Package ‘async’

May 26, 2023

Title Coroutines: Generators / Yield, Async / Await, and Streams

Version 0.3.2

Date 2023-05-24

URL https://crowding.github.io/async/,
https://github.com/crowding/async/

BugReports https://github.com/crowding/async/issues

Description Write sequential-looking code that pauses and resumes.
gen() creates a generator, an iterator that returns a
value and pauses each time it reaches a yield() call.
async() creates a promise, which runs until it reaches
a call to await(), then resumes when information is available.
These work similarly to generator and async constructs
from 'Python' or 'JavaScript'. Objects produced are
compatible with the 'iterators' and 'promises' packages.
Version 0.3 supports on.exit, single-step debugging,
stream() for making asynchronous iterators, and
delimited goto() in switch() calls.

License GPL-2

Encoding UTF-8

Depends R (>= 4.1)

Imports iterors, nseval (>= 0.4.3), later, promises, testthat (>=
3.0.0), stringr, methods

Suggests rmarkdown, knitr, dplyr, curl, audio, profvis, ggplot2, XML

Collate 'async-package.R' 'util.R' 'cps.R' 'signals.R' 'syntax.R'
'coroutine.R' 'pump.R' 'run.R' 'gen.R' 'async.R' 'channel.R'
'stream.R' 'collect.R' 'all_names.R' 'walk.R' 'all_indices.R'
'graph.R' 'trans.R' 'munge.R' 'inline.R'

RoxygenNote 7.2.3

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel true

NeedsCompilation no

Author Peter Meilstrup [aut, cre]

Maintainer Peter Meilstrup <`peter.meilstrup@gmail.com`>

Repository CRAN

Date/Publication 2023-05-25 23:30:02 UTC

R topics documented:

`async` *Create an asynchronous task from sequential code.*

Description

`async({...})`, with an expression written in its argument, allows that expression to be evaluated in an asynchronous, or non-blocking manner. `async` returns an object with class `c("async", "promise")` which implements the promise interface.

Usage

```
async(
  expr,
  ...,
  split_pipes = TRUE,
  compileLevel = getOption("async.compileLevel"),
  debugR = FALSE,
  debugInternal = FALSE,
  trace = getOption("async.verbose")
)

await(prom, error)
```

Arguments

`expr`	An expression, to be executed asynchronously.
`...`	Undocumented.
`split_pipes`	Rewrite chained calls that use `await` (see below)
`compileLevel`	Compilation level; same options as for gen.
`debugR`	Set TRUE to enter the browser immediately on executing the first R expression.
`debugInternal`	Set TRUE to single-step at implementation level, immediately upon execution.
`trace`	Enable verbose logging by passing a function to `trace`, like `trace=cat`. This function should take a character argument.
`prom`	A promise, or something that can be converted to such by `promises::as.promise()`.
`error`	This argument will be forced if the promise rejects. If it is a function, it will be called with the error condition.

Details

An example Shiny app using `async/await` is on Github: `https://github.com/crowding/cranwhales-await`

When an `async` object is activated, it will evaluate its expression until it reaches the keyword `await`. The `async` object will return to its caller and preserve the partial state of its evaluation. When the awaited promise is resolved, evaluation continues from where the `async` left off.

When an async block finishes (either by reaching the end, or using `return()`), the promise resolves with the resulting value. If the async block stops with an error, the promise is rejected with that error.

Async blocks and generators are conceptually related and share much of the same underlying mechanism. You can think of one as "output" and the other as "input". A generator pauses until a value is requested, runs until it has a value to output, then pauses again. An async runs until it requires an external value, pauses until it receives the value, then continues.

The syntax rules for an `async` are analogous to those for `gen()`; `await` must appear only within the arguments of functions for which there is a pausable implementation (See `[pausables()]`). For `async` the default `split_pipes=TRUE` is enabled; this will rearrange some expressions to satisfy this requirement.

