Package 'coro'

November 5, 2024

Title 'Coroutines' for R

Version 1.1.0

Description Provides 'coroutines' for R, a family of functions that can be suspended and resumed later on. This includes 'async' functions (which await) and generators (which yield). 'Async' functions are based on the concurrency framework of the 'promises' package. Generators are based on a dependency free iteration protocol defined in 'coro' and are compatible with iterators from the 'reticulate' package.

License MIT + file LICENSE

URL https://github.com/r-lib/coro, https://coro.r-lib.org/

BugReports https://github.com/r-lib/coro/issues

Depends R (>= 3.5.0)

Imports rlang (>= 0.4.12)

Suggests knitr, later (>= 1.1.0), magrittr (>= 2.0.0), promises, reticulate, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Lionel Henry [aut, cre], Posit Software, PBC [cph, fnd]

Maintainer Lionel Henry <lionel@posit.co>

Repository CRAN

Date/Publication 2024-11-05 10:30:09 UTC

Contents

`async` *Make an async function*

Description

`async()` functions are building blocks for cooperative concurrency.

- They are *concurrent* because they are jointly managed by a scheduler in charge of running them.
- They are *cooperative* because they decide on their own when they can no longer make quick progress and need to **await** some result. This is done with the `await()` keyword which suspends the async function and gives control back to the scheduler. The scheduler waits until the next async operation is ready to make progress.

The async framework used by `async()` functions is implemented in the later and promises packages:

- You can chain async functions created with coro to promises.
- You can await promises. You can also await futures created with the future package because they are coercible to promises.

Usage

```
async(fn)

await(x)
```

Arguments

`fn`	An anonymous function within which `await()` calls are allowed.
`x`	An awaitable value, i.e. a promise.

Value

A function that returns a promises::promise() invisibly.

See Also

async_generator() and await_each(); coro_debug() for step-debugging.

Examples

```
# This async function counts down from `n`, sleeping for 2 seconds
# at each iteration:
async_count_down <- async(function(n) {
  while (n > 0) {
    cat("Down", n, "\n")
    await(async_sleep(2))
    n <- n - 1
  }
})

# This async function counts up until `stop`, sleeping for 0.5
# seconds at each iteration:
async_count_up <- async(function(stop) {
  n <- 1
  while (n <= stop) {
    cat("Up", n, "\n")
    await(async_sleep(0.5))
    n <- n + 1
  }
})

# You can run these functions concurrently using `promise_all()`
if (interactive()) {
  promises::promise_all(async_count_down(5), async_count_up(5))
}
```

---

async_collect *Collect elements of an asynchronous iterator*

Description

async_collect() takes an asynchronous iterator, i.e. an iterable function that is also awaitable. async_collect() returns an awaitable that eventually resolves to a list containing the values returned by the iterator. The values are collected until exhaustion unless n is supplied. The collection is grown geometrically for performance.

Usage

```
async_collect(x, n = NULL)
```

Arguments

x	An iterator function.
n	The number of elements to collect. If x is an infinite sequence, n must be supplied to prevent an infinite loop.

Examples

```
# Emulate an async stream by yielding promises that resolve to the
# elements of the input vector
generate_stream <- async_generator(function(x) for (elt in x) yield(elt))

# You can await `async_collect()` in an async function. Once the
# list of values is resolved, the async function resumes.
async(function() {
  stream <- generate_stream(1:3)
  values <- await(async_collect(stream))
  values
})
```

---

async_generator *Construct an async generator*

Description

An async generator constructs iterable functions that are also awaitables. They support both the `yield()` and `await()` syntax. An async iterator can be looped within async functions and iterators using `await_each()` on the input of a `for` loop.

The iteration protocol is derived from the one described in `iterator`. An async iterator always returns a promise. When the iterator is exhausted, it returns a resolved promise to the exhaustion sentinel.

Usage

```
async_generator(fn)

await_each(x)
```

Arguments

fn	An anonymous function describing an async generator within which `await()` calls are allowed.
x	An awaitable value, i.e. a promise.

Value

A generator factory. Generators constructed with this factory always return `promises::promise()`.

See Also

async() for creating awaitable functions; async_collect() for collecting the values of an async iterator; coro_debug() for step-debugging.

