Package: promises (via r-universe)

October 28, 2026

Type Package

Title Abstractions for Promise-Based Asynchronous Programming

Version 1.5.0.9000

Description Provides fundamental abstractions for doing asynchronous programming in R using promises. Asynchronous programming is useful for allowing a single R process to orchestrate multiple tasks in the background while also attending to something else. Semantics are similar to 'JavaScript' promises, but with a syntax that is idiomatic R.

License MIT + file LICENSE

URL https://rstudio.github.io/promises/, https://github.com/rstudio/promises

BugReports https://github.com/rstudio/promises/issues

Depends R (>= 4.1.0)

Imports fastmap (>= 1.1.0), later, lifecycle, magrittr (>= 1.5), otel (>= 0.2.0), R6, rlang

Suggests future (>= 1.21.0), knitr, mirai, otelsdk (>= 0.2.0), purrr, rmarkdown, spelling, testthat (>= 3.0.0), vembedr

VignetteBuilder knitr, rmarkdown

Config/Needs/website brand.yml, rsconnect, tidyverse/tidytemplate

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.1

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-05-27

Encoding UTF-8

Language en-US

Repository https://posit-dev-shinycoreci.r-universe.dev

Date/Publication 2026-10-28 19:46:01 UTC

RemoteUrl https://github.com/rstudio/promises

RemoteRef HEAD

RemoteSha 281e6a5ab666e41d42604a571a92dc10b635793b

Contents

future_promise_queue 2
hybrid_then 5
is.promise 7
pipes 8
promise 9
promise_all 11
promise_map 12
promise_reduce 13
promise_resolve 14
then 14
wait_for 17
with_otel_span 18
with_promise_domain 24

Index **26**

`future_promise_queue` **future** *promise*

Description

When submitting **future** work, **future** (by design) will block the main R session until a worker becomes available. This occurs when there is more submitted **future** work than there are available **future** workers. To counter this situation, we can create a promise to execute work using future (using `future_promise()`) and only begin the work if a **future** worker is available.

Usage

```
future_promise_queue()

future_promise(
  expr = NULL,
  envir = parent.frame(),
  ...,
  substitute = TRUE,
  queue = future_promise_queue()
)
```

Arguments

expr	An R expression. While the expr is eventually sent to future::future(), please use the same precautions that you would use with regular promises::promise() expressions. future_promise() may have to hold the expr in a promise() while waiting for a **future** worker to become available.
envir	The environment from where global objects should be identified.
...	extra parameters provided to future::future()
substitute	If TRUE, argument expr is substitute():d, otherwise not.
queue	A queue that is used to schedule work to be done using future::future(). This queue defaults to future_promise_queue() and requires that method queue$schedule_work(fn) exist. This method should take in a function that will execute the promised **future** work.

Details

Using future_promise() is recommended whenever a continuous runtime is used, such as with **plumber** or **shiny**.

For more details and examples, please see the vignette("future_promise", "promises") vignette.

Value

Unlike future::future(), future_promise() returns a promise() object that will eventually resolve the **future** expr.

Functions

- future_promise_queue(): Default future_promise() work queue to use. This function returns a WorkQueue that is cached per R session.
- future_promise(): Creates a promise() that will execute the expr using future::future().

See Also

WorkQueue

Examples

```
## Not run: # Relative start time
start <- Sys.time()
# Helper to force two `future` workers
with_two_workers <- function(expr) {
  if (!require("future")) {
    message("`future` not installed")
    return()
  }
  old_plan <- future::plan(future::multisession, workers = 2)
  on.exit({future::plan(old_plan)}, add = TRUE)
  start <<- Sys.time()
```

```
  force(expr)
  while(!later::loop_empty()) {Sys.sleep(0.1); later::run_now()}
  invisible()
}
# Print a status message. Ex: `"PID: XXX; 2.5s promise done"`
print_msg <- function(pid, msg) {
  message(
    "PID: ", pid, "; ",
    round(difftime(Sys.time(), start, units = "secs"), digits = 1), "s " ,
    msg
  )
}

# `"promise done"` will appear after four workers are done and the main R session is not blocked
# The important thing to note is the first four times will be roughly the same
with_two_workers({
  promise_resolve(Sys.getpid()) |>
    then(\(x) {print_msg("promise done")})
  for (i in 1:6) {
    future::future({Sys.sleep(1); Sys.getpid()}) |>
      then(\(x) {print_msg("future done")})
  }
})
{
#> PID: XXX; 2.5s promise done
#> PID: YYY; 2.6s future done
#> PID: ZZZ; 2.6s future done
#> PID: YYY; 2.6s future done
#> PID: ZZZ; 2.6s future done
#> PID: YYY; 3.4s future done
#> PID: ZZZ; 3.6s future done
}

# `"promise done"` will almost immediately, before any workers have completed
# The first two `"future done"` comments appear earlier the example above
with_two_workers({
  promise_resolve(Sys.getpid()) |>
    then(\(x) {print_msg("promise")})
  for (i in 1:6) {
    future_promise({Sys.sleep(1); Sys.getpid()}) |>
      then(\(x) {print_msg("future done")})
  }
})
{
#> PID: XXX; 0.2s promise done
#> PID: YYY; 1.3s future done
#> PID: ZZZ; 1.4s future done
#> PID: YYY; 2.5s future done
#> PID: ZZZ; 2.6s future done
#> PID: YYY; 3.4s future done
#> PID: ZZZ; 3.6s future done
}
## End(Not run)
```

hybrid_then *Asynchronous or synchronous* then()

Description

This is a helper function that behaves like then, however if is.promising() returns FALSE then the handlers will be executed immediately.

