core.async API Reference

API for clojure.core.async - Facilities for async programming and communication 0.1.0 (in development)

Full namespace name: clojure.core.async

Overview

Project home page is http://github.com/clojure/core.async/

Public Variables and Functions

<!
function

```
Usage: (<! port)
```

```
takes a val from port. Must be called inside a (go ...) block. Will
return nil if closed. Will park if nothing is available.
```

Source

<!!
function

```
Usage: (<!! port)
```

```
takes a val from port. Will return nil if closed. Will block
if nothing is available.
```

Source

>!
function

```
Usage: (>! port val)
```

```
puts a val into port. nil values are not allowed. Must be called
inside a (go ...) block. Will park if no buffer space is available.
Returns true unless port is already closed.
```

Source

>!!
function

```
Usage: (>!! port val)
```

```
puts a val into port. nil values are not allowed. Will block if no
buffer space is available. Returns true unless port is already closed.
```

Source

admix
function

```
Usage: (admix mix ch)
```

```
Adds ch as an input to the mix
```

Source

alt!
macro

```
Usage: (alt! & clauses)
```

```
Makes a single choice between one of several channel operations,
as if by alts!, returning the value of the result expr corresponding
to the operation completed. Must be called inside a (go ...) block.

Each clause takes the form of:

channel-op[s] result-expr

where channel-ops is one of:
```

```
take-port - a single port to take
[take-port | [put-port put-val] ...] - a vector of ports as per alts!
:default | :priority - an option for alts!

and result-expr is either a list beginning with a vector, whereupon that
vector will be treated as a binding for the [val port] return of the
operation, else any other expression.

(alt!
  [c t] ([val ch] (foo ch val))
  x ([v] v)
  [[out val]] :wrote
  :default 42)

Each option may appear at most once. The choice and parking
characteristics are those of alts!.
```

Source

alt!!
macro

```
Usage: (alt!! & clauses)
```

```
Like alt!, except as if by alts!!, will block until completed, and
not intended for use in (go ...) blocks.
```

Source

alts!
function

```
Usage: (alts! ports & {:as opts})
```

```
Completes at most one of several channel operations. Must be called
inside a (go ...) block. ports is a vector of channel endpoints,
which can be either a channel to take from or a vector of
[channel-to-put-to val-to-put], in any combination. Takes will be
made as if by <!, and puts will be made as if by >!. Unless
the :priority option is true, if more than one port operation is
ready a non-deterministic choice will be made. If no operation is
ready and a :default value is supplied, [default-val :default] will
be returned, otherwise alts! will park until the first operation to
become ready completes. Returns [val port] of the completed
operation, where val is the value taken for takes, and a
boolean (true unless already closed, as per put!) for puts.

opts are passed as :key val ... Supported options:

:default val - the value to use if none of the operations are immediately ready
:priority true - (default nil) when true, the operations will be tried in order.

Note: there is no guarantee that the port exps or val exprs will be
used, nor in what order should they be, so they should not be
depended upon for side effects.
```

Source

alts!!
function

```
Usage: (alts!! ports & {:as opts})
```

```
Like alts!, except takes will be made as if by <!!, and puts will
be made as if by >!!, will block until completed, and not intended
for use in (go ...) blocks.
```

Source

buffer
function

```
Usage: (buffer n)
```

```
Returns a fixed buffer of size n. When full, puts will block/park.
```

Source

chan
function

```
Usage: (chan)
```

```
       (chan buf-or-n)
```

```
Creates a channel with an optional buffer. If buf-or-n is a number,
will create and use a fixed buffer of that size.
```

Source

close!

function

```
Usage: (close! chan)
```

```
Closes a channel. The channel will no longer accept any puts (they
will be ignored). Data in the channel remains available for taking, until
exhausted, after which takes will return nil. If there are any
pending takes, they will be dispatched with nil. Closing a closed
channel is a no-op. Returns nil.
```

Source

do-alts

function

```
Usage: (do-alts fret ports opts)
```

```
returns derefable [val port] if immediate, nil if enqueued
```

Source

dropping-buffer

function

```
Usage: (dropping-buffer n)
```

```
Returns a buffer of size n. When full, puts will complete but
val will be dropped (no transfer).
```

Source

filter<

function

```
Usage: (filter< p ch)
       (filter< p ch buf-or-n)
```

```
Takes a predicate and a source channel, and returns a channel which
contains only the values taken from the source channel for which the
predicate returns true. The returned channel will be unbuffered by
default, or a buf-or-n can be supplied. The channel will close
when the source channel closes.
```