Examples

```
# Creates awaitable functions that transform their inputs into a stream
generate_stream <- async_generator(function(x) for (elt in x) yield(elt))

# Maps a function to a stream
async_map <- async_generator(function(.i, .fn, ...) {
  for (elt in await_each(.i)) {
    yield(.fn(elt, ...))
  }
})

# Example usage:
if (interactive()) {
  library(magrittr)
  generate_stream(1:3) %>% async_map(`*`, 2) %>% async_collect()
}
```

---

async_sleep *Sleep asynchronously*

Description

Sleep asynchronously

Usage

```
async_sleep(seconds)
```

Arguments

seconds	The number of second to sleep.

Value

A chainable promise.

`as_iterator` *Transform an object to an iterator*

Description

`as_iterator()` is a generic function that transforms its input to an iterator function. The default implementation is as follows:

- Functions are returned as is.
- Other objects are assumed to be vectors with `length()` and `[[` methods.

Methods must return functions that implement coro's iterator protocol.

`as_iterator()` is called by coro on the RHS of `in` in `for` loops. This applies within generators, async functions, and `loop()`.

Usage

```
as_iterator(x)

## Default S3 method:
as_iterator(x)
```

Arguments

`x`	An object.

Value

An iterable function.

Examples

```
as_iterator(1:3)

i <- as_iterator(1:3)
loop(for (x in i) print(x))
```

collect *Iterate over iterator functions*

Description

`loop()` and `collect()` are helpers for iterating over iterator functions such as generators.

- `loop()` takes a `for` loop expression in which the collection can be an iterator function.
- `collect()` loops over the iterator and collects the values in a list.

Usage

```
collect(x, n = NULL)

loop(loop)
```

Arguments

`x`	An iterator function.
`n`	The number of elements to collect. If `x` is an infinite sequence, `n` must be supplied to prevent an infinite loop.
`loop`	A `for` loop expression.

Value

`collect()` returns a list of values; `loop()` returns the `exhausted()` sentinel, invisibly.

See Also

`async_collect()` for async generators.

Examples

```
generate_abc <- generator(function() for (x in letters[1:3]) yield(x))
abc <- generate_abc()

# Collect 1 element:
collect(abc, n = 1)

# Collect all remaining elements:
collect(abc)

# With exhausted iterators collect() returns an empty list:
collect(abc)


# With loop() you can use `for` loops with iterators:
abc <- generate_abc()
loop(for (x in abc) print(x))
```

coro_debug *Debug a generator or async function*

Description

- Call `coro_debug()` on a `generator()`, `async()`, or `async_generator()` function to enable step-debugging.
- Alternatively, set `options(coro_debug = TRUE)` for step-debugging through all functions created with coro.

Usage

```
coro_debug(fn, value = TRUE)
```

Arguments

`fn`	A generator factory or an async function.
`value`	Whether to debug the function.

generator *Create a generator function*

Description

`generator()` creates an generator factory. A generator is an iterator function that can pause its execution with `yield()` and resume from where it left off. Because they manage state for you, generators are the easiest way to create iterators. See `vignette("generator")`.

The following rules apply:

- Yielded values do not terminate the generator. If you call the generator again, the execution resumes right after the yielding point. All local variables are preserved.
- Returned values terminate the generator. If called again after a `return()`, the generator keeps returning the `exhausted()` sentinel.

Generators are compatible with all features based on the iterator protocol such as `loop()` and `collect()`.

Usage

```
generator(fn)

gen(expr)
```

Arguments

fn	A function template for generators. The function can yield() values. Within a generator, for loops have iterator support.
expr	A yielding expression.

See Also

yield(), coro_debug() for step-debugging.

