Inside core.async
Channels

Rich Hickey
Warning!

- Implementation details
- Subject to change
The Problems

- Single channel implementation
- For use from both dedicated threads and go threads simultaneously, on same channel
- alt and atomicity
- multi-read/write
- concurrency
SPI

Channel

impl/put! [val handler]

impl/take! [handler]
Anatomy

Channel

user-specified data buffer

puts (fifo)  data

buffer(data)

closed?

takes (fifo)

internal linked queues of handlers
Invariants

• Never pending puts and takes
• Never takes and anything in buffer
• Never puts and room in buffer
• take! and put! use channel mutex
• no global mutex
put!

Channel

<table>
<thead>
<tr>
<th>puts (fifo)</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>takes (fifo)</td>
<td></td>
</tr>
<tr>
<td>buffer(data)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>closed?</td>
<td></td>
</tr>
<tr>
<td>completes handler</td>
<td></td>
</tr>
</tbody>
</table>
put! - windowed buffers

Channel

<table>
<thead>
<tr>
<th>puts (fifo)</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sliding buffer(data)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Channel

dropping buffer(data)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>takes (fifo)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

impl/put!
[val handler]

dropped
close!

- all pending takes complete with nil (closed)
- subsequent puts complete with nil (already closed)
- subsequent takes consume ordinarily until empty
  any pending puts complete with true
takes then complete with nil
Queue Limits

- puts and takes queues are not unbounded
- 1024 pending ops limit
- will throw if exceeded
- not for buffering, use buffers/windowing
alt(s!!)

- attempts more than one op
- on more than one channel
- without global mutex
  nor multi-channel locks
- exactly one op can succeed
alt implications

- registration of handlers is not atomic
- completion might occur before registrations are finished
  or any time thereafter
- completion of one alternative must ‘disable’ the others atomically
- cleanup
Handlers

- Wrapper around a callback
- SPI
  - active?
  - commit -> callback-fn
  - lock-id -> unique-id
- java.util.concurrent.locks.Lock: lock, unlock
take/put handlers

• simple wrapper on callback
• lock is no-op
• lock-id is 0
• active? always true
• commit -> the callback
alt handlers

• each op handler wraps its own callback, but delegates rest to shared ‘flag’ handler

• flag handler has lock

  a boolean active? flag that starts true and makes one-time atomic transition

• commit transitions shared flag and returns callback

  must be called under lock
alt handlers
alt concurrency

- no global or multi-channel locking
- but channel does multi-handler locking
  - some ops commit both a put and take
- lock-ids used to ensure consistent lock acquisition order
alt cleanup

- ‘disabled’ handlers will still be in queues
- channel ops purge
SPI revisited

• handler callback only invoked on async completion

• only 2 scenarios

• when not ‘parked’, op happens immediately

  • callback is not used

  • non-nil return value is op return
Channel takes (fifo)
buffer(data)
data
closed?

impl/put!
[val handler]

completes handler
take!

Channel

takes (fifo)

buffer (data)

data

closed?

.impl/take! [handler]

completes handler
Wiring !/!!

- blocking ops (!!)
  create promise
  callback delivers
  only deref promise on nil return from op

- parking go ops (!)
  IOC state machine code is callback
Summary

• You don’t need to know any of this
• But understanding the ‘machine’ can help you make good decisions