Package 'furrr'

May 16, 2018

Version 0.1.0

Title Apply Mapping Functions in Parallel using Futures

Depends R (>= 3.2.0), future (>= 1.6.2)

Imports globals (>= 0.10.3), rlang (>= 0.2.0), purrr (>= 0.2.4)

Suggests listenv (>= 0.6.0), dplyr (>= 0.7.4), testthat

Maintainer Davis Vaughan <dvaughan@business-science.io>

Description
Implementations of the family of map() functions from 'purrr' that can be resolved using any 'future'-supported backend, e.g. parallel on the local machine or distributed on a compute cluster.

License LGPL (>= 2.1)

LazyLoad TRUE

URL https://github.com/DavisVaughan/furrr

BugReports https://github.com/DavisVaughan/furrr/issues

RoxygenNote 6.0.1

NeedsCompilation no

Author Davis Vaughan [aut, cre],
Matt Dancho [aut]

Repository CRAN

Date/Publication 2018-05-16 16:04:29 UTC

R topics documented:

`fold` *Efficient fold / reduce / accumulate / combine of a vector*

Description

This function comes from the `future.apply` package.

Usage

```
fold(x, f, left = TRUE, unname = TRUE, threshold = 1000L)
```

Arguments

`x`	A vector.
`f`	A binary function, i.e. a function take takes two arguments.
`left`	If `TRUE`, vector is combined from the left (the first element), otherwise the right (the last element).
`unname`	If `TRUE`, function `f` is called as `f(unname(y), x[[ii]])`, otherwise as `f(y, x[[ii]])`, which may introduce name `"y"`. [[ii]: R:[ii [[ii]: R:[ii
`threshold`	An integer (>= 2) specifying the length where the recursive divide'and'conquer call will stop and incremental building of the partial value is performed. Using `threshold = +Inf` will disable recursive folding.

Details

In order for recursive folding to give the same results as non-recursive folding, binary function `f` must be *associative* with itself, i.e. `f(f(x[[1]], x[[2]]), x[[3]])` equals `f(x[[1]], f(x[[2]]), x[[3]])`.

This function is a more efficient (memory and speed) of [`base::Reduce(f, x, right = !left, accumulate = FALSE)`][ba
especially when `x` is long.

[[1]: R:[1 [[2]: R:[2 [[3]: R:[3 [[1]: R:[1 [[2]: R:[2 [[3]: R:[3 [base::Reduce]: R:base::Reduce

Value

A vector.

future_imap *Apply a function to each element of a vector, and its index via futures*

Description

These functions work exactly the same as purrr::imap() functions, but allow you to map in parallel.

Usage

```
future_imap(.x, .f, ..., .progress = FALSE, .options = future_options())

future_imap_chr(.x, .f, ..., .progress = FALSE, .options = future_options())

future_imap_dbl(.x, .f, ..., .progress = FALSE, .options = future_options())

future_imap_int(.x, .f, ..., .progress = FALSE, .options = future_options())

future_imap_lgl(.x, .f, ..., .progress = FALSE, .options = future_options())

future_imap_dfr(.x, .f, ..., .id = NULL, .progress = FALSE,
  .options = future_options())

future_imap_dfc(.x, .f, ..., .progress = FALSE, .options = future_options())
```

Arguments

.x	A list or atomic vector.
.f	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. ~ .x + 2, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use . • For a two argument function, use .x and .y • For more arguments, use ..1, ..2, ..3 etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. Within a list, wrap strings in get-attr() to extract named attributes. If a component is not present, the value of .default will be returned.
...	Additional arguments passed on to .f.
.progress	A logical, for whether or not to print a progress bar for multiprocess, multisession, and multicore plans.

.options	The future specific options to use with the workers. This must be the result from a call to future_options().
.id	If not NULL a variable with this name will be created giving either the name or the index of the data frame.

Value

A vector the same length as .x.

Examples

```
library(furrr)

plan(multiprocess)


future_imap_chr(sample(10), ~ paste0(.y, ": ", .x))
```

---

future_invoke_map *Invoke functions via futures*

Description

These functions work exactly the same as purrr::invoke_map() functions, but allow you to invoke in parallel.