Usage

```
hybrid_then(expr, on_success = NULL, on_failure = NULL, ..., tee = FALSE)
```

Arguments

expr	An expression that evaluates to either a promise or a non-promise value.
on_success	A function to be called when no error occurs synchronously or asynchronously. When invoked, the function will be called with a single argument: the resolved value. Optionally, the function can take a second parameter .visible if you care whether the promise was resolved with a visible or invisible value. Can return a value or a promise.
on_failure	A function to be called if an error occurs synchronously or asynchronously. Takes one argument: the error object. Can return a value or a promise to recover from the error, or throw a new error. If on_failure is provided and doesn't throw an error (or return a promise that fails) then this is the async equivalent of catching an error.
...	Reserved for future use. Currently must be empty.
tee	If TRUE, ignore the return value of the callback, and use the original value of expr as the result. For on_failure with tee = TRUE, the callback executes but the original error is re-thrown afterward.

Details

Execution paths:

- If expr evaluates to a promise (p), it will call p |> then(on_success, on_failure).
- If expr evaluates to a non-promise value (x), it will call on_success(x).
- If expr throws an error (e) during calculation, it will call on_failure(e).

In all cases, the on_success and on_failure callbacks are executed (when provided).

Value

- If expr evaluates to a promise, a promise with a single followup promise to handle the on_success or on_failure callbacks.
- If expr evaluates to a non-promise value, the result of the synchronous operation after being processed by on_success or on_failure.
- If a callback returns a promise (and tee = FALSE), the result is always a promise.

Utility

This function is useful for writing functions that need to execute followup behavior *now* or within a promise. This is different behavior than `then()` where *everything* is made into a promise.

`hybrid_then()` allows authors to keep synchronous execution on the same *tick* without requiring the use of a followup promise. This is particularly appealing for situations where the author does not control the execution flow for items that may be either synchronous or asynchronous, such as within `{plumber2}`.

Error Handling

If no `on_failure` callback is provided and an error occurs, the error is re-thrown immediately (for synchronous errors) or propagated through the returned promise (for asynchronous errors).

`on_failure` only handles a failure of `expr`. An error thrown by `on_success` is not passed to `on_failure`; it is re-thrown (sync) or rejects the returned promise (async), like `then()`.

If an `on_failure` callback is provided but it throws an error, that new error replaces the original error. With `tee = TRUE`, even if `on_failure` executes successfully, the original error is still re-thrown.

Callback Return Values

Callbacks can return any value, including promises. If a callback returns a promise, the entire `hybrid_then()` call will return a promise, even if the input was synchronous. This allows seamless transitions between synchronous and asynchronous execution.

With `tee = TRUE`, the callback's return value is ignored, even if it is a promise: it is not waited on, and its result or error does not change the result. A synchronous input still gives a synchronous result.

See Also

`then()`, `is.promising()`, `promise_resolve()`

Examples

```
# Basic usage - works with both sync and async values
add_to <- function(x, k) {
  hybrid_then(
    x,
    on_success = function(value) {
      value + k
    },
    on_failure = function(err) {
      message("Error: ", err$message)
      NA_real_
    }
  )
}

# Synchronous
42 |> add_to(100)
```

```
#> [1] 142

# Synchronous error
add_to({stop("Bad input!")}, 8)
#> Error: Bad input!
#> [1] NA

## Not run:
# Asynchronous
promise_resolve(42) |>
  add_to(8) |>
  then(print)
# When resolved...
#> [1] 50

# Error handling - asynchronous
promise_resolve(stop("Bad async input!")) |>
  add_to(8) |>
  then(print)
# When resolved...
#> Error: Bad async input!
#> [1] NA

# Chaining multiple operations
# (Move the `promise_resolve()` around to see sync vs async behavior)
1 |>
  hybrid_then(on_success = \(x) x + 1) |>
  hybrid_then(on_success = \(x) promise_resolve(x * 2)) |>
  hybrid_then(on_success = \(x) x - 1) |>
  hybrid_then(print)
# When resolved...
#> [1] 3

## End(Not run)
```

`is.promise` *Coerce to a promise*

Description

Use `is.promise` to determine whether an R object is a promise. Use `as.promise` (an S3 generic method) to attempt to coerce an R object to a promise, and `is.promising` (another S3 generic method) to test whether `as.promise` is supported. mirai::mirai objects have an `as.promise` method defined in the mirai package, and this package provides one for converting future::Future objects into promises.

Usage

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

```
is.promising(x)

as.promise(x)
```

Arguments

x	An R object to test or coerce.

Value

`as.promise` returns a promise object, or throws an error if the object cannot be converted.

