Clojure and concurrency

Learn about Clojure's four concurrency models

Michael Galpin

September 14, 2010

The Clojure programming language has gained a lot of attention recently. The attention, however, is not for some of the obvious reasons, such as it being a modern Lisp dialect or that it runs on top of the Java™ Virtual Machine. The features that are drawing many people to it are its concurrency features. Clojure is perhaps most well known for supporting the Software Transactional Memory (STM) model natively. STM, however, is not always the best solution for every concurrency problem. Clojure includes support for other paradigms in the form of agents and atoms. This article examines each of the concurrency approaches that Clojure provides and explores when each is most appropriate.

Getting started

In this article you will examine the Clojure programming language and its concurrency features. This is not an introductory article to Clojure, so some familiarity with it is assumed. To run the examples, you need Clojure 1.1, which in turn requires Java 1.5 or higher. In this article, Java 1.6.0_20 was used. See Resources for links to these tools. You can download the source code for this article from the Download table below.

Concurrency state of the union

For years now, software developers have been hearing about how concurrent programming was going to become the de facto way to program. The main reason given for this is that the speed of computer processors has leveled out, while the number of processors on a given computer has increased. Indeed Moore's Law has continued to be true because of these additional processors per chip. This is summed up well in Wikipedia (see Resources for a link):

"Parallel computation has recently become necessary to take full advantage of the gains allowed by Moore's law. For years, processor makers consistently delivered increases in clock rates and instruction-level parallelism, so that single-threaded code executed faster on newer processors..."
with no modification. Now, to manage CPU power dissipation, processor makers favor multi-core chip designs, and software has to be written in a multi-threaded or multi-process manner to take full advantage of the hardware."

The above paragraph would make a great call to action for this article. However, this kind of rhetoric has been prevalent for many years now, and yet a lot of developers are still happily writing single-threaded code. One big reason for this is the domination of the Internet. A significant number of new applications are web applications. Server-side web application development is mostly single-threaded programming. The web server takes advantage of the server's many cores to allow it to handle many simultaneous requests from the user, but each such request can often be handled by single-threaded code. This is a good thing, and is one of the many reasons for the success of web applications. Thus, all of those cores showing up on a user's laptop and desktop computer do not come into play for many developers.

The success of the web is not the only reason for the lack of emergence of concurrent programming. Indeed, if you examine web application development and its history, you cannot help but notice how much easier it has become for developers. From PHP and JSPs to Ruby on Rails, web development has become easier and made it possible for developers to do more and more amazing things on the web. Contrast this with concurrent programming. In the most popular programming languages (C++ and Java, for example), the constructs of concurrent programming (threads, locks) have not changed much in decades. Concurrent programming has always been difficult and continues to be difficult. Thus, it is avoided. It has become one of those things where you might have one or two gurus in your company who are the only ones that anybody trusts to do any kind of non-trivial concurrent programming.

This is where newer, more modern programming languages come into play, and Clojure is a great example. It has concurrency built into it at a low level. You do not have to deal with threads and locks. Instead you get simpler, less problematic models to work with. You can put your concentration back on your application logic and not worry so much about creating a deadlock that will bring your system to a sudden halt. Let's take a look at the concurrency constructs that are built in to Clojure.

**Clojure's flavors of concurrency**

As mentioned earlier, the most popular programming languages offer some very basic concurrency features: *threads* and *locks*. For example, Java 5 and 6 introduced numerous new utility APIs for concurrency, but most of these were either utilities built on threads and locks, such as thread pools and various types of locks; or they were data structures with better concurrency/performance characteristics. There was no change to the fundamentals of how to design concurrent programs. You still have to solve the same puzzles, and the solutions are just as brittle. You just have less boilerplate code to write.

Clojure is fundamentally different in all respects. It does not give you the usual primitives, threads and locks. Instead you get entirely different concurrent programming models that include no mention of threads or locks. Notice the use of the word *models*—plural. Clojure has four different concurrency models. Each one of these models can be considered an abstraction on top of...
threads and locks. Let’s take a look at each of these concurrency models, starting with the simplest one: **vars**.

