The State Machines of core.async

Under the covers `core.async` creates a state machine to turn synchronous / blocking looking code into asynchronous / non-blocking code. Coming from a C# background I knew this state machine method was used with the async / await syntax added in .NET 4.5. What I didn't know was what this actually meant. Since this was language level syntax it was opaque to me in how it was implemented.

Enter `core.async` which is implemented as a library due to the magic of macros. I initially wanted to see how the macro works, but considering I've never written a macro (don't spread that around please) I put that off for another day. Instead I simply expanded the macro in an attempt to see what this whole state machine thing was about.

The State Machine

I'm going to use a pretty trivial function to examine the state machine.

```
(defn fake-search [kind]
  (fn [c query]
    (go
      (<! (timeout (rand-int 100)))
      (>! c [kind query]))))
```

If we expand the `go` macro from this function we can get the state machine function (among other things). I've pulled the state machine function out of the expansion and included it below shortening some namespaces and changing some names for readability. Don't worry about trying to read it all!

```
(fn state-machine
  ([]
   (ioc-macros/aset-all! (java.util.concurrent.atomic.AtomicReferenceArray. 6)
                         0 state-machine
                         1 1))
  ([state_3730]
   (let
     [old-frame__2202__auto__ (clojure.lang.Var/getThreadBindingFrame)]
     (try
       (clojure.lang.Var/resetThreadBindingFrame
         (ioc-macros/aget-object state_3730 3))
       (loop []
         (let
           [result (case (int (ioc-macros/aget-object state_3730 1))
                     3 (let [inst_3728 (ioc-macros/aget-object state_3730 2)
                             state_3730 state_3730]
                         (ioc-macros/return-chan state_3730 inst_3728))
                     2 (let [inst_3725 (ioc-macros/aget-object state_3730 2)
                             inst_3726 (vector kind query)
                             state_3730 (ioc-macros/aset-all! state_3730 5 inst_3725)]
                         (ioc-macros/put! state_3730 3 c inst_3726))
                     1 (let [inst_3722 (rand-int 100)
                             inst_3723 (timeout inst_3722)
                             state_3730 state_3730]
                         (ioc-macros/take! state_3730 2 inst_3723)))]
           (if (identical? result :recur)
             (recur)
             result)))
       (finally
         (clojure.lang.Var/resetThreadBindingFrame
           old-frame__2202__auto__))))))
```

There is a lot here. There are three things I want to point out. The first is the `java.util.concurrent.atomic.AtomicReferenceArray`. This is where all of the state of the state machine is stored. It has the following special indices:

- 0 : FN-IDX : the state machine function
- 1 : STATE-IDX : the current state
- 2 : VALUE-IDX : the value the expression in the go block evaluate to
- 3 : BINDINGS-IDX : the captured-bindings
- 4 : USER-START-IDX : the channel that will be returned by the go block

The `aset-all!` function mutates the array with index / value pairs while the `aget-object` gets the value for an index from the array.

The second is that the function either accepts a state (the above array) or if none is given initializes the state.

```
([]
 (ioc-macros/aset-all! (java.util.concurrent.atomic.AtomicReferenceArray. 6)
                       0 state-machine
                       1 1))
```

Here we create a new array with 6 indices, set the 0th index to the state machine function and set the 1st index to 1 which stores the current state of the state machine (we start at state 1).

If instead the function is given a state array it does a `case` statement off of the current state value executing that state.

```
(let
  [result (case (int (ioc-macros/aget-object state_3730 1))
            3 (let [inst_3728 (ioc-macros/aget-object state_3730 2)
                    state_3730 state_3730]
                (ioc-macros/return-chan state_3730 inst_3728))
            2 (let [inst_3725 (ioc-macros/aget-object state_3730 2)
                    inst_3726 (vector kind query)
                    state_3730 (ioc-macros/aset-all! state_3730 5 inst_3725)]
                (ioc-macros/put! state_3730 3 c inst_3726))
            1 (let [inst_3722 (rand-int 100)
                    inst_3723 (timeout inst_3722)
                    state_3730 state_3730]
                (ioc-macros/take! state_3730 2 inst_3723)))]
```

This state machine has three states: 1, 2, and 3. Since the initial state is 1 the first thing we'll do in this state machine is the `take!` function at the bottom.