Usage

```
future_invoke_map(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())

future_invoke_map_chr(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())

future_invoke_map_dbl(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())

future_invoke_map_int(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())

future_invoke_map_lgl(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())

future_invoke_map_dfr(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())
```

```
future_invoke_map_dfc(.f, .x = list(NULL), ..., .env = NULL,
  .progress = FALSE, .options = future_options())
```

Arguments

.f	A list of functions.
.x	A list of argument-lists the same length as .f (or length 1). The default argument, list(NULL), will be recycled to the same length as .f, and will call each function with no arguments (apart from any supplied in ....)
...	Additional arguments passed to each function.
.env	Environment in which do.call() should evaluate a constructed expression. This only matters if you pass as .f the name of a function rather than its value, or as .x symbols of objects rather than their values.
.progress	A logical, for whether or not to print a progress bar for multiprocess, multisession, and multicore plans.
.options	The future specific options to use with the workers. This must be the result from a call to future_options().

Examples

```
plan(multiprocess)


df <- dplyr::tibble(
  f = c("runif", "rpois", "rnorm"),
  params = list(
    list(n = 10),
    list(n = 5, lambda = 10),
    list(n = 10, mean = -3, sd = 10)
  )
)

future_invoke_map(df$f, df$params)
```

---

future_map *Apply a function to each element of a vector via futures*

Description

These functions work exactly the same as purrr::map() functions, but allow you to run the map in parallel. The documentation is adapted from both purrr::map(), and future.apply::future_lapply(), so look there for more details.

Usage

```
future_map(.x, .f, ..., .progress = FALSE, .options = future_options())

future_map_chr(.x, .f, ..., .progress = FALSE, .options = future_options())

future_map_dbl(.x, .f, ..., .progress = FALSE, .options = future_options())

future_map_int(.x, .f, ..., .progress = FALSE, .options = future_options())

future_map_lgl(.x, .f, ..., .progress = FALSE, .options = future_options())

future_map_dfr(.x, .f, ..., .id = NULL, .progress = FALSE,
  .options = future_options())

future_map_dfc(.x, .f, ..., .progress = FALSE, .options = future_options())

future_map_if(.x, .p, .f, ..., .progress = FALSE,
  .options = future_options())

future_map_at(.x, .at, .f, ..., .progress = FALSE,
  .options = future_options())
```

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. Within a list, wrap strings in `get-attr()` to extract named attributes. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to `.f`.
`.progress`	A logical, for whether or not to print a progress bar for multiprocess, multisession, and multicore plans.
`.options`	The `future` specific options to use with the workers. This must be the result from a call to `future_options()`.
`.id`	If not `NULL` a variable with this name will be created giving either the name or the index of the data frame.

.p	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as .x. Alternatively, if the elements of .x are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where .p evaluates to TRUE will be modified.
.at	A character vector of names or a numeric vector of positions. Only those elements corresponding to .at will be modified.

Value

All functions return a vector the same length as .x.

future_map() returns a list, future_map_lgl() a logical vector, future_map_int() an integer vector, future_map_dbl() a double vector, and future_map_chr() a character vector. The output of .f will be automatically typed upwards, e.g. logical -> integer -> double -> character.

Examples

```
library(furrr)
library(dplyr) # for the pipe


plan(multiprocess)


1:10 %>%
  future_map(rnorm, n = 10) %>%
  future_map_dbl(mean)

# If each element of the output is a data frame, use
# future_map_dfr to row-bind them together:
mtcars %>%
  split(.$cyl) %>%
  future_map(~ lm(mpg ~ wt, data = .x)) %>%
  future_map_dfr(~ as.data.frame(t(as.matrix(coef(.)))))

# You can be explicit about what gets exported to the workers

# To see this, use multisession (NOT multicore if on a Mac as the forked workers
# still have access to this environment)

plan(multisession)

x <- 1
y <- 2

# This will fail, y is not exported (no black magic occurs)
# future_map(1, ~y, .options = future_options(globals = "x"))

# y is exported
future_map(1, ~y, .options = future_options(globals = "y"))
```

future_map2 *Map over multiple inputs simultaneously via futures*

Description

These functions work exactly the same as purrr::map2() functions, but allow you to run the map in parallel. Note that "parallel" as described in purrr is just saying that you are working with multiple inputs, and parallel in this case means that you can work on multiple inputs AND process them all in parallel as well.