**Thread local vars**

The simplest type of Clojure concurrency model is vars. Vars are just declarations of variables and their values. **Listing 1** shows a simple example of using vars in Clojure.

### Listing 1. Clojure vars

```
1:1 user=> (defstruct item :title :current-price)
#'user/item
1:2 user=> (defstruct bid :user :amount)
#'user/bid
1:3 user=> (def history ())(#'user/history
1:4 user=> (def droid (struct item "Droid X" 0))(#'user/droid
1:5 user=> (defn place-offer \[offer\]
    (binding \[history (cons offer history)
      droid (assoc droid :current-price (get offer :amount))\]
    (println droid history))(#'user/place-offer
1:9 user=> (place-offer {:user "Anthony" :amount 10})
{:title Droid X, :current-price 10} ({:user Anthony, :amount 10}) nil
1:17 user=> (println droid) ;there should be no change
{:title Droid X, :current-price 0} nil
```

The first thing you do in Listing 1 is declare a pair of data structures, `item` and `bid`. Next, you create a var called `history` that is just an empty list and then a var called `droid` that is an item. Next, you create a function called `place-offer`. This takes a bid and changes the `current-price` of `droid` and adds the bid to `history`. Notice that to do this you used the binding macro. This changes the thread local value of the var. So in the scope of the `place-offer` function’s execution, the values that `droid` and `history` point to will be different. However, outside of the execution, the values are unchanged. Remember that in Clojure everything is immutable by default. Binding vars simply allows you to change things in a thread local scope. If any other threads were to read the values, you would see no change. For situations where you just need to mutate state as part of the execution of some discrete task, this is a simple way to accomplish that. If you want to change state in a way that other threads will see it, then you might want to use Clojure’s **atoms**.

**Simple, synchronous atoms**

An **atom** is a variable whose state can be changed. They are very simple to use and are completely synchronous. In other words, if you call a function that changes the value of an atom, then when that function returns, you are guaranteed that all threads will see the new value. **Listing 2** shows an example of using atoms.
Listing 2. Clojure atoms

This code builds on the previous example in Listing 1. This time you redefine droid and history as atoms by using the atom function. By using the atom function, you get an atom object that is a wrapper around the initial value. Now in your new place-offer function, you change the value of droid by using the reset! function. Notice that in place you prepended droid and history with an @ symbol. This tells Clojure to de-reference the pointer and gives you the actual value. Next, you invoke the new place-offer function, and after that you can print droid and see that the value has indeed changed. Notice that in place-offer, you only changed one atom, droid. You did not change the history atom. You could certainly use reset! on it as well. However, there would be no guarantees about both changes being visible. In other words, it would be possible for one thread to see the value of droid change, but not the value of history change. To get that kind of consistency, you need coordination. You need transactions. You need refs.

Transactional refs

Clojure’s refs provide its most powerful flavor of concurrency. This is Clojure’s Software Transactional Memory (STM) implementation. Refs are similar to atoms. Often you will only require a single extra line of code, compared to atoms. The main advantage is coordination. With refs, you can change the state of multiple objects in a single transaction. The transaction will be atomic, consistent, and isolated—the ACI of ACID (no durability, since this is all in memory). The isolated property implies that any observer will either see all of the changes in the transaction, or none. This is not the case with atoms. Listing 3 shows an example of using refs.

