Package 'later2'

September 3, 2026

Type Package

Title Utilities for Scheduling Functions to Execute Later with Event Loops

Version 0.1

Description Executes arbitrary R or C functions some time after the current time, after the R execution stack has emptied. The functions are scheduled in an event loop. This is a derived work from the 'later' package aiming to reduce the number of dependencies.

License Apache License 2.0

URL https://github.com/pachadotdev/later2

BugReports https://github.com/pachadotdev/later2/issues

Depends R (>= 3.5)

Suggests tinytest, parallel, litedown

LinkingTo cpp4r

Encoding UTF-8

VignetteBuilder litedown

NeedsCompilation yes

Author Winston Chang [aut] (ORCID: <https://orcid.org/0000-0002-1576-2126>),
Joe Cheng [aut],
Charlie Gao [aut] (ORCID: <https://orcid.org/0000-0002-0750-061X>),
Posit Software, PBC [aut, cph] (ROR: <https://ror.org/03wc8by49>),
Marcus Geelnard [ctb, cph] (TinyCThread library,
https://tinycthread.github.io/),
Evan Nemerson [ctb, cph] (TinyCThread library,
https://tinycthread.github.io/),
Mauricio Vargas Sepulveda [aut, cre] (ORCID:
<https://orcid.org/0000-0003-1017-7574>)

Maintainer Mauricio Vargas Sepulveda <m.vargas.sepulveda@gmail.com>

Repository CRAN

Date/Publication 2026-09-03 12:50:02 UTC

Contents

`later2-package` *Utilities for Scheduling Functions to Execute Later with Event Loops*

Description

Executes arbitrary R or C functions some time after the current time, after the R execution stack has emptied. The functions are scheduled in an event loop. This is a derived work from the 'later' package aiming to reduce the number of dependencies.

`create_loop` *Private event loops*

Description

Normally, later uses a global event loop for scheduling and running functions. However, in some cases, it is useful to create a *private* event loop to schedule and execute tasks without disturbing the global event loop. For example, you might have asynchronous code that queries a remote data source, but want to wait for a full back-and-forth communication to complete before continuing in your code – from the caller's perspective, it should behave like synchronous code, and not do anything with the global event loop (which could run code unrelated to your operation). To do this, you would run your asynchronous code using a private event loop.

`create_loop` creates and returns a handle to a private event loop, which is useful when for scheduling tasks when you do not want to interfere with the global event loop.

`destroy_loop` destroys a private event loop.

`exists_loop` reports whether an event loop exists – that is, that it has not been destroyed.

`current_loop` returns the currently-active event loop. Any calls to `later()` or `run_now()` will use the current loop by default.

`with_loop` evaluates an expression with a given event loop as the currently-active loop.

`with_temp_loop` creates an event loop, makes it the current loop, then evaluates the given expression. Afterwards, the new event loop is destroyed.

`global_loop` returns a handle to the global event loop.

Usage

```
create_loop(parent = current_loop())
destroy_loop(loop)
exists_loop(loop)
current_loop()
with_temp_loop(expr)
with_loop(loop, expr)
global_loop()
```

Arguments

loop	A handle to an event loop.
expr	An expression to evaluate.
parent	The parent event loop for the one being created. Whenever the parent loop runs, this loop will also automatically run, without having to manually call run_now() on this loop. If NULL, then this loop will not have a parent event loop that automatically runs it; the only way to run this loop will be by calling run_now() on this loop.

Value

An object of class ‘event_loop‘.

---

is.promise *Determine whether an object is a promise*

Description

Determine whether an object is a promise

Usage

```
is.promise(x)
```

Arguments

x	An R object to test.

Value

‘TRUE‘ if ‘x‘ is a promise object created by [promise()], ‘FALSE‘ otherwise.

later *Executes a function later*

Description

Schedule an R function or formula to run after a specified period of time. Similar to JavaScript's 'setTimeout' function. Like JavaScript, R is single-threaded so there's no guarantee that the operation will run exactly at the requested time, only that at least that much time will elapse.

The mechanism used by this package is inspired by Simon Urbanek's [background](https://github.com/s-u/background) package and similar code in Rhttpd.