So that defines the state function, but we still need to get it rolling. That is handled right after the function is defined.

```
(let [f__2332__auto__ (fn state-machine
                        ; ... see above ...
                        )
      state__2333__auto__ (-> (f__2332__auto__)
                              (ioc-macros/aset-all!
                                ioc-macros/USER-START-IDX c__2330__auto__
                                ioc-macros/BINDINGS-IDX captured-bindings__2331__auto__))]
  (ioc-macros/run-state-machine state__2333__auto__))
```

After the state machine function is defined we call it with no arguments which creates and initializes the array (with the state machine function and the starting state) and then returns it. The state machine array then has two more indices set. The user start index is 4 and it gets set to the channel that the `go` block will return (remember go is an expression that returns its last value on a channel).

Then the binding index (3) is set with bindings captured outside of this code snippet using `Var/getThreadBindingFrame`. It is kinda cool that you can just grab and save bindings like this.

At this point the array looks like:

- 0 : FN-IDX : the state machine function

- 1 : STATE-IDX : 1, the current state
- 2 : VALUE-IDX : nil, the value the expression in the go block evaluate to
- 3 : BINDINGS-IDX : the captured-bindings
- 4 : USER-START-IDX : the channel that will be returned by the go block

And finally, we `run-state-machine` passing in the state machine array.

```
(defn run-state-machine [state]
  ((aget-object state FN-IDX) state))
```

`run-state-machine` is pretty unremarkable. We grab the function out of the array and then execute it passing the array as the only argument.

So after all is said and done we get back to the `take!` function being called as the first state in the state machine.

`take!`

Here is our first state in the state machine.

```
(let [inst_3722 (rand-int 100)
      inst_3723 (timeout inst_3722)
      state_3730 state_3730]
  (ioc-macros/take! state_3730 2 inst_3723))
```

The original code this corresponds to is `(<! (timeout (rand-int 100)))`.

In this call `state_3730` is the state machine’s mutable array and `inst_3723` is the channel resulting from `(timeout (rand-int 100))`. The middle argument, 2, is the next state following this one.

And now for `take!`’s implementation.

```
(defn take! [state blk c]
  (if-let [cb (impl/take! c (fn-handler
                              (fn [x]
                                (aset-all! state VALUE-IDX x STATE-IDX blk)
                                (run-state-machine state))))]
    (do (aset-all! state VALUE-IDX @cb STATE-IDX blk)
      :recur)
    nil))
```

There are two `take!`’s. There is the channel’s take, `impl/take!` above, and then the state machines take, the function shown.

There are two contexts in which `take!` executes:

- There **IS** a value on the channel to dequeue
- There **IS NOT** a value on the channel to dequeue

If there **IS NOT** a value to dequeue then the the channel’s `impl/take!` returns `nil` so the `if-let` is false and the whole `take!` expression evaluates to `nil`.

If there **IS** a value to dequeue the channel returns that value boxed (meaning we have to deref it to get the value). We mutate the state machine’s array to update the `VALUE-IDX` (2) to whatever we got off the channel and the state machine’s state to the next state, which would be 2 in this example, and then return `:recur`.

We need to look back at the state machine to see how it responds to both of these outcomes.

```
(loop []
  (let
    [result (case (int (ioc-macros/aget-object state_3730 1))
              3 (let [inst_3728 (ioc-macros/aget-object state_3730 2)
                      state_3730 state_3730]
                  (ioc-macros/return-chan state_3730 inst_3728))
```

```
            2 (let [inst_3725 (ioc-macros/aget-object state_3730 2)
                    inst_3726 (vector kind query)
                    state_3730 (ioc-macros/aset-all! state_3730 5 inst_3725)]
                (ioc-macros/put! state_3730 3 c inst_3726))
            1 (let [inst_3722 (rand-int 100)
                    inst_3723 (timeout inst_3722)
                    state_3730 state_3730]
                (ioc-macros/take! state_3730 2 inst_3723)))]
    (if (identical? result :recur)
      (recur)
      result)))
```

So `result` is either `nil` (nothing to dequeue) or `:recur` (we did dequeue something). In the `nil` case we exit the loop and return `nil`. The state machine function exits and the thread is released. In the `:recur` case we recurse on the loop. When we get `:recur` we also advanced the state machine to the next state. So if there is something to `take!` we immediately execute the next state (which is state 2) and keep control of the thread.

So this is a big part of how the state machines work. The last missing bit is when there is nothing to dequeue we want to fake block on the queue but here the state machine function just plain exits and releases the thread. It is up to something else to kick this state machine back up when a value arrives on the channel.