`is.promise` returns `TRUE` if the given value is a promise object, and `FALSE` otherwise.

`is.promising` returns `TRUE` if the given value is a promise object or if it can be converted to a promise object using `as.promise`, and `FALSE` otherwise.

---

pipes *Promise pipe operators*

Description

With R 4.1, the promise pipe operators are **[Superseded]** by `then`, `catch`, and `finally` methods when used in tandem with the function shorthand (`\(x) rhs(x)`) and `|>`.

Usage

```
lhs %...>% rhs

lhs %...T>% rhs

lhs %...!% rhs

lhs %...T!% rhs
```

Arguments

lhs	A promise object.
rhs	A function call using the magrittr semantics. It can return either a promise or non-promise value, or throw an error.

Details

Promise-aware pipe operators, in the style of magrittr. Like magrittr pipes, these operators can be used to chain together pipelines of promise-transforming operations. Unlike magrittr pipes, these pipes wait for promise resolution and pass the unwrapped value (or error) to the `rhs` function call.

The `>` variants are for handling successful resolution, the `!` variants are for handling errors. The `T` variants of each return the lhs instead of the rhs, which is useful for pipeline steps that are used for side effects (printing, plotting, saving).

1. `promise %...>% func()` is equivalent to `promise %>% then(func)`.
2. `promise %...!% func()` is equivalent to `promise %>% catch(func)`.
3. `promise %...T>% func()` is equivalent to `promise %T>% then(func)`.
4. `promise %...T!% func()` is equivalent to `promise %T>% catch(func)` or `promise %>% catch(func, tee = TRUE)`.

One situation where 3. and 4. above break down is when `func()` throws an error, or returns a promise that ultimately fails. In that case, the failure will be propagated by our pipe operators but not by the magrittr-plus-function "equivalents".

For simplicity of implementation, we do not support the magrittr feature of using a `.` at the head of a pipeline to turn the entire pipeline into a function instead of an expression.

Value

A new promise.

See Also

https://rstudio.github.io/promises/articles/promises_03_overview.html#using-pipes

Examples

```
## Not run:
library(mirai)

mirai(cars) %...>%
  head(5) %...T>%
  print()

# If the read.csv fails, resolve to NULL instead
mirai(read.csv("http://example.com/data.csv")) %...!%
  { NULL }

## End(Not run)
```

---

promise *Create a new promise object*

Description

`promise()` creates a new promise. A promise is a placeholder object for the eventual result (or error) of an asynchronous operation. This function is not generally needed to carry out asynchronous programming tasks; instead, it is intended to be used mostly by package authors who want to write asynchronous functions that return promises.

Usage

```
promise(action)
```

Arguments

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

Details

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 (like the return value). If the task fails, call reject(reason), where reason is either an error object, or a character string.

It's important that asynchronous tasks kicked off from action be coded very carefully–in particular, all errors must be caught and passed to reject(). Failure to do so will cause those errors to be lost, at best; and the caller of the asynchronous task will never receive a response (the asynchronous equivalent of a function call that never returns, i.e. hangs).

The return value of action will be ignored.

Value

A promise object (see then).

action= formulas

[Superseded]

With {promises} depending on R >= 4.1, the shorthand of a formula, ~ fn(.) for action is no longer recommended by the {promises} package or tidyverse (for example, {purrr}) as we now have access to the function shorthand, \(x) fn(x). Please update your action code to use the new function shorthand syntax \(resolve, reject) resolve(arg1, arg2) instead of ~ { resolve(arg1, arg2) }. The magically created resolve/reject functions can be confusing when chained with other methods.

Examples

```
# Create a promise that resolves to a random value after 2 secs
p1 <- promise(\(resolve, reject) {
  later::later(\() resolve(runif(1)), delay = 2)
})

p1 |> then(print)

# Create a promise that errors immediately
p2 <- promise(\(resolve, reject) {
  reject("An error has occurred")
})
then(p2,
  onFulfilled = \(value) message("Success"),
  onRejected = \(err) message("Failure")
)
```

promise_all *Combine multiple promise objects*

Description

Use promise_all to wait for multiple promise objects to all be successfully fulfilled. Use promise_race to wait for the first of multiple promise objects to be either fulfilled or rejected.

Usage

```
promise_all(..., .list = NULL)

promise_race(..., .list = NULL)
```

Arguments

...	Promise objects. Either all arguments must be named, or all arguments must be unnamed. If .list is provided, then these arguments are ignored.
.list	A list of promise objects–an alternative to ....

Value

A promise.

For promise_all, if all of the promises were successful, the returned promise will resolve to a list of the promises' values; if any promise fails, the first error to be encountered will be used to reject the returned promise.

For promise_race, the first of the promises to either fulfill or reject will be passed through to the returned promise. As in JavaScript, racing no promises returns a promise that never settles.

