core.async

Train you a core.async for great concurrency

Motivation

- core.async is a tool for encoding asynchronous behavior that does not rely on callbacks or events

- callbacks do not encourage sustainable software development, can easily slip into “callback hell”

- events are a way to loosely couple callbacks from their callers, but do not solve any other fundamental problems with callbacks

Foundational Basis

- core.async is an implementation of Communicating Sequential Processes
- Hoare first described the idea in 1978
- Core concepts are messages and channels
- Passing messages between processes allows them to share information

Core core.async operations

- Sending data (via: put!, >!, >!!)
- Receiving data (via: take!, <!, <!!)
- Channel and buffer manipulation (via: chan, buffer, map, filter, et al)
- Process initialization (via: go, thread)

When to use?

- To express concurrent calculation of data pipelines
- To express reactive behavior to some external stimulus
- To incorporate the passage of time into program behavior

When to shelve?

- Absence of indicators
- Backward error propagation is important
- In essentially synchronous computation

Costs / Benefits: Conceptual

- Costs: reified, managed join points between processes, source obfuscation of go blocks

- Benefits: single join point to express data sources and sinks, explicit incorporation of back pressure

Computational Costs/ Benefits

- Costs: Managed thread pools, channels incur costs for multiple writers/readers universally, potential to deadlock when channels form closed cycles
- Benefits: Convenient to start new async processes, excellent abstraction over multiprocess communication, will help you warm up all of those cores! Managed thread pool of go blocks lighter weight than managing threads if you keep many processes active.

Alternatives

- Lamina: Same conceptual foundation, different interface (I prefer core.async), no ClojureScript impl

- Pulsar: Actor model but also uses channels and lightweight processes. Includes a core.async similar api layer at co.paralleluniverse.pulsar.async, also lacks ClojureScript impl.

Use

- Channel creation

(chan <buf or n>)

- buf is a buffer as created with buffer

- n, a number, will implicitly create a new fixed buffer of cardinality n

Buffers

- Enforce back pressure in the queuing system, three types out of the box w/ core.async

- Fixed buffers (cause puts to block when buffer has n elements)

- Dropping buffers (puts always complete immediately, newest items are discarded when buffer has n elements)

- Sliding buffers (puts always complete immediately, oldest items are discarded when buffer has n elements)

Writing

(put! <ch> <val> <callback>)

- Writes val to ch. Callback will be invoked once val is read from ch. All other writes are in terms of put!

(>! <ch> <val>)

(>!! <ch> <val>)

- Write val to ch, semantically blocking until the operation completes. >! can only be used in go blocks, and pauses the go process until val is read. >!! will pause the executing thread until the write completes.

Reading

(take! <ch> <callback>)

- Reads a val from ch, invoking callback with the read val as its first and only argument.

(<! <ch>)

(<!! <ch>)

- Reads a val from ch, returning the read value. These likewise semantically block. <! can only be used in go blocks, pausing their go process until a value can be read. <!! will pause the executing thread until the read completes.

Go processes

- Go processes have lots of interesting magic baked into them:
    - Go macro analyzes the body to create a new code structure, reordering the body expressed as a series of blocking operations into a sequence of callbacks using put! and take!
    - Re-ordered body is then placed in a special executor pool that runs all unblocked processes that it can until all processes are blocking on some input. With sufficient input, it will never stop!
    - All of this re-writing and restructuring will destroy some source code level information. Keep go bodies simple! Focus on channel io only, breaking involved logic out to fns of the values read.

Go

```
(go

  (loop [v (<! ch1)]

    (println v)

    (>! ch2 (inc v))

    (recur (<! ch1))))
```

Go cont.

- Repeatedly reads vals from ch1, prints them, and then writes the vals to ch2.

- Vals are not written to ch2 faster than they can be read from ch1.

- Processing happens in a managed thread, governed both by how fast values are provided on ch1 and how fast they are consumed from ch2.

- If any operation cannot complete immediately, the process is shelved so that other processes that are not blocked can execute in the thread pool.

All of the fundamentals

- Everything happens in terms of channels and operations against them.
- Every channel implicitly has at least 2 buffers, possibly adding a third buffer:
    - A buffer of readers, waiting to receive values.
    - A buffer of writers, waiting to write values.
    - Optionally, a buffer of values, waiting to be read.
- Only one of the buffer of readers and buffer of writers may be non-empty. The value buffer may only be non-empty if the reader buffer is empty.
- All of this explicit buffering makes explicit the fact that this queuing system is finite and limited, and can be overwhelmed if not managed to its workload.

Multiple abstractions built on top of channels

- As collections:
  - (to-chan coll) returns a channel that provides each element of coll.
  - map<, filter<, remove<, mapcat<, reduce all defined against channels, returning a channel that produces values consistent with the collection semantics

Abstractions cont.

- As broadcasting queues:
  - (pub <ch> <topic-fn>) creates a pub/sub mechanism. topic-fn should accept vals written to ch, and return a value that will be treated as the topic. pub will return a publication instance
  - (sub <pub> <topic> <ch>) will subscribe ch to pub. Each value which matches topic will be written to ch.
- Note! EVERY subscriber must accept a value through the pub before ANY subscriber may be offered the next value!

Abstractions cont.

- As audio tech:
    - (mix <ch>) returns a new mixer, which will mix its inputs and output them on ch.
    - (admix <mix> <ch>) adds ch to pre-existing mix as an input.
    - (unmix <mix> <ch>) removes it
    - Can solo, pause, or mute with (toggle <mix> <state-map>)

Closing our channel…

- core.async is a tool for concurrent processes to communicate with each other
- It's built out of a small set of primitives that undergird a broader set of abstractions
- Allows for many to many interactions between processes mediated by channels
- Effectively minimizes the quantity of blocking threads at any given time with a sound foundational model.

Further reading

- API Reference at: http://clojure.github.io/core.async/api-index.html

- Main source at: https://github.com/clojure/core.async/blob/master/src/main/clojure/clojure/core/async.clj

- Interesting comparison at: http://adambard.com/blog/clojure-concurrency-smorgasbord/

- David Nolen describes why core.async is important for JS: http://swannodette.github.io/2013/07/12/communicating-sequential-processes/