This something is the channel.

```
(impl/take! c (fn-handler
                (fn [x]
                  (aset-all! state VALUE-IDX x STATE-IDX blk)
                  (run-state-machine state))))
```

When we call `impl/take!` on the channel we give it a `Handler` which includes a callback that takes a single argument. In this case the callback does two things. It mutates the state machine's array just like the case where there was a value to dequeue – the `VALUE-IDX` is updated with the argument and the state machine is advanced to the next state. The second thing it does is re-run the state machine.

This covers a lot of the magic behind the state machines and how things are done non-blocking. Under the hood its still all callbacks, it is just that the callback hell is managed by the channels / library and not by humans.

So in the simple case the state machine is able to execute from start to finish without giving up the thread – every channel has something to take or something ready for a put and we immediately recurse through the state machine until end. In other cases the state machine exits, the thread is released to do other things, and it is up to channels to restart the state machine when values are ready to dequeue.

After this state has completed – either immediately or via a callback – the state machines array is now:

- 0 : FN-IDX : the state machine function
- 1 : STATE-IDX : 2, the current state
- 2 : VALUE-IDX : the value received from `timeout` channel (`nil` since it closes on timeout)
- 3 : BINDINGS-IDX : the captured-bindings
- 4 : USER-START-IDX : the channel that will be returned by the go block

put!

So now we are experts on `take!` and the state machine in general. Lets look at the next state which is a `put!`.

```
(let [inst_3725 (ioc-macros/aget-object state_3730 2)
      inst_3726 (vector kind query)
      state_3730 (ioc-macros/aset-all! state_3730 5 inst_3725)]
  (ioc-macros/put! state_3730 3 c inst_3726))
```

Again, `state_3730` is the state machine's mutable array, 3 is the next state, and `inst_3726` is the evaluation of `(vector kind query)`. The last bit is the `c` in there. Lets look back at the original function.

```
(defn fake-search [kind]
  (fn [c query]
```

```
    (go
      (<! (timeout (rand-int 100)))
      (>! c [kind query]))))
```

The `c` in the state machine is just the channel `c` from this function. Whenever the state machine is executed it makes sure all the vars are as the function expects them to be. Again, this blows my mind.

So the first parts of the `put!` call are the state machine array and the next state. The last bits are the channel, `c`, and the value we want to put on that channel, `(vector kind query)`.

```
(defn put! [state blk c val]
  (if-let [cb (impl/put! c val (fn-handler (fn []
                                              (aset-all! state VALUE-IDX nil STATE-IDX blk)
                                              (run-state-machine state))))]
    (do (aset-all! state VALUE-IDX @cb STATE-IDX blk)
      :recur)
    nil))
```

This looks pretty familiar. The only difference between this and `take!` is that the callback in the `Handler` does not take an argument and the value put into the state machine is `nil` (where in `take!` it is the value dequeued from the channel).

So again either we immediately put onto the channel successfully and `:recur` or we fake block and the channel is responsible for executing the callback when the channel is ready to enqueue a value.

When I first read about `core.async` it wasn't immediately obvious to me that putting onto a channel is also blocking.

```
(let [c (chan)]
  (go
    (>! c :foo)
    (prn "this never runs!")))
```

This go routine will never finish – no one ever takes from `c`. This is also where buffers come in – this won't block if there is room in a buffer to place the value. But by default channels don't have a buffer and the put blocks until another thread takes from the channel.

Again, that was probably obvious to most people!

return-chan

And finally, on to our last state. One difference between `go` (the language) and `core.async` is that `go` is an expression in clojure. It returns a channel which has the value of the last expression put on it and immediately closed. This is the job of our last state.

```
(let [inst_3728 (ioc-macros/aget-object state_3730 2)
      state_3730 state_3730]
  (ioc-macros/return-chan state_3730 inst_3728))
```

The `VALUE-IDX` (2) in the state machine array has been keeping track of the last value as it goes through the state machine and the `USER-START-IDX` (4) has the channel returned from the go expression. `return-chan` handles the rest. In this case `inst_3728` is the value from the state array (which is `nil` since the last expression was a put) and again `state_3730` is the state machine's array.

```
(defn return-chan [state value]
  (let [c (aget-object state USER-START-IDX)]
    (when-not (nil? value)
      (impl/put! c value (fn-handler (fn [] nil))))
    (impl/close! c)
    c))
```

This is a pretty straight forward function. We grab the channel out of the state machine's array. We check for `nil` since you cannot put `nil` into a channel. If not `nil` we put onto the channel and give a noop callback (there is nothing left to do after this state). In every case we close and return the channel.

do-alts!