Examples

```
# A generator statement creates a generator factory. The
# following generator yields three times and then returns `"d"`.
# Only the yielded values are visible to the callers.
generate_abc <- generator(function() {
  yield("a")
  yield("b")
  yield("c")
  "d"
})

# Equivalently:
generate_abc <- generator(function() {
  for (x in c("a", "b", "c")) {
    yield(x)
  }
})

# The factory creates generator instances. They are iterators
# that you can call successively to obtain new values:
abc <- generate_abc()
abc()
abc()

# Once a generator has returned it keeps returning `exhausted()`.
# This signals to its caller that new values can no longer be
# produced. The generator is exhausted:
abc()
abc()

# You can only exhaust a generator once but you can always create
# new ones from a factory:
abc <- generate_abc()
abc()


# As generators implement the coro iteration protocol, you can use
# coro tools like `loop()`. It makes it possible to loop over
# iterators with `for` expressions:
loop(for (x in abc) print(x))
```

```
# To gather values of an iterator in a list, use `collect()`. Pass
# the `n` argument to collect that number of elements from a
# generator:
abc <- generate_abc()
collect(abc, 1)

# Or drain all remaining elements:
collect(abc)


# coro provides a short syntax `gen()` for creating one-off
# generator _instances_. It is handy to adapt existing iterators:
numbers <- 1:10
odds <- gen(for (x in numbers) if (x %% 2 != 0) yield(x))
squares <- gen(for (x in odds) yield(x^2))
greetings <- gen(for (x in squares) yield(paste("Hey", x)))

collect(greetings)


# Arguments passed to generator instances are returned from the
# `yield()` statement on reentry:
new_tally <- generator(function() {
  count <- 0
  while (TRUE) {
    i <- yield(count)
    count <- count + i
  }
})
tally <- new_tally()
tally(1)
tally(2)
tally(10)
```

`iterator` *Iterator protocol*

Description

An **iterator** is a function that implements the following protocol:

- Calling the function advances the iterator. The new value is returned.
- When the iterator is exhausted and there are no more elements to return, the symbol `quote(exhausted)` is returned. This signals exhaustion to the caller.
- Once an iterator has signalled exhaustion, all subsequent invokations must consistently return `coro::exhausted()` or `as.symbol(".__exhausted__.")`.
- The iterator function may have a `close` argument taking boolean values. When passed a `TRUE` value, it indicates early termination and the iterator is given the opportunity to clean up resources.

Cleanup must only be performed once, even if the iterator is called multiple times with `close = TRUE`.

An iterator is allowed to not have any `close` argument. Iterator drivers must check for the presence of the argument. If not present, the iterator can be dropped without cleanup.

An iterator passed `close = TRUE` must return `coro::exhausted()` and once closed, an iterator must return `coro::exhausted()` when called again.

```
iterator <- as_iterator(1:3)

# Calling the iterator advances it
iterator()
#> [1] 1
iterator()
#> [1] 2

# This is the last value
iterator()
#> [1] 3

# Subsequent invokations return the exhaustion sentinel
iterator()
#> .__exhausted__.
```

Because iteration is defined by a protocol, creating iterators is free of dependency. However, it is often simpler to create iterators with generators, see `vignette("generator")`. To loop over an iterator, it is simpler to use the `loop()` and `collect()` helpers provided in this package.

Usage

```
exhausted()

is_exhausted(x)
```

Arguments

x	An object.

Properties

Iterators are **stateful**. Advancing the iterator creates a persistent effect in the R session. Also iterators are **one-way**. Once you have advanced an iterator, there is no going back and once it is exhausted, it stays exhausted.

Iterators are not necessarily finite. They can also represent infinite sequences, in which case trying to exhaust them is a programming error that causes an infinite loop.

The exhausted sentinel

Termination of iteration is signalled via a sentinel value, `as.symbol(".__exhausted__.")`. Alternative designs include:

- A condition as in python.
- A rich value containing a termination flag as in Javascript.

The sentinel design is a simple and efficient solution but it has a downside. If you are iterating over a collection of elements that inadvertently contains the sentinel value, the iteration will be terminated early. To avoid such mix-ups, the sentinel should only be used as a temporary value. It should be created from scratch by a function like `coro::exhausted()` and never stored in a container or namespace.

`yield` *Yield a value from a generator*

Description

The `yield()` statement suspends `generator()` functions. It works like `return()` except that the function continues execution at the yielding point when it is called again.

`yield()` can be called within loops and if-else branches but for technical reasons it can't be used anywhere in R code:

- `yield()` cannot be called as part of a function argument. Code such as `list(yield())` is illegal.
- `yield()` does not cross function boundaries. You can't use it a lambda function passed to `lapply()` for instance.

Usage

```
yield(x)
```

Arguments

`x`	A value to yield.

See Also

`generator()` for examples.

Index