To avoid bugs due to reentrancy, by default, scheduled operations only run when there is no other R code present on the execution stack; i.e., when R is sitting at the top-level prompt. You can force past-due operations to run at a time of your choosing by calling [run_now()].

Error handling is not particularly well-defined and may change in the future. options(error=browser) should work and errors in 'func' should generally not crash the R process, but not much else can be said about it at this point. If you must have specific behavior occur in the face of errors, put error handling logic inside of 'func'.

Usage

```
later(func, delay = 0, loop = current_loop())
```

Arguments

`func`	A function or one-sided formula (e.g. '~ .x + 1', using '.' or '.x' as the argument pronoun).
`delay`	Number of seconds in the future to delay execution. There is no guarantee that the function will be executed at the desired time, but it should not execute earlier.
`loop`	A handle to an event loop. Defaults to the currently-active loop.

Value

A function, which, if invoked, will cancel the callback. The function will return `TRUE` if the callback was successfully cancelled and `FALSE` if not (this occurs if the callback has executed or has been cancelled already).

Examples

```
# Example of formula style
later(~ cat("Hello from the past\n"), 3)

# Example of function style
later(function() {
  print(summary(cars))
}, 2)
```

later_fd *Executes a function when a file descriptor is ready*

Description

Schedule an R function or formula to run after an indeterminate amount of time when file descriptors are ready for reading or writing, subject to an optional timeout.

On the occasion the system-level ‘poll‘ (on Windows ‘WSAPoll‘) returns an error, the callback will be made on a vector of all ‘NA‘s. This is indistinguishable from a case where the ‘poll‘ succeeds but there are error conditions pending against each file descriptor. If no file descriptors are supplied, the callback is scheduled for immediate execution and made on the empty logical vector ‘logical(0)‘.

Usage

```
later_fd(
  func,
  readfds = integer(),
  writefds = integer(),
  exceptfds = integer(),
  timeout = Inf,
  loop = current_loop()
)
```

Arguments

func	A function that takes a single argument, a logical vector that indicates which file descriptors are ready (a concatenation of ‘readfds‘, ‘writefds‘ and ‘exceptfds‘). This may be all ‘FALSE‘ if the ‘timeout‘ argument is non-‘Inf‘. File descriptors with error conditions pending are represented as ‘NA‘, as are invalid file descriptors such as those already closed.
readfds	Integer vector of file descriptors, or Windows SOCKETs, to monitor for being ready to read.
writefds	Integer vector of file descriptors, or Windows SOCKETs, to monitor being ready to write.
exceptfds	Integer vector of file descriptors, or Windows SOCKETs, to monitor for error conditions pending.
timeout	Number of seconds to wait before giving up, and calling ‘func‘ with all ‘FALSE‘. The default ‘Inf‘ implies waiting indefinitely. Specifying ‘0‘ will check once without blocking, and supplying a negative value defaults to a timeout of 1s.
loop	A handle to an event loop. Defaults to the currently-active loop.

Value

A delayed function evaluation.

Examples

```
## Not run:
# Unix-only example

# Use the base R 'parallel' package to fork child processes and obtain
# real, pollable file descriptors. A child's `fd` becomes ready for
# reading once the child finishes running.

# 1. timeout: prints FALSE, FALSE
job1 <- parallel::mcparallel({
  Sys.sleep(1)
  TRUE
})
job2 <- parallel::mcparallel({
  Sys.sleep(1)
  TRUE
})
fd1 <- job1$fd[1]
fd2 <- job2$fd[1]
later_fd(print, c(fd1, fd2), timeout = 0.1)
Sys.sleep(0.2)
run_now()

# 2. fd1 ready: prints TRUE, FALSE
job1 <- parallel::mcparallel(TRUE)
fd1 <- job1$fd[1]
Sys.sleep(0.1)
later_fd(print, c(fd1, fd2), timeout = 1)
Sys.sleep(0.1)
run_now()

# 3. both ready: prints TRUE, TRUE
job2 <- parallel::mcparallel(TRUE)
fd2 <- job2$fd[1]
Sys.sleep(0.1)
later_fd(print, c(fd1, fd2), timeout = 1)
Sys.sleep(0.1)
run_now()

# 4. fd2 ready: prints FALSE, TRUE
parallel::mccollect(job1)
job1 <- parallel::mcparallel({
  Sys.sleep(1)
  TRUE
})
fd1 <- job1$fd[1]
later_fd(print, c(fd1, fd2), timeout = 1)
Sys.sleep(0.1)
run_now()

# 5. fds invalid: prints NA, NA
parallel::mccollect(job1)
```

```
parallel::mccollect(job2)
later_fd(print, c(fd1, fd2), timeout = 0)
Sys.sleep(0.1)
run_now()

## End(Not run)
```