Examples

```
p1 <- promise(\(resolve, reject) later::later(\() resolve(1), delay = 1))
p2 <- promise(\(resolve, reject) later::later(\() resolve(2), delay = 2))

# Resolves after 1 second, to the value: 1
promise_race(p1, p2) |>
  then(\(x) {
    cat("promise_race:\n")
    str(x)
  })

# Resolves after 2 seconds, to the value: list(1, 2)
promise_all(p1, p2) |>
  then(\(x) {
    cat("promise_all:\n")
    str(x)
  })
```

promise_map *Promise-aware lapply/map*

Description

Similar to base::lapply() or purrr::map, but promise-aware: the .f function is permitted to return promises, and while lapply returns a list, promise_map returns a promise that resolves to a similar list (of resolved values only, no promises).

Usage

```
promise_map(.x, .f, ...)
```

Arguments

.x	A vector (atomic or list) or an expression object (but not a promise). Other objects (including classed objects) will be coerced by base::as.list.
.f	The function to be applied to each element of .x. The function is permitted, but not required, to return a promise.
...	Optional arguments to .f.

Details

promise_map processes elements of .x serially; that is, if .f(.x[[1]]) returns a promise, then .f(.x[[2]]) will not be invoked until that promise is resolved. If any such promise rejects (errors), then the promise returned by promise_map immediately rejects with that err.

Value

A promise that resolves to a list (of values, not promises).

Examples

```
# Waits x seconds, then returns x*10
wait_this_long <- function(x) {
  promise(\(resolve, reject) {
    later::later(\() resolve(x*10), delay = x)
  })
}

promise_map(
  list(A=1, B=2, C=3),
  wait_this_long
) |>
  then(print)
```

promise_reduce *Promise-aware version of Reduce*

Description

Similar to purrr::reduce (left fold), but the function .f is permitted to return a promise. promise_reduce will wait for any returned promise to resolve before invoking .f with the next element; in other words, execution is serial. .f can return a promise as output but should never encounter a promise as input (unless .x itself is a list of promises to begin with, in which case the second parameter would be a promise).

Usage

```
promise_reduce(.x, .f, ..., .init)
```

Arguments

.x	A vector or list to reduce. (Not a promise.)
.f	A function that takes two parameters. The first parameter will be the "result" (initially .init, and then set to the result of the most recent call to func), and the second parameter will be an element of .x.
...	Other arguments to pass to .f
.init	The initial result value of the fold, passed into .f when it is first executed. If missing, the first element of .x is used and the fold starts with the second element. If .init is missing and .x is empty, promise_reduce() throws an error right away instead of returning a rejected promise, as this is a mistake in the call itself.

Value

A promise that will resolve to the result of calling .f on the last element (or .init if .x had no elements). If any invocation of .f results in an error or a rejected promise, then the overall promise_reduce promise will immediately reject with that error.

Examples

```
# Returns a promise for the sum of e1 + e2, with a 0.5 sec delay
slowly_add <- function(e1, e2) {
  promise(\(resolve, reject) {
    later::later(\() resolve(e1 + e2), delay = 0.5)
  })
}

# Prints 55 after a little over 5 seconds
promise_reduce(1:10, slowly_add, .init = 0) |>
  then(print)
```

promise_resolve *Create a resolved or rejected promise*

Description

Helper functions to conveniently create a promise that is resolved to the given value (or rejected with the given reason).

Usage

```
promise_resolve(value)

promise_reject(reason)
```

Arguments

value	A value, or promise, that the new promise should be resolved to. This expression will be lazily evaluated, and if evaluating the expression raises an error, then the new promise will be rejected with that error as the reason.
reason	An error message string, or error object.

Examples

```
promise_resolve(mtcars) |>
  then(head) |>
  then(print)

promise_reject("Something went wrong") |>
  catch(tee = TRUE, \(e) message(conditionMessage(e)))
```

then *Access the results of a promise*

Description

Use the `then` function to access the eventual result of a promise (or, if the operation fails, the reason for that failure). Regardless of the state of the promise, the call to `then` is non-blocking, that is, it returns immediately; so what it does *not* do is immediately return the result value of the promise. Instead, you pass logic you want to execute to `then`, in the form of function callbacks. If you provide an `onFulfilled` callback, it will be called upon the promise's successful resolution, with a single argument `value`: the result value. If you provide an `onRejected` callback, it will be called if the operation fails, with a single argument `reason`: the error that caused the failure.

Usage

```
then(promise, onFulfilled = NULL, onRejected = NULL, ..., tee = FALSE)

catch(promise, onRejected, ..., tee = FALSE)

finally(promise, onFinally)
```

Arguments

promise	A promise object. The object can be in any state.
onFulfilled	A function that will be invoked if the promise value successfully resolves. When invoked, the function will be called with a single argument: the resolved value. Optionally, the function can take a second parameter `.visible` if you care whether the promise was resolved with a visible or invisible value. The function can return a value or a promise object, or can throw an error; these will affect the resolution of the promise object that is returned by `then()`.
onRejected	A function taking the argument `error`. The function can return a value or a promise object, or can throw an error. If `onRejected` is provided and doesn't throw an error (or return a promise that fails) then this is the async equivalent of catching an error.
...	Ignored.
tee	If `TRUE`, ignore the return value of the callback, and use the original value instead. This is useful for performing operations with side-effects, particularly logging to the console or a file. If the callback itself throws an error, and `tee` is `TRUE`, that error will still be used to fulfill the the returned promise (in other words, `tee` only has an effect if the callback does not throw).
onFinally	A function with no arguments, to be called when the async operation either succeeds or fails. Usually used for freeing resources that were used during async operations.