The above functions, `take!`, `puts!`, and `return-chan` cover my simple state machine. One thing we left out is selection from multiple channel with `alts`.

```
(go
  (let [[v ch] (alts! [c1 c2])]
    (println "Read" v "from" ch)))
```

This is a pretty simple go routine which selects from a pair of channels and prints out a message. If we expand the macro the first state looks like this:

```
(let [inst_3786 (vector c1 c2)
      state_3793 state_3793]
  (ioc-alts! state_3793 2 inst_3786))
```

We have an additional function here, `ioc-alts!` which takes the state machine array, next state, and a vector of channels. Lets take a look at it.

```
(defn ioc-alts! [state cont-block ports & {:as opts}]
  (ioc/aset-all! state ioc/STATE-IDX cont-block)
  (when-let [cb (clojure.core.async/do-alts
                  (fn [val]
                    (ioc/aset-all! state ioc/VALUE-IDX val)
                    (ioc/run-state-machine state))
                  ports
                  opts)]
    (ioc/aset-all! state ioc/VALUE-IDX @cb)
    :recur))
```

So this is written a little differently than `take!` and `put!` but it is actually pretty much the same thing. We advance the state of the state machine immediately instead of in the callback. The callback then takes care of setting the return value of the state machine.

Instead of dealing directly with a channel via `impl/take!` and `impl/put!` we delegate to `do-alts!` giving it a callback function. If this returns `nil` the expression evaluates to `nil`, otherwise we get `:recur` and continue the state machine.

For the last piece lets take a look at `do-alts`. Don't try to read it all at once! I'll go over it in detail.

```
(defn do-alts
  "returns derefable [val port] if immediate, nil if enqueued"
  [fret ports opts]
  (let [flag (alt-flag)
        n (count ports)
        ^ints idxs (random-array n)
        priority (:priority opts)
        ret
        (loop [i 0]
          (when (< i n)
            (let [idx (if priority i (aget idxs i))
                  port (nth ports idx)
                  wport (when (vector? port) (port 0))
                  vbox (if wport
                         (let [val (port 1)]
                           (impl/put! wport val (alt-handler flag #(fret [nil wport]))))
                         (impl/take! port (alt-handler flag #(fret [% port]))))]
              (if vbox
                (channels/box [@vbox (or wport port)])
                (recur (inc i))))))]
    (or
      ret
      (when (contains? opts :default)
        (.lock ^Lock flag)
```

```
        (let [got (and (impl/active? flag) (impl/commit flag))]
          (.unlock ^Lock flag)
          (when got
            (channels/box [(:default opts) :default])))))))
```

So this is a lot of code, and we call channels ports for some reason, but it really isn't too bad. Again, there are two broad cases: there **IS** something ready to dequeue or enqueue or there **IS NOT** something ready to dequeue or enqueue.

Starting with the `let` statement we create a flag using `alt-flag`. At a high level you can think of a flag as a shared lock / coordinator between all of the potential callback's we are going to give to each channel in the `alts!` expression. In our example we have `c1` and `c2` and when we take from `c1` we want to make sure we don't also take from `c2`. The `flag` handles this coordination for us. How this is done is pretty cool and I hope to cover it in a later post.

The first case we want to check for is the case in which a channel **IS** ready to enqueue or dequeue a value. This is done by looping over all of the channels. The order this is done depends on whether a priority flag is passed which would make the order the channels are given in the vector the order they are checked. Otherwise it is done randomly. The `loop` expression is the construct for doing this check.

```
(loop [i 0]
  (when (< i n)
    (let [idx (if priority i (aget idxs i))
          port (nth ports idx)
          wport (when (vector? port) (port 0))
          vbox (if wport
                 (let [val (port 1)]
                   (impl/put! wport val (alt-handler flag #(fret [nil wport]))))
                 (impl/take! port (alt-handler flag #(fret [% port]))))]
      (if vbox
        (channels/box [@vbox (or wport port)])
        (recur (inc i))))))
```

This is basically a for loop in clojure. We start at 0 and loop while `i` is less than `n` (the number of channels). On each iteration the channel (port) we check is either the index value `i` or the ith random integer of the `idxs` array (`idx (if priority i (aget idxs i))`). If the channel is a vector then it is a put operation with a channel and value pair such as `[c3 "foo"]` otherwise it is a take and just a channel such as `c1`.