When `split_pipes=FALSE`, `await()` can only appear in the arguments of pausables and not ordinary R functions. This is an inconvenience as it prevents using `await()` in a pipeline. With `split_pipes=TRUE` applies some syntactic sugar: if an `await()` appears in the leftmost, unnamed, argument of an R function, the pipe will be "split" at that call using a temporary variable. For instance, either

```
async(makeRequest() |> await() |> sort())
```

or, equivalently,

```
async(sort(await(makeRequest())))
```

will be effectively rewritten to something like

```
async({.tmp <- await(makeRequest()); sort(.tmp)})
```

This works only so long as `await` appears in calls that evaluate their leftmost arguments normally. `split_pipes` can backfire if the outer call has other side effects; for instance `suppressWarnings(await(x))` will be rewritten as `{.tmp <- await(x); suppressWarnings(x)}`, which would defeat the purpose.

If `async` is given a function expression, like `async(function(...) ...)`, it will return an "async function" i.e. a function that constructs an async.

Value

`async()` returns an object with class "promise," as defined by the promises package (i.e., rather than the kind of promise used in R's lazy evaluation.)

In the context of an `async` or `stream`, `await(x)` returns the resolved value of a promise `x`, or stops with an error.

Examples

```
myAsync <- async(for (i in 1:4) {
  await(delay(5))
  cat(i, "\n")
})
```

`awaitNext` *Wait for the next value from a channel or stream.*

Description

`awaitNext` can be used within an async or stream coroutine. When reached, `awaitNext` will register to receive the next element from an async or a coroutine object.

Usage

```
awaitNext(strm, or, err)
```

Arguments

`strm`	A channel or stream object.
`or`	This argument will be evaluated and returned in the case the channel closes. If not specified, awaiting on a closed stream will raise an error with message "StopIteration".
`err`	A function to be called if the channel throws an error condition.

Value

In the context of an `async` or `stream`, `awaitNext(x)` returns the resolved value of a promise `x`, or stops with an error.

`channel` *An object representing a sequence of future values.*

Description

A `channel` is an object that represents a sequence of values yet to be determined. It is something like a combination of a promise and an iteror.

Usage

```
channel(obj, ...)

## S3 method for class '`function`'
channel(
  obj,
  ...,
  max_queue = 500L,
  max_awaiting = 500L,
  wakeup = function(...) NULL
)

is.channel(x)
```

Arguments

`obj`	A user-provided function; it will receive three callback functions as arguments, in order, `emit(val)`, `reject(err)` and `close()`
`...`	Specialized channel methods may take other arguments.
`max_queue`	The maximum number of outgoing values to store if there are no listeners. Beyond this, calling `emit` will return an error.
`max_awaiting`	The maximum number of pending requests. If there are this many outstanding requests, for values, calling `nextThen(ch, ...)` or `nextElem(ch)` will raise an error.
`wakeup`	You may optionally provide a callback function here. It will be called when the queue is empty and there is at least one listener/outstanding promise.
`x`	an object.

Details

The channel interface is intended to represent and work with asynchronous, live data sources, for instance event logs, non-blocking connections, paginated query results, reactive values, and other processes that yield a sequence of values over time.

`channel` is an S3 method and will attempt to convert the argument `obj` into a channel object according to its class.

The friendly way to obtain values from a channel is to use `awaitNext` or `for` loops within an async or stream coroutine.

The low-level interface to obtain values from a channel is to call nextThen`(ch, onNext=, onError=, onClose=, ...)]`, providing callback functions for at least `onNext(val)`. Those callbacks will be appended to an internal queue, and will be called as soon as data is available, in the order that requests were received.

You can also treat a channel as an iteror over promises, calling `nextOr(pri)` to return a promise representing the next available value. Each promise created this way will be resolved in the order that data come in. Note that this way there is no special signal for end of iteration; a promise will reject with a condition message `"StopIteration"` to signal end of iteration.

Be careful with the iterator-over-promises interface though: if you call `as.list.iteror(pr)` you may get stuck in an infinite loop, as `as.list` keeps calling `nextElem` and receives more promises to represent values that exist only hypothetically. This is one reason for the `max_listeners` limit.

The friendly way to create a channel with custom behavior is to use a stream coroutine. Inside of `stream()` call await to wait on promises, awaitNext to wait on other streams and yield to yield values. To signal end of iteration use `return()` (which will discard its value) and to signal an error use `stop()`.

