Package 'crew'

March 25, 2024

Title A Distributed Worker Launcher Framework

Description In computationally demanding analysis projects,
statisticians and data scientists asynchronously
deploy long-running tasks to distributed systems,
ranging from traditional clusters to cloud services.
The 'NNG'-powered 'mirai' R package by Gao (2023)
<doi:10.5281/zenodo.7912722> is a sleek
and sophisticated scheduler that
efficiently processes these intense workloads.
The 'crew' package extends 'mirai' with a unifying
interface for third-party worker launchers.
Inspiration also comes from packages.
'future' by Bengtsson (2021) <doi:10.32614/RJ-2021-048>,
'rrq' by FitzJohn and Ashton (2023) <https://github.com/mrc-ide/rrq>,
'clustermq' by Schubert (2019) <doi:10.1093/bioinformatics/btz284>),
and 'batchtools' by Lang, Bischel, and Surmann (2017)
<doi:10.21105/joss.00135>.

Version 0.9.1

License MIT + file LICENSE

URL https://wlandau.github.io/crew/, https://github.com/wlandau/crew

BugReports https://github.com/wlandau/crew/issues

Depends R (>= 4.0.0)

Imports cli (>= 3.1.0), data.table, getip, later, mirai (>= 0.12.0),
nanonext (>= 0.12.0), processx, promises, ps, R6, rlang, stats,
tibble, tidyselect, tools, utils

Suggests knitr (>= 1.30), markdown (>= 1.1), rmarkdown (>= 2.4),
testthat (>= 3.0.0)

Encoding UTF-8

Language en-US

VignetteBuilder knitr

Config/testthat/edition 3

RoxygenNote 7.3.1

NeedsCompilation no

Author William Michael Landau [aut, cre]
(<https://orcid.org/0000-0003-1878-3253>),
Daniel Woodie [ctb],
Eli Lilly and Company [cph]

Maintainer William Michael Landau <will.landau.oss@gmail.com>

Repository CRAN

Date/Publication 2024-03-25 16:40:02 UTC

R topics documented:

crew-package *crew: a distributed worker launcher framework*

Description

In computationally demanding analysis projects, statisticians and data scientists asynchronously deploy long-running tasks to distributed systems, ranging from traditional clusters to cloud services. The NNG-powered `mirai` R package is a sleek and sophisticated scheduler that efficiently processes these intense workloads. The `crew` package extends `mirai` with a unifying interface for third-party worker launchers. Inspiration also comes from packages `future`, `rrq`, `clustermq`, and `batchtools`.

crew_assert *Crew assertion*

Description

Assert that a condition is true.

Usage

```
crew_assert(value = NULL, ..., message = NULL, envir = parent.frame())
```

Arguments

`value`	An object or condition.
`...`	Conditions that use the `"."` symbol to refer to the object.
`message`	Optional message to print on error.
`envir`	Environment to evaluate the condition.

Value

`NULL` (invisibly). Throws an error if the condition is not true.

See Also

Other utility: `crew_clean()`, `crew_deprecate()`, `crew_eval()`, `crew_random_name()`, `crew_retry()`, `crew_terminate_process()`, `crew_terminate_signal()`, `crew_worker()`

Examples

```
crew_assert(1 < 2)
crew_assert("object", !anyNA(.), nzchar(.))
tryCatch(
  crew_assert(2 < 1),
  crew_error = function(condition) message("false")
)
```

crew_async *Local asynchronous client object.*

Description

Create an R6 object to manage local asynchronous quick tasks with error detection.

Usage

```
crew_async(workers = NULL)
```

Arguments

workers	Number of local mirai daemons to run asynchronous tasks. If NULL, then tasks will be evaluated synchronously.

Details

crew_async() objects are created inside launchers to allow launcher plugins to run local tasks asynchronously, such as calls to cloud APIs to launch serious remote workers.

Value

An R6 async client object.

See Also

Other async: crew_class_async

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
x <- crew_async()
x$start()
out <- x$eval(1 + 1)
mirai::call_mirai_(out)
out$data # 2
x$terminate()
}
```

crew_class_async R6 *async class.*

Description

R6 class for async configuration.

Details

See crew_async().

Active bindings

workers See crew_async().

instance Name of the current instance.

Methods

Public methods:

- crew_class_async$new()
- crew_class_async$validate()
- crew_class_async$start()
- crew_class_async$terminate()
- crew_class_async$started()
- crew_class_async$asynchronous()
- crew_class_async$eval()

Method new(): TLS configuration constructor.

Usage:

```
crew_class_async$new(workers = NULL)
```

Arguments:

workers Argument passed from crew_async().

Returns: An R6 object with TLS configuration.

Method validate(): Validate the object.

Usage:

```
crew_class_async$validate()
```

Returns: NULL (invisibly).

Method start(): Start the local workers and error handling socket.

Usage:

```
crew_class_async$start()
```

Details: Does not create workers or an error handling socket if `workers` is `NULL` or the object is already started.

Returns: `NULL` (invisibly).

Method `terminate()`**:** Start the local workers and error handling socket.

Usage:

```
crew_class_async$terminate()
```

Details: Waits for existing tasks to complete first.

Returns: `NULL` (invisibly).

Method `started()`**:** Show whether the object is started.

Usage:

```
crew_class_async$started()
```

Returns: Logical of length 1, whether the object is started.

Method `asynchronous()`**:** Show whether the object is asynchronous (has real workers).

Usage:

```
crew_class_async$asynchronous()
```

Returns: Logical of length 1, whether the object is asynchronous.

Method `eval()`**:** Run a local asynchronous task using a local compute profile.

Usage:

```
crew_class_async$eval(
  command,
  substitute = TRUE,
  data = list(),
  packages = character(0L),
  library = NULL
)
```

Arguments:

`command` R code to run.

`substitute` Logical of length 1, whether to substitute `command`. If `FALSE`, then `command` must be an expression object or language object.

`data` Named list of data objects required to run `command`.

`packages` Character vector of packages to load.

`library` Character vector of library paths to load the packages from.

Details: Used for launcher plugins with asynchronous launches and terminations. If `processes` is `NULL`, the task will run locally. Otherwise, the task will run on a local process in the local `mirai` compute profile.

Returns: If the `processes` field is `NULL`, a list with an object named `data` containing the result of evaluating `expr` synchronously. Otherwise, the task is evaluated asynchronously, and the result is a `mirai` task object. Either way, the `data` element of the return value will contain the result of the task.

See Also

Other async: `crew_async()`

crew_class_client R6 *client class.*

Description

R6 class for mirai clients.

Details

See crew_client().

Active bindings

name See crew_client().

workers See crew_client().

host See crew_client().

port See crew_client().

tls See crew_client().

seconds_interval See crew_client().

seconds_timeout See crew_client().

relay Relay object for event-driven programming on a downstream condition variable.

started Whether the client is started.

dispatcher Process ID of the mirai dispatcher

Methods

Public methods:

- crew_class_client$new()
- crew_class_client$validate()
- crew_class_client$start()
- crew_class_client$terminate()
- crew_class_client$condition()
- crew_class_client$resolved()
- crew_class_client$summary()

Method new(): mirai client constructor.

Usage:

```
crew_class_client$new(
  name = NULL,
  workers = NULL,
  host = NULL,
  port = NULL,
  tls = NULL,
```

```
  seconds_interval = NULL,
  seconds_timeout = NULL,
  relay = NULL
)
```

Arguments:

`name` Argument passed from `crew_client()`.

`workers` Argument passed from `crew_client()`.

`host` Argument passed from `crew_client()`.

`port` Argument passed from `crew_client()`.

`tls` Argument passed from `crew_client()`.

`seconds_interval` Argument passed from `crew_client()`.

`seconds_timeout` Argument passed from `crew_client()`.

`relay` Argument passed from `crew_client()`.

Returns: An `R6` object with the client.

Examples:

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
client$log()
client$terminate()
}
```

Method `validate()`**:** Validate the client.

Usage:

```
crew_class_client$validate()
```

Returns: `NULL` (invisibly).

Method `start()`**:** Start listening for workers on the available sockets.

Usage:

```
crew_class_client$start()
```

Returns: `NULL` (invisibly).

Method `terminate()`**:** Stop the mirai client and disconnect from the worker websockets.

Usage:

```
crew_class_client$terminate()
```

Returns: `NULL` (invisibly).

Method `condition()`**:** Get the `nanonext` condition variable which tasks signal on resolution.

Usage:

```
crew_class_client$condition()
```

Returns: The `nanonext` condition variable which tasks signal on resolution. The return value is `NULL` if the client is not running.

Method `resolved()`**:** Get the true value of the `nanonext` condition variable.

Usage:

```
crew_class_client$resolved()
```

Details: Subtracts a safety offset which was padded on start.

Returns: The value of the `nanonext` condition variable.

Method `summary()`**:** Show an informative worker log.