Formulas

[Superseded]

With `{promises}` depending on R >= 4.1, the shorthand of a formula, `~ fn(.)` is no longer recommended by the `{promises}` package or tidyverse (for example, `{purrr}`) as we now have access to the function shorthand, `\(x) fn(x)`. Please update your code to use the new function shorthand syntax `\(x) fn(x, arg1, args2)` instead of `~ fn(., arg1, arg2)`. The `.` can be confusing when chained with other methods.

Chaining promises

The first parameter of `then` is a promise; given the stated purpose of the function, this should be no surprise. However, what may be surprising is that the return value of `then` is also a (newly created) promise. This new promise waits for the original promise to be fulfilled or rejected, and for `onFulfilled` or `onRejected` to be called. The result of (or error raised by) calling `onFulfilled`/`onRejected` will be used to fulfill (reject) the new promise.

```
promise_a <- get_data_frame_async()
promise_b <- then(promise_a, onFulfilled = head)
```

In this example, assuming `get_data_frame_async` returns a promise that eventually resolves to a data frame, `promise_b` will eventually resolve to the first 10 or fewer rows of that data frame.

Note that the new promise is considered fulfilled or rejected based on whether `onFulfilled`/`onRejected` returns a value or throws an error, not on whether the original promise was fulfilled or rejected. In other words, it's possible to turn failure to success and success to failure. Consider this example, where we expect `some_async_operation` to fail, and want to consider it an error if it doesn't:

```
promise_c <- some_async_operation()
promise_d <- then(promise_c,
  onFulfilled = function(value) {
    stop("That's strange, the operation didn't fail!")
  },
  onRejected = function(reason) {
    # Great, the operation failed as expected
    NULL
  }
)
```

Now, `promise_d` will be rejected if `promise_c` is fulfilled, and vice versa.

Warning: Be very careful not to accidentally turn failure into success, if your error handling code is not the last item in a chain!

```
some_async_operation() |>
  catch(function(reason) {
    warning("An error occurred: ", reason)
  }) |>
  then(function() {
    message("I guess we succeeded...?")  # No!
  })
```

In this example, the `catch` callback does not itself throw an error, so the subsequent `then` call will consider its promise fulfilled!

Convenience functions

For readability and convenience, we provide `catch` and `finally` functions.

The `catch` function is equivalent to `then`, but without the `onFulfilled` argument. It is typically used at the end of a promise chain to perform error handling/logging.

The `finally` function is similar to `then`, but takes a single no-argument function that will be executed upon completion of the promise, regardless of whether the result is success or failure. It is typically used at the end of a promise chain to perform cleanup tasks, like closing file handles or database connections. Unlike `then` and `catch`, the return value of `finally` is ignored; however, if an error is thrown in `finally`, that error will be propagated forward into the returned promise.

Unhandled errors

If a promise rejects and no `onRejected` handler is attached by the time the event loop has run twice after the rejection, promises prints `Unhandled promise error: <message>` to standard error. A handler attached later still runs, but the message has already been printed. Attach handlers in the same tick that creates the promise to avoid the message.

Visibility

`onFulfilled` functions can optionally have a second parameter `.visible`, which will be `FALSE` if the result value is invisible.

`wait_for` *Wait for a Promise*

Description

Synchronizes a promise by waiting for it to complete and then retrieving its value.

Usage

```
wait_for(promise, ..., timeout = Inf, loop = later::current_loop())
```

Arguments

`promise`	A promise object, or an object that can be converted to one with `as.promise()` (such as a future or a mirai).
`...`	Ignored.
`timeout`	The most seconds to wait. If the promise has not settled by then, an error of class `promises_timeout` is thrown and the promise is left pending. Defaults to `Inf` (wait forever). With `0`, only an already settled promise returns.
`loop`	the **later** loop on which the promise has been scheduled. Defaults to `later::current_loop()`.

Value

The value of the resolved promise. An error will be thrown if the promise is rejected.

Event loop

`wait_for()` runs a single event loop, given by `loop`. This must be the loop that runs the promise's callbacks. If it is not, `wait_for()` waits until `timeout`, which is forever by default. The default, `later::current_loop()`, is the right choice whenever you did not create a custom loop yourself. Do not call `wait_for()` from inside a `later` callback to wait on work that depends on that callback finishing.

Interrupts

An interrupt (Esc in RStudio, Ctrl+C in a terminal) means "stop waiting". It exits `wait_for()` and leaves the promise pending. Callbacks that have not started yet stay on the loop and run the next time the loop runs. If the interrupt lands while a callback is running, that callback stops part way and is not run again, so promises that depend on it may never settle. `wait_for()` does not cancel the underlying work, because only the creator of the work knows how to cancel it.