The low-level interface to create a channel with custom behavior is to call `channel(function(emit, reject, cancel) {...})`, providing your own function definition; your function will receive those three callback methods as arguments. Then use whatever means to arrange to call `emit(val)` some time in the future as data comes in. When you are done emitting values, call the `close()` callback. To report an error call `reject(err)`; the next requestor will receive the error. If there is more than one listener, other queued listeners will get a `close` signal.

Value

a channel object, supporting methods "nextThen" and "nextOr"

`is.channel(x)` returns TRUE if its argument is a channel object.

Author(s)

Peter Meilstrup

`combine` *Combine several channels into one.*

Description

`combine(...)` takes any number of promise or channel objects. It awaits each one, and returns a channel object which re-emits every value from its targets, in whatever order they are received.

Usage

```
combine(...)
```

Arguments

`...`	Each argument should be a promise or a channel.

Value

a channel object.

Author(s)

Peter Meilstrup

`debugAsync` *Toggle single-step debugging for a coroutine.*

Description

Toggle single-step debugging for a coroutine.

Usage

```
debugAsync(x, R, internal, trace)
```

Arguments

`x`	A coroutine object as constructed by (async, gen or stream).
`R`	Set TRUE to step through expressions at user level
`internal`	Set TRUE to step through at coroutine implementation level.
`trace`	Set TRUE or provide a print function to print each R expression evaluated in turn.

Value

a `list(R=, internal=, trace=)` with the current debug state.

`delay` *Asynchronous pause.*

Description

"delay" returns a promise which resolves only after the specified number of seconds. This uses the R event loop via later. In an `[async]` construct you can use `await(delay(secs))` to yield control, for example if you need to poll in a loop.

Usage

```
delay(secs, expr = NULL)
```

Arguments

`secs`	The promise will resolve after at least this many seconds.
`expr`	The value to resolve with; will be forced after the delay.

Value

An object with class "promise".

Examples

```
# print a message after a few seconds
async({await(delay(10)); cat("Time's up!\n")})
```

`format.coroutine` *Query / display coroutine properties and state.*

Description

The coroutine `format` method displays its source code, its effective environment, whether it is running or finished, and a label indicating its last known state. The `summary` method returns the same information in a list.

`summary(obj)` returns a list with information on a coroutine's state, including:

- `code`: the expression used to create the coroutine;
- `state`: the current state (see below);
- `node`: is a character string that identifies a location in the coroutine source code; for example, a typical state string might be ".{.<-2.await__then", which can be read like "in the first argument of `\{`, in the second argument of `<-`, in a call to `await()`, at internal node `then`.";
- `envir`: the environment where the coroutine is evaluating R expressions;
- `err`: the error object, if the coroutine caught an error.

`summary(g)$state` for a generator g might be "yielded", "running" (if nextElem is *currently* being called,) "stopped" (for generators that have stopped with an error,) or "finished" (for generators that have finished normally.)

`summary(a)$state` of an async might be "pending", "resolved" or "rejected".

`summary(s)$state` on a stream might be "resolved", "rejected", "running", "woken", "yielding", or "yielded".

Usage

```
## S3 method for class 'coroutine'
format(x, ...)

## S3 method for class 'coroutine'
summary(object, ...)

## S3 method for class 'generator'
summary(object, ...)

## S3 method for class 'async'
summary(object, ...)

## S3 method for class 'stream'
summary(object, ...)
```

Arguments

`x`	A coroutine.
`...`	Undocumented.
`object`	a coroutine (async, generator, or stream) object.

`gather` *Collect iterator / channel items into a vector.*

Description

`gather` takes a channel as argument and returns a promise. All values emitted by the channel will be collected into a vector matching the prototype mode. After the source channel closes, the promise will resolve with the collected vector.

Method `as.promise.channel` is a synonym for `gather`.

