes (ref {}))
(def all-series (ref {}))
```

The meat of our solution involves adding episodes using assoc. Much like the swap! function does for atoms, and the send function does for agents, alter takes a reference, function, and arguments.

Unlike swap! and send!, alter must be called inside of a transaction. To create a transaction, we wrap our calls to alter inside of dosync. This acts like a database transaction, and it turns our multiple calls to alter into one atomic unit.

If one thread was to modify either map while another was in the process of doing so, one thread would win and its transaction would commit. The other would roll its transaction back and try it again. This means that, just like with the atoms we saw earlier, we need to avoid side effects in our transactions.

```
(defn add-new-episode [current-episode]
  (let [current-series (:series current-episode)]
    (dosync
      (alter all-episodes
        #(assoc % (:id current-episode) current-episode))
      (alter all-series
        #(assoc % (:id current-series) current-series)))))
```

Now we can create a few test episodes, and add them to our system.

```
(def e1 {:id 1
         :name "33"
         :series {:id 10 :name "Battlestar Galactica"}})
```

```
(def e2 {:id 2
         :name "Water"
         :series {:id 10 :name "Battlestar Galactica"}})
(def e3 {:id 3
         :name "The Target"
         :series {:id 11 :name "The Wire"}})
=> (add-new-episode e1)
{10 {:name "Battlestar Galactica", :id 10}}
=> (add-new-episode e2)
{10 {:name "Battlestar Galactica", :id 10}}
=> (add-new-episode e3)
{11 {:name "The Wire", :id 11},
 10 {:name "Battlestar Galactica", :id 10}}
```

Both of our maps contain the correct value, and they got there in a thread-safe way!

```
=> @all-episodes
{3 {:name "The Target",
    :series {:name "The Wire", :id 11},
    :id 3},
 2 {:name "Water",
    :series {:name "Battlestar Galactica", :id 10},
    :id 2},
 1 {:name "33",
    :series {:name "Battlestar Galactica", :id 10},
    :id 1}}
=> @all-series
{11 {:name "The Wire", :id 11},
 10 {:name "Battlestar Galactica", :id 10}}
```

That wraps up our look at Clojure's concurrency. We've only scratched the surface here. Clojure's API docs[1] present an excellent, if concise, description of the reference types we've seen here and they're a great place to look for more information. And to dig deeper into Clojure, have a look at Chapter 20, *Clojure's Exceptional*, on page ?, where Stuart Halloway talks about Clojure's handling of exceptions.

As you can see, Clojure and Scala both rely on the JVM and have some similarities and some differences in syntax and in their implementation of functional principles. In the next chapter, Dave Thomas introduces Elixir, a functional language with a very different approach to all of that.

1. http://clojure.github.io/clojure/