Examples

```
# Returns a promise for the sum of e1 + e2, with a 1 sec delay
slowly_add <- function(e1, e2) {
  promise(\(resolve, reject) {
    later::later(\() resolve(e1 + e2), delay = 1)
  })
}

# Returns 3 after 1 second
res <- wait_for(slowly_add(1, 2))
res
```

`with_otel_span` *OpenTelemetry integration*

Description

otel provides tools for integrating with OpenTelemetry, a framework for observability and tracing in distributed systems.

These methods are intended to enhance the framework to be used with the **promises** package, not as a generic replacement.

Developer note - Barret 2025/09: This otel span handoff promise domain topic is complex and has been discussed over many hours. Many advanced Shiny/R developers are not even aware of promise domains (very reasonable!), therefore this topic requires more in-depth documentation and examples.

Usage

```
with_otel_span(name, expr, ..., tracer, attributes = NULL)

with_otel_promise_domain(expr)

local_otel_promise_domain(envir = parent.frame())
```

Arguments

name	Character string. The name of the otel span.
expr	An expression to evaluate within the otel span context.
...	Additional arguments passed to otel::start_span().
tracer	(Required) An {otel} tracer. It is required to provide your own tracer from your own package. See otel::get_tracer() for more details.
attributes	Attributes passed through otel::as_attributes() (when not NULL)
envir	The "local" environment in which to add the promise domain. When the environment is exited, the promise domain is removed.

Functions

- with_otel_span(): Creates an OpenTelemetry span, executes the given expression within it, and ends the span.

  This method **requires** the use of with_otel_promise_domain() to be within the execution stack.

  This function is designed to handle both synchronous and asynchronous (promise-based) operations. For promises, the span is automatically ended when the promise resolves or rejects.

  Returns the result of evaluating expr. If expr returns a promise, the span will be automatically ended when the promise completes.

  This function differs from synchronous otel span operations in that it installs a promise domain and properly handles asynchronous operations. In addition, the internal span will be ended either when the function exits (for synchronous operations) or when a returned promise completes (for asynchronous operations).

  If OpenTelemetry is not enabled, the expression will be evaluated without any tracing context.

- with_otel_promise_domain(): Adds an idempotent handoff Active OpenTelemetry span promise domain.

  Package authors are required to use this function to have otel span context persist across asynchronous boundaries. This method is only needed once per promise domain stack. If you are unsure, feel free to call with_otel_promise_domain() as the underlying promise domain will only be added if not found within the current promise domain stack. If your package **only** works within Shiny apps, Shiny will have already added the domain so no need to add it yourself. If your package works outside of Shiny and you use {promises} (i.e. {chromote}), then you'll need to use this wrapper method.

  This method adds a *handoff* Active OpenTelemetry span promise domain to the expression evaluation. This *handoff* promise domain will only run once on reactivation. This is critical if there are many layered with_otel_span() calls, such as within Shiny reactivity. For example, if we nested many with_otel_span() calls of which each call added a promise domain that reactivated each otel span on restore, we'd reactivate k otel span objects (O(k)) when we only need to activate the **last** span (O(1)).

  Returns the result of evaluating expr within the otel span promise domain.

- local_otel_promise_domain(): Local OpenTelemetry span promise domain

  Adds an OpenTelemetry span promise domain to the local scope. This is useful for {coro} operations where encapsulating the coro operations inside a with_*() methods is not allowed.

  When not using {coro}, please prefer to use with_otel_span() or with_otel_promise_domain().

Definitions

- Promise domain: An environment in which has setup/teardown methods. These environments can be composed together to facilitate execution context for promises. In normal R execution, this can be achieved with scope / stack. But for complex situations, such as the currently open graphics device, async operations require promise domains to setup/teardown these contexts to function properly. Otherwise a multi-stage promise that adds to the graphics device at each stage will only ever print to the most recently created graphics device, not the associated graphics device. These promise domains are not automatically created, they must be manually added to the execution stack, for example `with_otel_promise_domain()` does this for OpenTelemetry spans.
- Promise domain restoration: When switching from one promise chain to another, the execution context is torn down and then re-established. This re-establishment is called "promise domain restoration". During this process, the promise domains are restored in their previously established combination order.
- Promise chain: A set of promise objects to execute over multiple async ticks.
- Async tick: the number of times an event loop must run to move computation forward. (Similar to a JavaScript event loop tick.)
- `then()` promise domain capture: When `then()` is called, it will capture the current promise domain. This promise domain is restored (only if needed) when evaluating the given `onFulfilled` and `onRejected` callbacks. This captured promise domain does not go into any downstream promise chain objects. The only way the promise domain is captured is exactly when the `then()` method is called.

`with_otel_promise_domain()` creates a promise domain that restores the currently active OpenTelemetry span from when a call to `promises::then()` is executed. Given the special circumstance where only the current otel span is needed to continue recording (not a full ancestry tree of otel spans), we can capture *just* the current otel span and reactivate that otel span during promise domain restoration.