Listing 3. Clojure refs

The code in Listing 3 is very similar to the code in Listing 2. Refs follow the same wrapper pattern you used with atoms. The place-offer function implementation begins with a call to dosync. This

Listing 2. Clojure atoms

1:21 user=> (def droid (atom (struct item "Droid X" 0)))
#'user/droid
1:22 user=> (def history (atom ()))
#'user/history
1:28 user=> (defn place-offer 
[offer]
(reset! droid (assoc @droid :current-price (get offer :amount))))
#'user/place-offer
1:33 user=> (place-offer {:user "Anthony" :amount 10})
{:title "Droid X", :current-price 10}
1:36 user=> (println @droid)
{:title Droid X, :current-price 10}
nil

Listing 3. Clojure refs

1:90 user=> (def droid (ref (struct item "Droid X" 0)))
#'user/droid
1:91 user=> (def history (ref ()))
#'user/history
1:92 user=> (defn place-offer [offer]
(dosync
(ref-set droid (assoc @droid :current-price (get offer :amount)))
(ref-set history (cons offer @history)))
)
1:97 user=> (place-offer {:user "Tony" :amount 22})
{:title Droid X, :current-price 22}
1:99 user=> (println @droid @history)
{:title Droid X, :current-price 22} {"user" Tony, :amount 22}
nil
function wraps a transaction. It provides the coordination that I referred to earlier. It allows you to change both droid and history and know that there will not be any dirty reads of the data. Just like with atoms, you can de-reference and print the values after the execution of the function and see that the values have changed.

Now you might be wondering about how exactly the STM works here. What happens if one thread calls the place-offer function with an amount of 25 at the same time as another thread calling it with an amount of 22? Clojure will make sure that values do not change in the middle of a transaction. So if a transaction reached the end of the dosync block, and the STM sees that another transaction has finished since the current one started, then the current one will be rolled back and run again. This makes it very important that only pure functions—that is, functions with no side-effects—are used as part of a transaction, as the function may be run multiple times. Clojure uses very high performance persistent data structures to make this kind of transactions/rollbacks efficient.

If you wanted to make sure that a new offer can only be taken if its amount is higher than the previous offer, then all you have to do is add a validation function to the ref declaration. In this case, if a change has been detected during a transaction, it will be rolled back and restarted. Now if the validation check fails, then the transaction will be aborted.

The key to using Clojure's STM is wrapping things inside the dosync function. An astute observer might point out that this is very similar to wrapping code inside a synchronized block, or inside a lock acquire/release flow. Of course, those traditional concurrency mechanisms are notoriously difficult. Clojure is more straightforward. If you are going to change state, then you must use dosync. You cannot change the state of a ref outside of a dosync. Further, Clojure's transactions can be composed. You can be inside a dosync block and call another function that also has a dosync block. You don't have to figure out some kind of lock to be shared by the functions. You also don't have to worry about deadlocks. Refs and atoms are both synchronous functions. If you don't need synchronous changes to your state, then agents can provide some advantages.

**Easy, asynchronous agents**

Often you need to change state, but you don't need to wait for it to be changed or you don't care about the ordering of changes if multiple threads can make changes. This is a common pattern, and Clojure provides a programming model to address this: agents. Listing 4 shows an example of using agents.
Listing 4. Clojure agents

Once again you start off by wrapping the initial values of droid and history by using the agent function. Then you define a new version of place-offer. This time you cannot just directly alter the values behind the agents. Instead you use the send function. This function takes an agent and another function as its parameters. The second function is another function that will be applied to the value of the agent. The resulting value will be used to replace the value of the agent. In Listing 4 an anonymous function has been used to pass to send. It should be noted that both atoms and refs also support this kind of semantics, where a function is passed and used to update the state. Next, notice that you used the await function. This blocks the thread until the agent has executed the function that was sent to it. It is a good way to make sure that the changes you want have actually been applied. Otherwise the asynchronous nature of agents would mean that you cannot be sure if the function sent has been applied to it.

Conclusion

This article has shown you each of Clojure's concurrency models. There are many different concurrency problems out there, but many of them will map nicely to one of Clojure's models. In such a case, you will have a much easier time solving the problem by taking advantage of Clojure's capabilities. For the times when your problem does not map well, you can take advantage of Clojure's interoperability with Java and use Java's threads and locks instead. This is what makes Clojure a language that you should keep in mind whenever dealing with any kind of task that will depend heavily on concurrency.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>auctions.clj.zip</td>
<td>1KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2010

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)