Package: promises (via r-universe)

November 26, 2024

Type Package

Title Abstractions for Promise-Based Asynchronous Programming

Version 1.3.1

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

Imports fastmap (>= 1.1.0), later, magrittr (>= 1.5), R6, Rcpp, rlang, stats

Suggests future (>= 1.21.0), knitr, purrr, rmarkdown, spelling, testthat, vembedr

LinkingTo later, Rcpp

VignetteBuilder knitr

Config/Needs/website rsconnect

Encoding UTF-8

Language en-US

RoxygenNote 7.3.2

NeedsCompilation yes

Author Joe Cheng [aut, cre], Posit Software, PBC [cph, fnd]

Maintainer Joe Cheng <joe@posit.co>

Repository CRAN

Date/Publication 2024-11-26 08:40:02 UTC

Contents

`future_promise_queue` **future** *promise*

Description

[Experimental]

Usage

```
future_promise_queue()

future_promise(
  expr = NULL,
  envir = parent.frame(),
  substitute = TRUE,
  globals = TRUE,
  packages = NULL,
  ...,
  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.
`substitute`	If TRUE, argument `expr` is `substitute()`:ed, otherwise not.
`globals`	(optional) a logical, a character vector, or a named list to control how globals are handled. For details, see section 'Globals used by future expressions' in the help for `future()`.

packages — (optional) a character vector specifying packages to be attached in the R environment evaluating the future.

... — extra parameters provided to `future::future()`

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

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.

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

```
# 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()) %...>% print_msg("promise done")
 for (i in 1:6) future::future({Sys.sleep(1); Sys.getpid()}) %...>% 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()) %...>% print_msg("promise")
 for (i in 1:6) future_promise({Sys.sleep(1); Sys.getpid()}) %...>% 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
}
```

`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. This package includes support 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

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.

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

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/overview.html#using-pipes

Examples

```
## Not run:
library(future)
plan(multisession)

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

# If the read.csv fails, resolve to NULL instead
future_promise(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)`, or a one-sided formula. See Details.

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`).

Examples

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

p1 %...>% print()

# Create a promise that errors immediately
p2 <- promise(~{
  reject("An error has occurred")
})
```

```
then(p2,
  onFulfilled = ~message("Success"),
  onRejected = ~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.

Examples

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

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

# Resolves after 2 seconds, to the value: list(1, 2)
promise_all(p1, p2) %...>% {
```

```
  cat("promise_all:\n")
  str(.)
}
```

`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(~later::later(~{
    resolve(x*10)
  }, delay = x))
}

promise_map(list(A=1, B=2, C=3), wait_this_long) %...>%
  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.

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(~later::later(~resolve(e1 + e2), delay = 0.5))
}

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

promise_reject("Something went wrong") %...T!%
  { message(conditionMessage(.)) }
```

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 (or formulas, see Details). 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)

catch(promise, onRejected, tee = FALSE)

finally(promise, onFinally)
```

Arguments

promise	A promise object. The object can be in any state.
onFulfilled	A function (or a formula–see Details) 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` (or a formula–see Details). 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.
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

For convenience, the `then()`, `catch()`, and `finally()` functions use `rlang::as_function()` to convert `onFulfilled`, `onRejected`, and `onFinally` arguments to functions. This means that you can use formulas to create very compact anonymous functions, using `.` to access the value (in the case of `onFulfilled`) or error (in the case of `onRejected`).

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 (or formula) 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.

Visibility

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

`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()` or the various pipes) 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