When promise domains are captured

Asynchronous operation

- Creates `async_op` otel span
- Automatically ends the otel span (`async_op`) when the promise (p) resolves or rejects

The code below illustrates an example of when the promise domain are created/captured/restored and when otel span objects are created/activated/reactivated/ended.

```
# t0.0
p2 <- with_otel_promise_domain({
  # t0.1
  p <- with_otel_span("async_op", {
    # ... return a promise ...
    init_async_work() |> # t0.2
      then( # t0.3
        some_async_work # t1.0
      )
```

```
  }) # t0.4, t1.0, t2.0
  p |>
    then( # t0.5
      more_async_work # t3.0
    )
}) # t0.6

p_final <-
  p2 |> then( # t0.7
    final_work # t4.0
  )
```

An in-depth explanation of the execution timeline is below.

- At the first initial tick, `t0.*`:
  - `t0.0`: The code is wrapped in `with_otel_promise_domain()`
  - `t0.1`: The `async_op` otel span is created and activated
  - `t0.2`: Some async work is initiated
  - `t0.3`: `then()` is called, capturing the active `async_op` otel span (as it is called within `with_otel_promise_domain()`)
  - `t0.4`: The `with_otel_span()` call exits, but the `async_op` otel span is not ended as the promise is still pending. The returned promise has a `finally()` step added to it that will end the otel span `async_op` when `p` is resolved.
  - `t0.5`: Another `then()` is called, but there is no active otel span to capture
  - `t0.6`: The otel span promise domain call exits
  - `t0.7`: Another `then()` is called. No otel span will be captured as there is no active otel span / promise domain
- At the first followup tick, `t1.0`:
  - The active `async_op` otel span is reactivated during promise domain restoration for the duration of the `then` callback
  - The `some_async_work` function is called
- At tick, `t2.0`:
  - `some_async_work` has resolved
  - A hidden `finally()` step closes the otel span, `async_op`
  - `p` is now resolved
- At tick, `t3.0`:
  - There is no active otel span at `t0.5`, so no otel span is reactivated during promise domain restoration
  - The `more_async_work` function is executed
- At tick, `t4.0`:
  - `more_async_work` has resolved, therefore `p2` is now resolved
  - There was no otel span promise domain at `t0.7`, so no attempt is made to reactivate any otel span
  - The `final_work` function is executed
- At tick, `t5.0`:
  - `p_final` has resolved

Complexity

When reactivating the kth step in a promise chain, the currently active otel span (during the call to `then()`) will be reactivated during promise domain restoration (`O(1)`). To restore a chain of promises, the active otel span will be restored at each step (`O(n)`) due to the `n` calls to wrapping each `onFulfilled` and `onRejected` callbacks inside `then()`.

If we did NOT have a handoff promise domain for otel span restoration, a regular promise domain approach would be needed at each step to restore the active otel span. Each step would call `with_active_span()` `k` times (`O(k)`, where as handoff domain computes in `O(1)`). Taking a step back, to restore each otel span at for every step in a promise chain would then take `O(n^2)` time, not `O(n)`. The standard, naive promise domain approach does not scale for multiple similar promise domain restorations.

Execution model for `with_otel_promise_domain()`

1. `with_otel_promise_domain(expr)` is called.
   - The following steps all occur within `expr`.
2. Create an otel span object using `otel::start_span()`.
   - We need the otel span to be active during the a followup async operation. Therefore, `otel::start_local_active_span()` is not appropriate as the otel span would be ended when the function exits, not when the promise chain resolves.
3. Be sure your otel span is activated before calling `promises::then()`.
   - Activate it using `with_otel_span(name, expr)` (which also creates/ends the otel span) or `otel::with_active_span(span, expr)`.
4. Call `promises::then()`

- When `promises::then()` is called, the two methods (`onFulfilled` and `onRejected`) capture the currently active spans. (Performed by the initial `with_otel_promise_domain()`)

1. During reactivation of the promise chain step, the previously captured otel span is reactivated via `with_active_span()`. (Performed by the initial `with_otel_promise_domain()`)

OpenTelemetry span compatibility

For otel span objects to exist over may async ticks, the otel span must be created using `otel::start_span()` and later ended using `otel::end_span()`. Ending the otel span must occur **after** any promise chain work has completed.

If we were to instead use `otel::start_local_active_span()`, the otel span would be ended when the function exits, not when the promise chain completes. Even though the local otel span is created, activated, and eventually ended, the otel span will not exist during reactivation of the otel span promise domain.

`with_otel_span()` is a convenience method that creates, activates, and ends the otel span only after the returned promise (if any) resolves. It also properly handles both synchronous (ending the otel span within `on.exit()`) and asynchronous operations (ending the otel span within `promises::finally()`).