Source

filter>

function

```
Usage: (filter> p ch)
```

```
Takes a predicate and a target channel, and returns a channel which
supplies only the values for which the predicate returns true to the
target channel.
```

Source

go

macro

```
Usage: (go & body)
```

```
Asynchronously executes the body, returning immediately to the
calling thread. Additionally, any visible calls to <!, >! and alt!/alts!
channel operations within the body will block (if necessary) by
'parking' the calling thread rather than tying up an OS thread (or
the only JS thread when in ClojureScript). Upon completion of the
operation, the body will be resumed.

Returns a channel which will receive the result of the body when
completed
```

Source

go-loop

macro

```
Usage: (go-loop bindings & body)
```

```
Like (go (loop ...))
```

Source

into
function

```
Usage: (into coll ch)
```

```
Returns a channel containing the single (collection) result of the
items taken from the channel conjoined to the supplied
collection. ch must close before into produces a result.
```

Source

map
function

```
Usage: (map f chs)
       (map f chs buf-or-n)
```

```
Takes a function and a collection of source channels, and returns a
channel which contains the values produced by applying f to the set
of first items taken from each source channel, followed by applying
f to the set of second items from each channel, until any one of the
channels is closed, at which point the output channel will be
closed. The returned channel will be unbuffered by default, or a
buf-or-n can be supplied
```

Source

map<
function

```
Usage: (map< f ch)
```

```
Takes a function and a source channel, and returns a channel which
contains the values produced by applying f to each value taken from
the source channel
```

Source

map>
function

```
Usage: (map> f ch)
```

```
Takes a function and a target channel, and returns a channel which
applies f to each value before supplying it to the target channel.
```

Source

mapcat<
function

```
Usage: (mapcat< f in)
       (mapcat< f in buf-or-n)
```

```
Takes a function and a source channel, and returns a channel which
contains the values in each collection produced by applying f to
each value taken from the source channel. f must return a
collection.

The returned channel will be unbuffered by default, or a buf-or-n
can be supplied. The channel will close when the source channel
closes.
```

Source

mapcat>
function

```
Usage: (mapcat> f out)
       (mapcat> f out buf-or-n)
```

```
Takes a function and a target channel, and returns a channel which
applies f to each value put, then supplies each element of the result
to the target channel. f must return a collection.

The returned channel will be unbuffered by default, or a buf-or-n
can be supplied. The target channel will be closed when the source
```

```
channel closes.
```

Source

merge
function

```
Usage: (merge chs)
       (merge chs buf-or-n)
```

```
Takes a collection of source channels and returns a channel which
contains all values taken from them. The returned channel will be
unbuffered by default, or a buf-or-n can be supplied. The channel
will close after all the source channels have closed.
```

Source

mix
function

```
Usage: (mix out)
```

```
Creates and returns a mix of one or more input channels which will
be put on the supplied out channel. Input sources can be added to
the mix with 'admix', and removed with 'unmix'. A mix supports
soloing, muting and pausing multiple inputs atomically using
'toggle', and can solo using either muting or pausing as determined
by 'solo-mode'.

Each channel can have zero or more boolean modes set via 'toggle':

:solo - when true, only this (ond other soloed) channel(s) will appear
        in the mix output channel. :mute and :pause states of soloed
        channels are ignored. If solo-mode is :mute, non-soloed
        channels are muted, if :pause, non-soloed channels are
        paused.

:mute - muted channels will have their contents consumed but not included in the mix
:pause - paused channels will not have their contents consumed (and thus also not included in the mix)
```

Source

mult
function

```
Usage: (mult ch)
```

```
Creates and returns a mult(iple) of the supplied channel. Channels
containing copies of the channel can be created with 'tap', and
detached with 'untap'.

Each item is distributed to all taps in parallel and synchronously,
i.e. each tap must accept before the next item is distributed. Use
buffering/windowing to prevent slow taps from holding up the mult.

Items received when there are no taps get dropped.

If a tap puts to a closed channel, it will be removed from the mult.
```

Source

onto-chan
function

```
Usage: (onto-chan ch coll)
       (onto-chan ch coll close?)
```

```
Puts the contents of coll into the supplied channel.

By default the channel will be closed after the items are copied,
but can be determined by the close? parameter.

Returns a channel which will close after the items are copied.
```

Source

partition
function

```
Usage: (partition n ch)
       (partition n ch buf-or-n)
```

```
Returns a channel that will contain vectors of n items taken from ch. The
final vector in the return channel may be smaller than n if ch closed before
the vector could be completely filled.

The output channel is unbuffered by default, unless buf-or-n is given
```