`collect` and `collector` are used in the implementation of the above functions. `collect` calls the function `fn` in its argument, supplying a callback of the form `function (val, name=NULL)`. I like to call it `emit`. While `fn` is running, it can call `emit(x)` any number of times. After `fn` returns, all the values passed to `emit` are returned in a vector, with optional names.

`collector()` works similarly to collect() but does not gather values when your inner function returns. Instead, it provides your inner function with two callbacks, one to add a value and the second to extract the value; so you can use that callback to extract values at a later time. For an example of `collector` usage see the definition of gather.

Usage

```
gather(ch, type = list())

## S3 method for class 'channel'
as.promise(x)

collect(fn, type = list())

collector(fn, type = list())
```

Arguments

`ch`	a channel object.
`type`	A prototype output vector (similar to the `FUN.VALUE` argument of vapply) Defaults to `list()`.
`x`	a channel.
`fn`	A function, which should accept a single argument, here called `emit`.

Value

`gather(ch, list())` returns a [promise] that eventually resolves with a list. If the channel emits an error, the promise will reject with that error. The partial results will be attached to the error's `attr(err, "partialResults")`.

`collect` returns a vector of the same mode as `type`.

Author(s)

Peter Meilstrup

Examples

```
ch <- stream(for (i in 1:10) {await(delay(0.1)); if (i %% 3 == 0) yield(i)})
## Not run:  ch |> gather(numeric(0)) |> then(\(x)cat(x, "\n"))

#cumulative sum with collect
cumsum <- function(vec) {
  total <- 0
  collect(type=0, function(emit) {
    for (i in vec) total <- emit(total+i)
  })
}

# `as.list.iteror` is implemented simply with `collect`:
as.list.iteror <- function(it) {
  collect(\(yield) repeat yield(nextOr(it, break)))
}
```

gen *Create an iterator using sequential code.*

Description

`gen({...})` with an expression written in its argument, creates a generator, an object which computes an indefinite sequence.

When written inside a generator expression, `yield(expr)` causes the generator to return the given value, then pause until the next value is requested.

When running in a generator expression, `yieldFrom(it))`, given a list or iteror in its argument, will yield successive values from that iteror until it is exhausted, then continue.

Usage

```
gen(
  expr,
  ...,
  split_pipes = FALSE,
  compileLevel = getOption("async.compileLevel")
)

yield(expr)

yieldFrom(it, err)
```

Arguments

`expr`	An expression, to be turned into an iterator.
`...`	Undocumented.
`split_pipes`	Silently rewrite expressions where "yield" appears in chained calls. See async.
`compileLevel`	Current levels are 0 (no compilation) or -1 (name munging only).
`it`	A list, iteror or compatible object.
`err`	An error handler

Details

On the "inside", that is the point of view of code you write in `{...}`, is ordinary sequential code using conditionals, branches, loops and such, outputting one value after another with `yield()`. For example, this code creates a generator that computes a random walk:

```
rwalk <- gen({
  x <- 0;
  repeat {
    x <- x + rnorm(1)
    yield(x)
  }
})
```

On the "outside," that is, the object returned by `gen()`, a generator behaves like an iterator over an indefinite collection. So we can collect the first 100 values from the above generator and compute their mean:

```
rwalk |> itertools2::take(100) |> as.numeric() |> mean()
```

When `nextOr(rwalk, ...)` is called, the generator executes its "inside" expression, in a local environment, until it reaches a call to `yield()`. THe generator 'pauses', preserving its execution state, and `nextElem` then returns what was passed to `yield`. The next time `nextElem(rwalk)` is called, the generator resumes executing its inside expression starting after the `yield()`.

If you call `gen` with a function expression, as in:

```
gseq <- gen(function(x) for (i in 1:x) yield(i))
```

then instead of returning a single generator it will return a *generator function* (i.e. a function that constructs and returns a generator.) The above is morally equivalent to:

```
gseq <- function(x) {force(x); gen(for (i in 1:x) yield(i))}
```

so the generator function syntax just saves you writing the force call.