Usage

```
future_map2(.x, .y, .f, ..., .progress = FALSE, .options = future_options())

future_map2_chr(.x, .y, .f, ..., .progress = FALSE,
  .options = future_options())

future_map2_dbl(.x, .y, .f, ..., .progress = FALSE,
  .options = future_options())

future_map2_int(.x, .y, .f, ..., .progress = FALSE,
  .options = future_options())

future_map2_lgl(.x, .y, .f, ..., .progress = FALSE,
  .options = future_options())

future_map2_dfr(.x, .y, .f, ..., .id = NULL, .progress = FALSE,
  .options = future_options())

future_map2_dfc(.x, .y, .f, ..., .progress = FALSE,
  .options = future_options())

future_pmap(.l, .f, ..., .progress = FALSE, .options = future_options())

future_pmap_chr(.l, .f, ..., .progress = FALSE, .options = future_options())

future_pmap_dbl(.l, .f, ..., .progress = FALSE, .options = future_options())

future_pmap_int(.l, .f, ..., .progress = FALSE, .options = future_options())

future_pmap_lgl(.l, .f, ..., .progress = FALSE, .options = future_options())

future_pmap_dfr(.l, .f, ..., .id = NULL, .progress = FALSE,
  .options = future_options())

future_pmap_dfc(.l, .f, ..., .progress = FALSE, .options = future_options())
```

Arguments

`.x`	Vectors of the same length. A vector of length 1 will be recycled.
`.y`	Vectors of the same length. A vector of length 1 will be recycled.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. Within a list, wrap strings in `get-attr()` to extract named attributes. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to `.f`.
`.progress`	A logical, for whether or not to print a progress bar for multiprocess, multisession, and multicore plans.
`.options`	The `future` specific options to use with the workers. This must be the result from a call to `future_options()`.
`.id`	If not `NULL` a variable with this name will be created giving either the name or the index of the data frame.
`.l`	A list of lists. The length of `.l` determines the number of arguments that `.f` will be called with. List names will be used if present.

Value

An atomic vector, list, or data frame, depending on the suffix. Atomic vectors and lists will be named if `.x` or the first element of `.l` is named.

If all input is length 0, the output will be length 0. If any input is length 1, it will be recycled to the length of the longest.

Examples

```
library(furrr)

plan(multiprocess)


x <- list(1, 10, 100)
y <- list(1, 2, 3)
z <- list(5, 50, 500)
```

```
future_map2(x, y, ~ .x + .y)

# Split into pieces, fit model to each piece, then predict
by_cyl <- split(mtcars, mtcars$cyl)
mods <- future_map(by_cyl, ~ lm(mpg ~ wt, data = .))
future_map2(mods, by_cyl, predict)

future_pmap(list(x, y, z), sum)

# Matching arguments by position
future_pmap(list(x, y, z), function(a, b ,c) a / (b + c))

# Vectorizing a function over multiple arguments
df <- data.frame(
  x = c("apple", "banana", "cherry"),
  pattern = c("p", "n", "h"),
  replacement = c("x", "f", "q"),
  stringsAsFactors = FALSE
)
future_pmap(df, gsub)
future_pmap_chr(df, gsub)
```

`future_modify` *Modify elements selectively via futures*

Description

These functions work exactly the same as `purrr::modify()` functions, but allow you to modify in parallel.

Usage

```
future_modify(.x, .f, ..., .progress = FALSE, .options = future_options())

future_modify_at(.x, .at, .f, ..., .progress = FALSE,
  .options = future_options())

future_modify_if(.x, .p, .f, ..., .progress = FALSE,
  .options = future_options())
```

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. Within a list, wrap strings in `get-attr()` to extract named attributes. If a component is not present, the value of `.default` will be returned.

`...` Additional arguments passed on to `.f`.

`.progress` A logical, for whether or not to print a progress bar for multiprocess, multisession, and multicore plans.

`.options` The `future` specific options to use with the workers. This must be the result from a call to `future_options()`.

`.at` A character vector of names or a numeric vector of positions. Only those elements corresponding to `.at` will be modified.