Usage:

```
crew_class_client$summary()
```

Returns: A `tibble` with information on the workers, or `NULL` if the client is not started. The `tibble` has 1 row per worker and the following columns:

- `worker`: integer index of the worker.
- `online`: `TRUE` if the worker is online and connected to the websocket URL, `FALSE` otherwise.
- `instances`: integer, number of instances of `mirai` daemons (`crew` workers) that have connected to the websocket URL during the life cycle of the listener.
- `assigned`: number of tasks assigned to the current websocket URL.
- `complete`: number of tasks completed at the current websocket URL.
- `socket`: websocket URL. `crew` changes the token at the end of the URL path periodically as a safeguard while managing workers.

See Also

Other client: `crew_client()`

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
client$log()
client$terminate()
}

## ------------------------------------------------
## Method `crew_class_client$new`
## ------------------------------------------------

if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
client$log()
client$terminate()
}
```

`crew_class_controller` *Controller class*

Description

R6 class for controllers.

Details

See `crew_controller()`.

Active bindings

`client` Router object.

`launcher` Launcher object.

`tasks` A list of `mirai::mirai()` task objects.

`pushed` Number of tasks pushed since the controller was started.

`popped` Number of tasks popped since the controller was started.

`log` Tibble with per-worker metadata about tasks.

`error` Tibble of task results (with one result per row) from the last call to `map(error = "stop)`.

`backlog` Character vector of explicitly backlogged tasks.

Methods

Public methods:

- `crew_class_controller$new()`
- `crew_class_controller$validate()`
- `crew_class_controller$empty()`
- `crew_class_controller$nonempty()`
- `crew_class_controller$resolved()`
- `crew_class_controller$unresolved()`
- `crew_class_controller$unpopped()`
- `crew_class_controller$saturated()`
- `crew_class_controller$start()`
- `crew_class_controller$launch()`
- `crew_class_controller$scale()`
- `crew_class_controller$push()`
- `crew_class_controller$walk()`
- `crew_class_controller$map()`
- `crew_class_controller$pop()`
- `crew_class_controller$collect()`
- `crew_class_controller$promise()`

- crew_class_controller$wait()
- crew_class_controller$push_backlog()
- crew_class_controller$pop_backlog()
- crew_class_controller$summary()
- crew_class_controller$terminate()

Method `new()`**:** `mirai` controller constructor.

Usage:

```
crew_class_controller$new(client = NULL, launcher = NULL)
```

Arguments:

`client` Router object. See crew_controller().

`launcher` Launcher object. See crew_controller().

Returns: An R6 controller object.

Examples:

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
launcher <- crew_launcher_local()
controller <- crew_controller(client = client, launcher = launcher)
controller$start()
controller$push(name = "task", command = sqrt(4))
controller$wait()
controller$pop()
controller$terminate()
}
```

Method `validate()`**:** Validate the client.

Usage:

```
crew_class_controller$validate()
```

Returns: `NULL` (invisibly).

Method `empty()`**:** Check if the controller is empty.

Usage:

```
crew_class_controller$empty(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: A controller is empty if it has no running tasks or completed tasks waiting to be retrieved with `push()`.

Returns: `TRUE` if the controller is empty, `FALSE` otherwise.

Method `nonempty()`**:** Check if the controller is nonempty.

Usage:

```
crew_class_controller$nonempty(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: A controller is empty if it has no running tasks or completed tasks waiting to be retrieved with `push()`.

Returns: `TRUE` if the controller is empty, `FALSE` otherwise.

Method `resolved()`**:** Number of resolved `mirai()` tasks.

Usage:

```
crew_class_controller$resolved(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: `resolved()` is cumulative: it counts all the resolved tasks over the entire lifetime of the controller session.

Returns: Non-negative integer of length 1, number of resolved `mirai()` tasks. The return value is 0 if the condition variable does not exist (i.e. if the client is not running).

Method `unresolved()`**:** Number of unresolved `mirai()` tasks.

Usage:

```
crew_class_controller$unresolved(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: Non-negative integer of length 1, number of unresolved `mirai()` tasks.

Method `unpopped()`**:** Number of resolved `mirai()` tasks available via `pop()`.

Usage:

```
crew_class_controller$unpopped(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: Non-negative integer of length 1, number of resolved `mirai()` tasks available via `pop()`.

Method `saturated()`**:** Check if the controller is saturated.

Usage:

```
crew_class_controller$saturated(
  collect = NULL,
  throttle = NULL,
  controller = NULL
)
```

Arguments:

`collect` Deprecated in version 0.5.0.9003 (2023-10-02). Not used.

`throttle` Deprecated in version 0.5.0.9003 (2023-10-02). Not used.

`controller` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: A controller is saturated if the number of unresolved tasks is greater than or equal to the maximum number of workers. In other words, in a saturated controller, every available worker has a task. You can still push tasks to a saturated controller, but tools that use `crew` such as `targets` may choose not to.

Returns: `TRUE` if the controller is saturated, `FALSE` otherwise.

Method `start()`**:** Start the controller if it is not already started.

Usage:

```
crew_class_controller$start(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: Register the mirai client and register worker websockets with the launcher.