A generator expression can use any R functions, but a call to `yield` may only appear in the arguments of a "pausable" function. The `async` package has several built-in pausable functions corresponding to base R's control flow functions, such as `if`, `while`, `tryCatch`, `<-`, `{}`, `||` and so on (see pausables for more details.) A call to `yield` may only appear in an argument of one of these pausable functions. So this random walk generator:

```
rwalk <- gen({x <- 0; repeat {x <- yield(x + rnorm(1))}})
```

is legal, because `yield` appears within arguments to `{}`, `repeat`, and `<-`, for which this package has pausable definitions. However, this:

```
rwalk <- gen({x <- rnorm(1); repeat {x <- rnorm(1) + yield(x)}})
```

is not legal, because `yield` appears in an argument to `+`, which does not have a pausable definition.

Value

‘gen(...) returns an iteror.

`yield(x)` returns the same value x.

yieldFrom returns NULL, invisibly.

Examples

```
i_chain <- function(...) {
  iterators <- list(...)
  gen(for (it in iterators) yieldFrom(it))
}
```

`goto` *Coroutine switch with delimited goto.*

Description

The `switch` function implemented for coroutines in the `async` package is more strict than the one in base R. In a coroutine, `switch` will always either take one of the given branches or throw an error, whereas base R `switch` will silently return NULL if no branch matches switch argument. Otherwise, the same conventions apply as `base::switch()` (e.g. empty switch branches fall through; a character switch may have one unnamed argument as a default.)

Usage

```
goto(branch = NULL)
```

Arguments

`branch`	A character string naming the new branch. If missing or NULL, jumps back to re-evaluate the switch argument.

Details

Coroutine `switch` also supports a delimited form of `goto`. Within a branch, `goto("other_branch")` will stop executing the present branch and jump to the named branch. Calling `goto()` without arguments will jump back to re-evaluate the switch expression.

If a `goto` appears in a try-finally call, as in:

```
switch("branch",
   branch=tryCatch({...; goto("otherBranch")},
                   finally={cleanup()}),
   otherBranch={...}
)
```

the `finally` clause will be executed *before* switching to the new branch.

`graphAsync` *Draw a graph representation of a coroutine.*

Description

`graphAsync` will traverse the objects representing a generator or async and render a graph of its structure using Graphviz (if it is installed.)

Usage

```
graphAsync(
  obj,
  basename = if (is.name(substitute(obj))) as.character(substitute(obj)) else
    stop("Please specify basename"),
  type = "pdf",
  ...,
  envs = TRUE,
  vars = FALSE,
  handlers = FALSE,
  orphans = FALSE,
  dot = find_dot(),
  filename = paste0(basename, ".", type),
  dotfile = if (type == "dot") filename else paste0(basename, ".dot")
)
```

Arguments

`obj`	A generator, async or stream object.
`basename`	The base file name. If `basename="X"` and `type="pdf"` you will end up with two files, `"X.dot"` and `"X.pdf"`.
`type`	the output format. If "dot", we will just write a Graphviz dot file. If another extension like "pdf" or "svg", will write a DOT file and then attempt to invoke Graphviz `dot` (if it is available according to `Sys.which`) to produce the image. If `type=""` `graphAsync` will return graphviz DOT language as a character vector
`...`	Unused.
`envs`	If `TRUE`, multiple nodes that share the same environment will be grouped together in clusters.
`vars`	If `TRUE`, context variables used in each state node will be included on the graph, with edges indicating reads/stores.
`handlers`	If `TRUE`, state nodes will have thin edges connecting to trampoline handlers they call, in addition to the dashed edges connecting to the next transition.
`orphans`	If `TRUE`, nodes will be included even if there are no connections to them (this mostly being interface methods and unused handlers).
`dot`	Optional path to the `dot` executable.
`filename`	Optionally specify the output picture file name.
`dotfile`	Optionally specify the output DOT file name.

Details

`graphAsync` will write a Graphviz DOT format file describing the given generator or async/await block. The graph shows the generator as a state machine with nodes that connect to each other.

If `type` is something other than `dot` `graphAsync will then try to invoke Graphviz dot`‘ to turn the graph description into an image file.

The green octagonal node is where the program starts, while red "stop" and blue "return" are where it ends. Nodes in green type on dark background show code that runs in the host language unmodified; gray nodes implement control flow. Dark arrows carry a value; gray edges carry no value. A "semicolon" node receives a value and discards it.