See Also

otel::start_span(), otel::with_active_span(), otel::end_span()

Examples

```
## Not run:
# Common usage:
with_otel_promise_domain({
  # ... deep inside some code execution ...

  # Many calls to `with_otel_span()` within `with_otel_promise_domain()`
  with_otel_span("my_operation", {
    # ... do some work ...
  })
})

## End(Not run)
## Not run:
with_otel_promise_domain({
  # ... deep inside some code execution ...

  # Synchronous operation
  # * Creates `my_operation` span
  result <- with_otel_span("my_operation", {
    # ... do some work ...
    print(otel::get_active_span()$name) # "my_operation"

    # Nest (many) more spans
    prom_nested <- with_otel_span("my_nested_operation", {
      # ... do some more work ...
      promise_resolve(42) |>
        then(\(value) {
          print(otel::get_active_span()$name) # "my_nested_operation"
          print(value) # 42
        })
    })

    # Since `then()` is called during the active `my_operation` span,
    # the `my_operation` span will be reactivated in the `then()` callback.
    prom_nested |> then(\(value) {
      print(otel::get_active_span()$name) # "my_operation"
      value
    })
  })

  # Since `then()` is called where there is no active span,
  # there is no _active_ span in the `then()` callback.
  result |> then(\(value) {
    stopifnot(inherits(otel::get_active_span(), "otel_span_noop"))
    print(value) # 42
  })
})
```

```
## End(Not run)
```

`with_promise_domain` *Promise domains*

Description

Promise domains are used to temporarily set up custom environments that intercept and influence the registration of callbacks. Create new promise domain objects using `new_promise_domain`, and temporarily activate a promise domain object (for the duration of evaluating a given expression) using `with_promise_domain`.

Usage

```
with_promise_domain(domain, expr, replace = FALSE)

new_promise_domain(
  wrapOnFulfilled = identity,
  wrapOnRejected = identity,
  wrapSync = force,
  onError = force,
  ...,
  wrapOnFinally = NULL
)
```

Arguments

`domain`	A promise domain object to install while `expr` is evaluated.
`expr`	Any R expression, to be evaluated under the influence of `domain`.
`replace`	If `FALSE`, then the effect of the `domain` will be added to the effect of any currently active promise domain(s). If `TRUE`, then the current promise domain(s) will be ignored for the duration of the `with_promise_domain` call.
`wrapOnFulfilled`	A function that takes a single argument: a function that was passed as an `onFulfilled` argument to `then()`. The `wrapOnFulfilled` function should return a function that is suitable for `onFulfilled` duty.
`wrapOnRejected`	A function that takes a single argument: a function that was passed as an `onRejected` argument to `then()`. The `wrapOnRejected` function should return a function that is suitable for `onRejected` duty.
`wrapSync`	A function that takes a single argument: a (lazily evaluated) expression that the function should `force()`. This expression represents the `expr` argument passed to `with_promise_domain()`; `wrapSync` allows the domain to manipulate the environment before/after `expr` is evaluated.

onError	A function that takes a single argument: an error. onError will be called whenever an exception occurs in a domain (that isn't caught by a tryCatch). Providing an onError callback doesn't cause errors to be caught, necessarily; instead, onError callbacks behave like calling handlers.
...	Arbitrary named values that will become elements of the promise domain object, and can be accessed as items in an environment (i.e. using [[or $).
wrapOnFinally	A function that takes a single argument: a function that was passed as an onFinally argument to then(). The wrapOnFinally function should return a function that is suitable for onFinally duty. If wrapOnFinally is NULL (the default), then the domain will use both wrapOnFulfilled and wrapOnRejected to wrap the onFinally. If it's important to distinguish between normal fulfillment/rejection handlers and finally handlers, then be sure to provide wrapOnFinally, even if it's just base::identity().

Details

While with_promise_domain is on the call stack, any calls to then() (or higher level functions or operators, like catch()) will belong to the promise domain. In addition, when a then callback that belongs to a promise domain is invoked, then any new calls to then will also belong to that promise domain. In other words, a promise domain "infects" not only the immediate calls to then, but also to "nested" calls to then.

For more background, read the original design doc.

For examples, see the source code of the Shiny package, which uses promise domains extensively to manage graphics devices and reactivity.

Index

%...!% (pipes), 8
%...>% (pipes), 8
%...T!% (pipes), 8
%...T>% (pipes), 8

as.promise (is.promise), 7
as.promise(), *17*

base::identity(), *25*
base::lapply(), *12*

catch, *8*
catch (then), 14
catch(), *25*

environment, *3*

finally, *8*
finally (then), 14
force(), *24*
future::Future, *7*
future::future(), *3*
future_promise (future_promise_queue), 2
future_promise_queue, 2
future_promise_queue(), *3*

hybrid_then, 5

invisible, *17*
is.promise, 7
is.promising (is.promise), 7
is.promising(), *6*

later::current_loop(), *17*
local_otel_promise_domain
        (with_otel_span), 18

mirai::mirai, *7*

new_promise_domain
        (with_promise_domain), 24

otel::as_attributes(), *19*
otel::end_span(), *23*
otel::get_tracer(), *19*
otel::start_span(), *19*, *23*
otel::with_active_span(), *23*

pipes, 8
promise, 9
promise(), *3*
promise_all, 11
promise_map, 12
promise_race (promise_all), 11
promise_reduce, 13
promise_reject (promise_resolve), 14
promise_resolve, 14
promise_resolve(), *6*
purrr::map, *12*
purrr::reduce, *13*

substitute, *3*

then, *8*, *10*, 14
then(), *6*, *24*, *25*

wait_for, 17
with_otel_promise_domain
        (with_otel_span), 18
with_otel_span, 18
with_promise_domain, 24
with_promise_domain(), *24*
WorkQueue, *3*