`next_op_secs` *Relative time to next scheduled operation*

Description

Returns the duration between now and the earliest operation that is currently scheduled, in seconds. If the operation is in the past, the value will be negative. If no operation is currently scheduled, the value will be 'Inf'.

Usage

```
next_op_secs(loop = current_loop())
```

Arguments

`loop`	A handle to an event loop.

Value

A numeric value.

`promise` *Create a new promise object*

Description

'promise()' creates a new promise, synchronized with later2's event loop (instead of the 'later' package's event loop, as the 'promises' package does). A promise is a placeholder object for the eventual result (or error) of an asynchronous operation.

The 'action' function should be a piece of code that returns quickly, but initiates a potentially long-running, asynchronous task. If/when the task successfully completes, call 'resolve(value)' where 'value' is the result of the computation. If the task fails, call 'reject(reason)', where 'reason' is either an error object or a character string.

Usage

```
promise(action)
```

Arguments

action	A function with signature ‘function(resolve, reject)‘.

Value

A promise object (see [then()]).

Examples

```
p1 <- promise(function(resolve, reject) {
  later(function() resolve(runif(1)), delay = 2)
})
then(p1, print)
run_now(3)
```

---

run_now *Execute scheduled operations*

Description

Normally, operations scheduled with [later()] will not execute unless/until no other R code is on the stack (i.e. at the top-level). If you need to run blocking R code for a long time and want to allow scheduled operations to run at well-defined points of your own operation, you can call ‘run_now()‘ at those points and any operations that are due to run will do so.

If one of the callbacks throws an error, the error will _not_ be caught, and subsequent callbacks will not be executed (until ‘run_now()‘ is called again, or control returns to the R prompt). You must use your own [tryCatch][base::conditions] if you want to handle errors.

Usage

```
run_now(timeoutSecs = 0L, all = TRUE, loop = current_loop())
```

Arguments

timeoutSecs	Wait (block) for up to this number of seconds waiting for an operation to be ready to run. If ‘0‘, then return immediately if there are no operations that are ready to run. If ‘Inf‘ or negative, then wait as long as it takes (if none are scheduled, then this will block forever).
all	If ‘FALSE‘, ‘run_now()‘ will execute at most one scheduled operation (instead of all eligible operations). This can be useful in cases where you want to interleave scheduled operations with your own logic.
loop	A handle to an event loop. Defaults to the currently-active loop.

Value

A logical indicating whether any callbacks were actually run.

then *Access the results of a promise*

Description

Use ‘then()‘ to access the eventual result of a promise (or, if the operation fails, the reason for that failure). The call to ‘then()‘ is non-blocking: it returns immediately, and the return value is itself a new promise.

‘catch()‘ is equivalent to ‘then()‘, but without the ‘onFulfilled‘ argument; it is typically used at the end of a promise chain to perform error handling.

‘finally()‘ is similar to ‘then()‘, but takes a single no-argument function that is executed upon completion of the promise, regardless of whether the result is success or failure. The return value of the ‘onFinally‘ callback is ignored; an error thrown from it is propagated forward into the returned promise.

Usage

```
then(promise, onFulfilled = NULL, onRejected = NULL)
catch(promise, onRejected)
finally(promise, onFinally)
```

Arguments

promise	A promise object.
onFulfilled	A function to be invoked if ‘promise‘ resolves successfully. Called with the resolved value, and optionally a second ‘.visible‘ argument indicating whether the value is [visible][base::invisible()].
onRejected	A function taking a single ‘reason‘ argument, to be invoked if ‘promise‘ fails.
onFinally	A function with no arguments, called when ‘promise‘ either succeeds or fails.

Value

A new promise.

Index