Once the channel is sorted we attempt the `put!` or `take!` operation on the channel. Like before this either returns `nil` because the channel is not ready to enqueue or dequeue a value or a boxed value if the channel is ready. The `Handler` given to both `put!` and `take!` is not an `fn-handler` but an `alt-handler` which you should notice also takes the `flag` as well as a callback function. Again, this `flag` is required for coordination.

So if we get a value from `take!` or `put!` we are done – a channel operation was ready and completed. We break from the loop and return the pair of value and channel (which is boxed). Otherwise we advance the index and loop again. If we loop through all of the channels and none are ready then the loop evaluates to `nil`.

```
(or
  ret
  (when (contains? opts :default)
    (.lock ^Lock flag)
    (let [got (and (impl/active? flag) (impl/commit flag))]
      (.unlock ^Lock flag)
      (when got
        (channels/box [(:default opts) :default])))))
```

Here `ret` is the value of the loop expression. If it has a value the `or` expression returns it. If it is `nil` then we have two options: we wait for a channel operation to complete or if a default value was passed we return that default value.

If we have a default value we lock the flag, check if its active, and if so `commit` the flag. We have to lock the flag in order to check that it is active. This is because we are in a multi-threaded environment and it is possible that since we last checked the channels one of them has actually completed a take or put operation. When that happens the flag is set inactive. So if the flag is still active then channels are still waiting and none of the callbacks have been called so we should return the default value.

Since we are going to return a default value we want to cancel all of the callback's waiting on the channels. This is where `commit` comes in. When we commit the flag we make it inactive which also makes all of the callback's on the channels from this `alts!` expression inactive fulfilling the flag's coordination responsibility – a channel will never execute a callback on an inactive `Handler`.

Finally we unlock the flag and return the default value (again, boxed).

If there is no default value we simply return `nil` and all of the callback's on each channel remain active.

Just so you remember the callback functions look like:

```
(fn [val]
  (ioc/aset-all! state ioc/VALUE-IDX val)
  (ioc/run-state-machine state))
```

The state machine's state was already advanced, so it just updates the value and runs the machine.

One final detail is that when a callback that is part of this `alts!` expression is committed by a channel then this also commits the flag and the rest of the callbacks become inactive. This is how we ensure that each `alts!` expression only puts or takes from a single channel.

Conclusion

The first conclusion is that this post was long. If you read this, you are awesome, thanks.

The second conclusion is that macros are pretty cool. Even if I'm not smart enough to write this macro the ability to expand it and view the `core.async` source lets me understand how this stuff works. It doesn't remain in the realm of compiler writers (of which I am not a member). So thanks for that clojure.

Finally I used really simple functions here which resulted in simple state machines. I'm not sure how much of the state machines functionality this covers – there might be more advanced features my simple functions did not exercise.

7 Comments Huey Petersen Login

Sort by Best Share Favorite

Join the discussion…

Timothy Baldridge · 9 months ago

I have a gigantic grin on my face right now. This is spot on! The blog post I should have written, and never did. The parts of the state machine code that you didn't exercise basically boil down to loop, if and case. Nothing earth-shattering there, the are simply special block terminations that manipulate the STATE-IDX and then :recur.

Once again, awesome work!

9 | · Reply · Share ›

hueypetersen Mod → Timothy Baldridge · 9 months ago

Thanks!

I get nervous when I post stuff because I risk being wrong (on the internet no less!), so it feels good to know I lucked out :). Thanks for your work on the library, I've really enjoyed digging into it.

6 ^ | v • Reply • Share ›

cliftonk • 9 months ago

Thanks for going over the macroexpansions. Great article.

7 ^ | v • Reply • Share ›

Weng Fu • 9 months ago

Please post code examples in a language that is relevant to the real world. I suggest you use Microsoft Visual Basic 6 as a benchmark.

1 ^ | v • Reply • Share ›

James Gatannah • 6 months ago

Nice! Thank you for digging into this. I'm pretty amazed that it isn't much, much worse.

^ | v • Reply • Share ›

Howard M. Lewis Ship • 7 months ago

This was truly informative; I also learned that per-thread bindings visible when the go block executes are maintained while the go block executes ... very handy for the project I'm working on (alas!).

^ | v • Reply • Share ›

Nimalan Mahendran • 9 months ago

This is a great article; thanks for putting it together!

^ | v • Reply • Share ›

You can reach me at eyston@gmail.com or via twitter @hueypetersen.