Returns: `NULL` (invisibly).

Method `launch()`**:** Launch one or more workers.

Usage:

```
crew_class_controller$launch(n = 1L, controllers = NULL)
```

Arguments:

`n` Number of workers to try to launch. The actual number launched is capped so that no more than "`workers`" workers running at a given time, where "`workers`" is an argument of `crew_controller()`. The actual cap is the "`workers`" argument minus the number of connected workers minus the number of starting workers. A "connected" worker has an active websocket connection to the `mirai` client, and "starting" means that the worker was launched at most `seconds_start` seconds ago, where `seconds_start` is also an argument of `crew_controller()`.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: `NULL` (invisibly).

Method `scale()`**:** Auto-scale workers out to meet the demand of tasks.

Usage:

```
crew_class_controller$scale(throttle = TRUE, controllers = NULL)
```

Arguments:

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: The `scale()` method re-launches all inactive backlogged workers, then any additional inactive workers needed to accommodate the demand of unresolved tasks. A worker is "backlogged" if it was assigned more tasks than it has completed so far.
Methods `push()`, `pop()`, and `wait()` already invoke `scale()` if the `scale` argument is `TRUE`. For finer control of the number of workers launched, call `launch()` on the controller with the exact desired number of workers.

Returns: `NULL` (invisibly).

Method `push()`**:** Push a task to the head of the task list.

Usage:

```
crew_class_controller$push(
  command,
  data = list(),
  globals = list(),
  substitute = TRUE,
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  seconds_timeout = NULL,
  scale = TRUE,
  throttle = TRUE,
  name = NA_character_,
  save_command = FALSE,
  controller = NULL
)
```

Arguments:

`command` Language object with R code to run.

`data` Named list of local data objects in the evaluation environment.

`globals` Named list of objects to temporarily assign to the global environment for the task. This list should include any functions you previously defined in the global environment which are required to run tasks. See the `reset_globals` argument of `crew_controller_local()`.

`substitute` Logical of length 1, whether to call `base::substitute()` on the supplied value of the `command` argument. If `TRUE` (default) then `command` is quoted literally as you write it, e.g. `push(command = your_function_call())`. If `FALSE`, then `crew` assumes `command` is a language object and you are passing its value, e.g. `push(command = quote(your_function_call()))`. `substitute = TRUE` is appropriate for interactive use, whereas `substitute = FALSE` is meant for automated R programs that invoke `crew` controllers.

`seed` Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the `seed` argument of `set.seed()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`algorithm` Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the `kind` argument of `RNGkind()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

packages Character vector of packages to load for the task.

library Library path to load the packages. See the lib.loc argument of require().

seconds_timeout Optional task timeout passed to the .timeout argument of mirai::mirai() (after converting to milliseconds).

scale Logical, whether to automatically call scale() to auto-scale workers to meet the demand of the task load. Also see the throttle argument.

throttle TRUE to skip auto-scaling if it already happened within the last seconds_interval seconds. FALSE to auto-scale every time scale() is called. Throttling avoids overburdening the mirai dispatcher and other resources.

name Optional name of the task.

save_command Logical of length 1. If TRUE, the controller deparses the command and returns it with the output on pop(). If FALSE (default), the controller skips this step to increase speed.

controller Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: Invisibly return the mirai object of the pushed task. This allows you to interact with the task directly, e.g. to create a promise object with promises::as.promise().

Method `walk()`**:** Apply a single command to multiple inputs, and return control to the user without waiting for any task to complete.

Usage:

```
crew_class_controller$walk(
  command,
  iterate,
  data = list(),
  globals = list(),
  substitute = TRUE,
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  seconds_timeout = NULL,
  names = NULL,
  save_command = FALSE,
  scale = TRUE,
  throttle = TRUE,
  controller = NULL
)
```

Arguments:

command Language object with R code to run.

iterate Named list of vectors or lists to iterate over. For example, to run function calls f(x = 1, y = "a") and f(x = 2, y = "b"), set command to f(x, y), and set iterate to list(x = c(1, 2), y = c("a", "b")). The individual function calls are evaluated as f(x = iterate$x[[1]], y = iterate$y[[1]]) and f(x = iterate$x[[2]], y = iterate$y[[2]]). All the elements of iterate must have the same length. If there are any name conflicts between iterate and data, iterate takes precedence.

data Named list of constant local data objects in the evaluation environment. Objects in this list are treated as single values and are held constant for each iteration of the map.