`.p` A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as `.x`. Alternatively, if the elements of `.x` are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where `.p` evaluates to `TRUE` will be modified.

Details

From `purrr`) Since the transformation can alter the structure of the input; it's your responsibility to ensure that the transformation produces a valid output. For example, if you're modifying a data frame, `.f` must preserve the length of the input.

Value

An object the same class as .x

Examples

```
library(furrr)
library(dplyr) # for the pipe


plan(multiprocess)


# Convert each col to character, in parallel
future_modify(mtcars, as.character)

iris %>%
 future_modify_if(is.factor, as.character) %>%
 str()
```

```
mtcars %>% future_modify_at(c(1, 4, 5), as.character) %>% str()
```

---

future_options `future` *specific options*

Description

These options are used by `future()` internally to tweak the environment that the expressions are called in. The most important ones are `globals` and `packages` which allow you to be explicit about the variables and packages that are exported to each worker.

Usage

```
future_options(globals = TRUE, packages = NULL, seed = FALSE,
  lazy = FALSE, scheduling = 1)
```

Arguments

`globals`	A logical, a character vector, or a named list for controlling how globals are handled. For details, see `Global variables and packages`.
`packages`	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
`seed`	A logical or an integer (of length one or seven), or a list of `length(.x)` with pre-generated random seeds. For details, see below section.
`lazy`	Specifies whether the futures should be resolved lazily or eagerly (default).
`scheduling`	Average number of futures ("chunks") per worker. If `0.0`, then a single future is used to process all elements of `.x`. If `1.0` or `TRUE`, then one future per worker is used. If `2.0`, then each worker will process two futures (if there are enough elements in `.x`). If `Inf` or `FALSE`, then one future per element of `.x` is used.

Global variables and packages

By default, the `future` package will perform black magic to look up the global variables and packages that your `furrr` call requires, and it will export these to each worker. However, it is not always perfect, and can be refined with the `globals` and `packages` arguments.

`globals` may be used to control how globals should be handled similarly how the `globals` argument is used with `future()`. Since all function calls use the same set of globals, this function can do any gathering of globals upfront (once), which is more efficient than if it would be done for each future independently.

- If `TRUE` or `NULL`, then globals are automatically identified and gathered.
- If a character vector of names is specified, then those globals are gathered.
- If a named list, then those globals are used as is.
- In all cases, `.f` and any `...` arguments are automatically passed as globals to each future created as they are always needed.

packages may be used to control the packages that are exported to each worker.

- If a character vector of packages names is specified, those are exported to each worker.
- In all cases, purrr is exported, as it is always required on each worker.

Reproducible random number generation (RNG)

Unless seed = FALSE, this function guarantees to generate the exact same sequence of random numbers *given the same initial seed / RNG state* - this regardless of type of futures and scheduling ("chunking") strategy.

RNG reproducibility is achieved by pregenerating the random seeds for all iterations (over .x) by using L'Ecuyer-CMRG RNG streams. In each iteration, these seeds are set before calling .f(.x[[ii]], ...). *Note, for large* length(.x) *this may introduce a large overhead.*

As input (seed), a fixed seed (integer) may be given, either as a full L'Ecuyer-CMRG RNG seed (vector of 1+6 integers) or as a seed generating such a full L'Ecuyer-CMRG seed. If seed = TRUE, then .Random.seed is returned if it holds a L'Ecuyer-CMRG RNG seed, otherwise one is created randomly. If seed = NA, a L'Ecuyer-CMRG RNG seed is randomly created. If none of the function calls .f(.x[[ii]], ...) uses random number generation, then seed = FALSE may be used.

In addition to the above, it is possible to specify a pre-generated sequence of RNG seeds as a list such that length(seed) == length(.x) and where each element is an integer seed that can be assigned to .Random.seed. Use this alternative with caution. **Note that** as.list(seq_along(.x)) **is *not* a valid set of such** .Random.seed **values.**

In all cases but seed = FALSE, the RNG state of the calling R processes after this function returns is guaranteed to be "forwarded one step" from the RNG state that was before the call and in the same way regardless of seed, scheduling and future strategy used. This is done in order to guarantee that an R script calling future_map() multiple times should be numerically reproducible given the same initial seed.

Index