Some nodes share a context with other nodes, shown by an enclosing box. Contexts can have state variables, shown as a rectangular record; orange edges from functions to variables represent writes; blue edges represent reads.

Dashed edges represent a state transition that goes through a trampoline handler. Dashed edges have a Unicode symbol representing the type of trampoline; (DOUBLE VERTICAL BAR) for await/yield; (TOP ARC ANTICLOCKWISE ARROW WITH PLUS) or (TOP ARC CLOCKWISE ARROW WITH MINUS) to wind on or off an exception handler; (ANTICLOCKWISE TRIANGLE-HEADED BOTTOM U-SHAPED ARROW) for a plain trampoline with no side effects (done once per loop, to avoid overflowing the stack.) Meanwhile, a thin edge connects to the trampoline handler. (So the user-facing "yield" function registers a continuation to the next step but actually calls the generator's yield handler.)

Value

If `type=""`, a character vector of DOT source. Else The name of the file that was created.

Examples

```
randomWalk <- gen({x <- 0; repeat {yield(x); x <- x + rnorm(1)}})
## Not run:
graphAsync(randomWalk, "pdf")
# writes "randomWalk.dot" and invokes dot to make "randomWalk.pdf"

#or, display it in an R window with the Rgraphviz package:
g <- Rgraphviz::agread("randomWalk.dot")
Rgraphviz::plot(g)

## End(Not run)
#Or render an HTML sidget using DiagrammeR:
## Not run:
dot <- graphAsync(randomWalk, type="")
DiagrammeR::DiagrammeR(paste0(dot, collapse="\n"), type="grViz")

## End(Not run)
```

`nextThen` *Receive values from channels by callback.*

Description

`nextThen` is the callback-oriented interface to work with channel objects. Provide the channel callback functions to receive the next element, error, and closing signals; your callbacks will be stored in a queue and called when values are available.

Usage

```
nextThen(x, onNext, onError, onClose, ...)

subscribe(x, ...)
```

Arguments

`x`	A channel object
`onNext`	For nextThen, a function to be called with the next emitted value. For subscribe, a function to be called with each emitted value until the stream finishes.
`onError`	Function to be called if channel stops with an error. Note that if you call nextThen multiple times to register multile callbacks, only the first will receive onError; the rest will be called with onClose.
`onClose`	Function to be called if the channel finishes normally.
`...`	Undocumented.

Details

`subscribe` is similar to nextThen except that your `onNext` will be called for each value the channel emits. It is just implemented in terms of nextThen, with a callback that re-registers itself.

---

pausables *Pausable functions.*

Description

Coroutines rely on "pausable" workalikes for control flow functions like `if`, `while`, and so on. `pausables()` scans for and returns a list of all pausable functions visible from the present environment.

Usage

```
pausables(envir = caller(), packages = NULL)
```

Arguments

`envir`	The environment to search (defaulting to the calling environment).
`packages`	By default, will only look for pausable functions visible from the caller's environment. `packages` argument additionally specifies aditional packages to search. `packages=base::.packages()` will search all currently loaded packages. `[.packages(all.available=TRUE)]` will search all installped package.

Details

A pausable function is a public function that has a corresponding private function with a name endng with `_cps`. Most of these private functions are defined in `async` source file `cps.r`. For instance, `async:::for_cps` contains the pausable implementation of `for`.

Value

A list of expressions (either names or `:::` calls)

`run` *Execute a generator expression immediately, collecting yielded values.*

Description

`run(expr)` with an expression directly writen, will parse that expression as a coroutine, but then run it without pausing.

Usage

```
run(
  expr,
  type = list(),
  ...,
  split_pipes = FALSE,
  debugR = FALSE,
  debugInternal = FALSE,
  trace = getOption("async.verbose")
)
```

Arguments

`expr`	A generator expression, same as you would write in gen.
`type`	A value whose mode will determine the output vector mode (as in vapply.)
`...`	Undocumented.
`split_pipes`	See async; defaults to FALSE.
`debugR`	Will open a browser at the first and subsequent R evaluations allowing single-stepping through user code.
`debugInternal`	Will set a breakpoint at the implementation level, allowing single-stepping through `async` package code.
`trace`	a tracing function.