Source

partition-by
function

```
Usage: (partition-by f ch)
       (partition-by f ch buf-or-n)
```

```
Returns a channel that will contain vectors of items taken from ch. New
 vectors will be created whenever (f itm) returns a value that differs from
 the previous item's (f itm).

The output channel is unbuffered, unless buf-or-n is given
```

Source

pipe
function

```
Usage: (pipe from to)
       (pipe from to close?)
```

```
Takes elements from the from channel and supplies them to the to
channel. By default, the to channel will be closed when the from
channel closes, but can be determined by the close?  parameter. Will
stop consuming the from channel if the to channel closes
```

Source

pub
function

```
Usage: (pub ch topic-fn)
       (pub ch topic-fn buf-fn)
```

```
Creates and returns a pub(lication) of the supplied channel,
partitioned into topics by the topic-fn. topic-fn will be applied to
each value on the channel and the result will determine the 'topic'
on which that value will be put. Channels can be subscribed to
receive copies of topics using 'sub', and unsubscribed using
'unsub'. Each topic will be handled by an internal mult on a
dedicated channel. By default these internal channels are
unbuffered, but a buf-fn can be supplied which, given a topic,
creates a buffer with desired properties.

Each item is distributed to all subs in parallel and synchronously,
i.e. each sub must accept before the next item is distributed. Use
buffering/windowing to prevent slow subs from holding up the pub.

Items received when there are no matching subs get dropped.

Note that if buf-fns are used then each topic is handled
asynchronously, i.e. if a channel is subscribed to more than one
topic it should not expect them to be interleaved identically with
the source.
```

Source

put!
function

```
Usage: (put! port val)
       (put! port val fn1)
       (put! port val fn1 on-caller?)
```

```
Asynchronously puts a val into port, calling fn1 (if supplied) when
complete, passing true iff port is already closed. nil values are
not allowed. If on-caller? (default true) is true, and the put is
immediately accepted, will call fn1 on calling thread.  Returns
true unless port is already closed.
```

Source

reduce
function

```
Usage: (reduce f init ch)
```

```
f should be a function of 2 arguments. Returns a channel containing
the single result of applying f to init and the first item from the
channel, then applying f to that result and the 2nd item, etc. If
the channel closes without yielding items, returns init and f is not
called. ch must close before reduce produces a result.
```

Source

remove<
function

```
Usage: (remove< p ch)
       (remove< p ch buf-or-n)
```

```
Takes a predicate and a source channel, and returns a channel which
contains only the values taken from the source channel for which the
predicate returns false. The returned channel will be unbuffered by
default, or a buf-or-n can be supplied. The channel will close
when the source channel closes.
```

Source

remove>
function

```
Usage: (remove> p ch)
```

```
Takes a predicate and a target channel, and returns a channel which
supplies only the values for which the predicate returns false to the
target channel.
```

Source

sliding-buffer
function

```
Usage: (sliding-buffer n)
```

```
Returns a buffer of size n. When full, puts will complete, and be
buffered, but oldest elements in buffer will be dropped (not
transferred).
```

Source

solo-mode
function

```
Usage: (solo-mode mix mode)
```

```
Sets the solo mode of the mix. mode must be one of :mute or :pause
```

Source

split
function

```
Usage: (split p ch)
       (split p ch t-buf-or-n f-buf-or-n)
```

```
Takes a predicate and a source channel and returns a vector of two
channels, the first of which will contain the values for which the
predicate returned true, the second those for which it returned
false.

The out channels will be unbuffered by default, or two buf-or-ns can
be supplied. The channels will close after the source channel has
closed.
```

Source

sub
function

```
Usage: (sub p topic ch)
       (sub p topic ch close?)
```

```
Subscribes a channel to a topic of a pub.

By default the channel will be closed when the source closes,
but can be determined by the close? parameter.
```

Source

take
function

```
Usage: (take n ch)
       (take n ch buf-or-n)
```

```
Returns a channel that will return, at most, n items from ch. After n items
 have been returned, or ch has been closed, the return chanel will close.

The output channel is unbuffered by default, unless buf-or-n is given.
```

Source

take!
function

```
Usage: (take! port fn1)
       (take! port fn1 on-caller?)
```

```
Asynchronously takes a val from port, passing to fn1. Will pass nil
if closed. If on-caller? (default true) is true, and value is
immediately available, will call fn1 on calling thread.
Returns nil.
```

