Extracted from:

Programming Clojure

The Pragmatic Programmers

Programming Clojure

Stuart Halloway

Edited by Susannah Davidson Pfalzer

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://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-33-6

ISBN-13: 978-1-934356-33-3

Printed on acid-free paper.

P2.0 printing, September 2009

Version: 2009-10-27

Chapter 6

Concurrency

Concurrency is a fact of life and, increasingly, a fact of software. There are several reasons that programs need to do more than one thing at a time:

- Expensive computations may need to execute in parallel on multiple cores (or multiple boxes) in order to complete in a timely manner.
- Tasks that are blocked waiting for a resource should stand down and let other tasks use available processors.
- User interfaces need to remain responsive while performing long-running tasks.
- Operations that are logically independent are easier to implement if the platform can recognize and take advantage of their independence.

The challenge of concurrency is not making multiple things happen at once. It is easy enough to launch a bunch of threads or a bunch of processes. Rather, the challenge is *coordinating* multiple activities happening at the same time.

Clojure provides a powerful concurrency library, consisting of four APIs that enforce different concurrency models: refs, atoms, agents, and vars.

- Refs manage *coordinated, synchronous* changes to shared state.
- Atoms manage *uncoordinated, synchronous* changes to shared state.
- Agents manage *asynchronous* changes to shared state.
- Vars manage *thread-local* state.

Refs are updated within transactions managed by Clojure's Software Transactional Memory (STM) system. Agents also have the option of interacting with STM.

Each of these APIs is discussed in this chapter. At the end of the chapter, we will develop two sample applications:

- The Snake game demonstrates how to divide an application model into immutable and mutable components.
- Continuing the Lancet example, we will add a thread-safe runonce capability to make sure each Lancet target runs only once per build.

Before we dive in, let's review the problem these APIs were designed to solve: the difficulty of using locks.

6.1 The Problem with Locks

A big challenge for concurrent programs is managing mutable state. If mutable state can be accessed concurrently, then as a programmer you must be careful to protect that access. In most programming languages today, development proceeds as follows:

- Mutable state is the default, so mutable state is commingled through all layers of the codebase.
- Concurrency is implemented using independent flows of execution called *threads*.
- If mutable state can be reached by multiple threads, you must protect that state with *locks* that allow only one thread to pass at a time.

Choosing what and where to lock is a difficult task. If you get it wrong, all sorts of bad things can happen. *Race conditions* between threads can corrupt data. *Deadlocks* can stop an entire program from functioning at all. *Java Concurrency in Practice* [Goe06] covers these and other problems, plus their solutions, in detail. It is a terrific book, but it is difficult to read it and not be asking yourself "Is there another way?"

Yes, there is. In Clojure, *immutable state* is the default. Most data is immutable. The small parts of the codebase that truly benefit from mutability are distinct and must explicitly select one or more concurrency APIs. Using these APIs, you can split your models into two layers:

- A *functional model* that has no mutable state. Most of your code will normally be in this layer, which is easier to read, easier to test, and easier to run concurrently.
- A *mutable model* for the parts of the application that you find more convenient to deal with using mutable state (despite its disadvantages).

To manage the mutable model, you can use Clojure's concurrency library. In addition, you can still use locks and all the low-level APIs for Java concurrency. If after reviewing Clojure's options you decide that Java's concurrency APIs are a better fit, use Clojure's Java interop to call them from your Clojure program. Consult *Java Concurrency in Practice* [Goe06] for API details.

Now, let's get started working with mutable state in Clojure, using what is probably the most important part of the Clojure concurrency library: refs.

6.2 Refs and Software Transactional Memory

Most objects in Clojure are immutable. When you really want mutable data, you must be explicit about it, such as by creating a mutable *reference* (ref) to an immutable object. You create a ref with this:

```
(ref initial-state)
```

For example, you could create a reference to the current song in your music playlist:

```
(def current-track (ref "Mars, the Bringer of War"))
⇒   #'user/current-track
```

The ref wraps and protects access to its internal state. To read the contents of the reference, you can call deref:

```
(deref reference)
```

The deref function can be shortened to the @ reader macro. Try using both deref and @ to dereference current-track:

```
(deref current-track)
⇒   "Mars, the Bringer of War"

@current-track
⇒   "Mars, the Bringer of War"
```

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Clojure's Home Page
http://pragprog.com/titles/shcloj
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/shcloj.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)