Details

If the expression contains any calls to `yield()`, `run()` will collect all the values passed to yield() and return a list. If the expression contains a `yield()` but it is never called, `run()` returns an empty list. If the expression does not contain a `yield` at all, `run` returns the expression's final return value.

`run(expr)` is similar to `as.list(gen(expr))`, except `run(expr)` evaluates its expression directly in the calling environment, while `gen` creates a new enclosed environment to run in.

`run` is useful if you want to take advantage of coroutine language extensions, such as using `for` loops over iterators, or using `goto()` in `switch` statements, in otherwise synchronous code. If you want to collect a variable-length sequence of values but don't need those features, using collect directly will have better performance.

Value

If `expr` contains any `yield` calls, a vector of the same mode as `type`; otherwise the return value of `expr`.

Examples

```
run(type=0, {
  for (i in iterors::iseq(2, Inf, by=5)) {
    if (i %% 37 == 0) break
    else yield(i)
  }
})
```

`stream` *Create an asynchronous iterator by writing sequential code.*

Description

(Experimental as of async 0.3) `stream(...)` constructs a channel object, i.e. an asynchronous iterator, which will compute and return values according to sequential code written in `expr`. A `stream` is a coroutine wearing a channel interface in the same way that `async` is a coroutine wearing a promise interface, and a gen is a coroutine sitting behind an iteror interface.

Usage

```
stream(
  expr,
  ...,
  split_pipes = TRUE,
  lazy = TRUE,
  compileLevel = getOption("async.compileLevel"),
  debugR = FALSE,
  debugInternal = FALSE,
  trace = getOption("async.verbose")
)
```

Arguments

`expr`	A coroutine expression, using some combination of `yield`, `await`, `awaitNext`, `yieldFrom`, standard control flow operators and other calls.
`...`	Undocumented.
`split_pipes`	See description under async; defaults to `TRUE`.
`lazy`	If TRUE, start paused, and pause after `yield()` (see above.)
`compileLevel`	Compilation level.

`debugR`	Set TRUE to single-step debug at R level. Use `debugAsync()` to enable or disable debugging on a stream after it has been created.
`debugInternal`	Set TRUE to single-step debug at coroutine implementation level.
`trace`	An optional tracing function.

Details

In a stream expression, you can call `yield()` to emit a value, and `await()` to wait for a value from a promise. To have your stream wait for values from another stream or channel, call `awaitNext()`; you can also use `awaitNext` when you are writing an `async`. You can also use a simple `for` loop to consume all future values from a stream or channel.

The lower-level interface to consume values from a stream is by using nextThen from the channel interface.

Streams come in both "lazy" and "eager" varieties. If `lazy=TRUE`, a stream starts idle, and does not process anything until it is woken up by a call to its channel's `nextThen`. It will pause after reaching `yield` if there are no more outstanding requests. If `lazy=FALSE`, a stream will begin executing immediately, not pausing on `yield`, possibly queuing up emitted values until it needs to `await` something.

(For comparison, in this package, gen are lazy in that they do not start executing until a call to `nextOr` and pause immediately after `yield`, while async blocks are eager, starting at construction and running until they hit an `await`.)

Like its coroutine counterparts, if `stream` is given a function expression, like `stream(function(...) ...)`, it will return a "stream function" i.e. a function that constructs a stream object.

Value

An object with (at least) classes "stream", "channel", "coroutine", "iteror", "iter".

Author(s)

Peter Meilstrup

Examples

```
# emit values _no more than_ once per second
count_to <- stream(function(n, interval=1) {
  for (i in 1:n) {
    await(delay(interval))
    yield(i)
  }
})

accumulate <- stream(function(st, sum=0) {
  for (i in st) {sum <- sum + i; yield(sum)}
})

print_each <- async(function(st) for (i in st) print(i))
```

```
count_to(10) |> accumulate() |> print_each()
```

Index