Source

tap
function

```
Usage: (tap mult ch)
       (tap mult ch close?)
```

```
Copies the mult source onto the supplied channel.

By default the channel will be closed when the source closes,
but can be determined by the close? parameter.
```

Source

thread
macro

```
Usage: (thread & body)
```

```
Executes the body in another thread, returning immediately to the
calling thread. Returns a channel which will receive the result of
the body when completed.
```

Source

thread-call
function

```
Usage: (thread-call f)
```

```
Executes f in another thread, returning immediately to the calling
thread. Returns a channel which will receive the result of calling
f when completed.
```

Source

timeout
function

```
Usage: (timeout msecs)
```

```
Returns a channel that will close after msecs
```

Source

to-chan
function

```
Usage: (to-chan coll)
```

```
Creates and returns a channel which contains the contents of coll,
closing when exhausted.
```

Source

toggle

function

```
Usage: (toggle mix state-map)
```

```
Atomically sets the state(s) of one or more channels in a mix. The
state map is a map of channels -> channel-state-map. A
channel-state-map is a map of attrs -> boolean, where attr is one or
more of :mute, :pause or :solo. Any states supplied are merged with
the current state.

Note that channels can be added to a mix via toggle, which can be
used to add channels in a particular (e.g. paused) state.
```

Source

unblocking-buffer?

function

```
Usage: (unblocking-buffer? buff)
```

```
Returns true if a channel created with buff will never block. That is to say,
puts into this buffer will never cause the buffer to be full.
```

Source

unique

function

```
Usage: (unique ch)
       (unique ch buf-or-n)
```

```
Returns a channel that will contain values from ch. Consecutive duplicate
 values will be dropped.

The output channel is unbuffered by default, unless buf-or-n is given.
```

Source

unmix

function

```
Usage: (unmix mix ch)
```

```
Removes ch as an input to the mix
```

Source

unmix-all

function

```
Usage: (unmix-all mix)
```

```
removes all inputs from the mix
```

Source

unsub

function

```
Usage: (unsub p topic ch)
```

```
Unsubscribes a channel from a topic of a pub
```

Source

unsub-all

function

```
Usage: (unsub-all p)
       (unsub-all p topic)
```

```
Unsubscribes all channels from a pub, or a topic of a pub
```

Source

untap

function

```
Usage: (untap mult ch)
```

```
Disconnects a target channel from a mult
```

Source

untap-all
function

```
Usage: (untap-all mult)
```

```
Disconnects all target channels from a mult
```

Source

clojure.core.async.lab

```
core.async HIGHLY EXPERIMENTAL feature exploration

Caveats:

1. Everything defined in this namespace is experimental, and subject
to change or deletion without warning.

2. Many features provided by this namespace are highly coupled to
implementation details of core.async. Potential features which
operate at higher levels of abstraction are suitable for inclusion
in the examples.

3. Features provided by this namespace MAY be promoted to
clojure.core.async at a later point in time, but there is no
guarantee any of them will.
```

Types

BroadcastingWritePort
type

Fields: [write-ports]
Protocols: clojure.core.async.impl.protocols/WritePort
Interfaces:

MultiplexingReadPort
type

Fields: [mutex read-ports]
Protocols: clojure.core.async.impl.protocols/ReadPort
Interfaces:

Public Variables and Functions

->BroadcastingWritePort
function

```
Usage: (->BroadcastingWritePort write-ports)
```

```
Positional factory function for class clojure.core.async.lab.BroadcastingWritePort.
```

Source

->MultiplexingReadPort
function

```
Usage: (->MultiplexingReadPort mutex read-ports)
```

```
Positional factory function for class clojure.core.async.lab.MultiplexingReadPort.
```

Source

broadcast
function

```
Usage: (broadcast & ports)
```

```
Returns a broadcasting write port which, when written to, writes
the value to each of ports.

Writes to the broadcasting port will park until the value is written
to each of the ports used to create it. For this reason, it is
strongly advised that each of the underlying ports support buffered
writes.
```

Source

multiplex
function

```
Usage: (multiplex & ports)
```

```
Returns a multiplexing read port which, when read from, produces a
value from one of ports.

If at read time only one port is available to be read from, the
multiplexing port will return that value. If multiple ports are
available to be read from, the multiplexing port will return one
value from a port chosen non-deterministicly. If no port is
available to be read from, parks execution until a value is
available.
```

Source

spool
function

```
Usage: (spool s c)
       (spool s)
```

```
Take a sequence and puts each value on a channel and returns the channel.
If no channel is provided, an unbuffered channel is created. If the
sequence ends, the channel is closed.
```

Source