`globals` Named list of constant objects to temporarily assign to the global environment for each task. This list should include any functions you previously defined in the global environment which are required to run tasks. See the `reset_globals` argument of `crew_controller_local()`. Objects in this list are treated as single values and are held constant for each iteration of the map.

`substitute` Logical of length 1, whether to call `base::substitute()` on the supplied value of the `command` argument. If `TRUE` (default) then `command` is quoted literally as you write it, e.g. `push(command = your_function_call())`. If `FALSE`, then `crew` assumes `command` is a language object and you are passing its value, e.g. `push(command = quote(your_function_call()))`. `substitute = TRUE` is appropriate for interactive use, whereas `substitute = FALSE` is meant for automated R programs that invoke `crew` controllers.

`seed` Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the `seed` argument of `set.seed()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`algorithm` Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the `kind` argument of `RNGkind()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`packages` Character vector of packages to load for the task.

`library` Library path to load the packages. See the `lib.loc` argument of `require()`.

`seconds_timeout` Optional task timeout passed to the `.timeout` argument of `mirai::mirai()` (after converting to milliseconds).

`names` Optional character of length 1, name of the element of `iterate` with names for the tasks. If `names` is supplied, then `iterate[[names]]` must be a character vector.

`save_command` Logical of length 1, whether to store a text string version of the R command in the output.

`scale` Logical, whether to automatically scale workers to meet demand. See also the `throttle` argument.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controller` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: In contrast to `walk()`, `map()` blocks the local R session and waits for all tasks to complete.

Returns: Invisibly returns a list of `mirai` task objects for the newly created tasks. The order of tasks in the list matches the order of data in the `iterate` argument.

Method `map()`**:** Apply a single command to multiple inputs, wait for all tasks to complete, and return the results of all tasks.

Usage:

```
crew_class_controller$map(
  command,
```

```
  iterate,
  data = list(),
  globals = list(),
  substitute = TRUE,
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  seconds_interval = 0.5,
  seconds_timeout = NULL,
  names = NULL,
  save_command = FALSE,
  error = "stop",
  warnings = TRUE,
  verbose = interactive(),
  scale = TRUE,
  throttle = TRUE,
  controller = NULL
)
```

Arguments:

`command` Language object with R code to run.

`iterate` Named list of vectors or lists to iterate over. For example, to run function calls `f(x = 1, y = "a")` and `f(x = 2, y = "b")`, set `command` to `f(x, y)`, and set `iterate` to `list(x = c(1, 2), y = c("a", "b"))`. The individual function calls are evaluated as `f(x = iterate$x[[1]], y = iterate$y[[1]])` and `f(x = iterate$x[[2]], y = iterate$y[[2]])`. All the elements of `iterate` must have the same length. If there are any name conflicts between `iterate` and `data`, `iterate` takes precedence.

`data` Named list of constant local data objects in the evaluation environment. Objects in this list are treated as single values and are held constant for each iteration of the map.

`globals` Named list of constant objects to temporarily assign to the global environment for each task. This list should include any functions you previously defined in the global environment which are required to run tasks. See the `reset_globals` argument of `crew_controller_local()`. Objects in this list are treated as single values and are held constant for each iteration of the map.

`substitute` Logical of length 1, whether to call `base::substitute()` on the supplied value of the `command` argument. If `TRUE` (default) then `command` is quoted literally as you write it, e.g. `push(command = your_function_call())`. If `FALSE`, then `crew` assumes `command` is a language object and you are passing its value, e.g. `push(command = quote(your_function_call()))`. `substitute = TRUE` is appropriate for interactive use, whereas `substitute = FALSE` is meant for automated R programs that invoke `crew` controllers.

`seed` Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the `seed` argument of `set.seed()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`algorithm` Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the `kind` argument of `RNGkind()` if not `NULL`. If

algorithm and seed are both NULL, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by mirai::nextstream(). See vignette("parallel", package = "parallel") for details.

packages Character vector of packages to load for the task.

library Library path to load the packages. See the lib.loc argument of require().

seconds_interval Number of seconds to wait between auto-scaling operations while waiting for tasks to complete.

seconds_timeout Optional task timeout passed to the .timeout argument of mirai::mirai() (after converting to milliseconds).

names Optional character of length 1, name of the element of iterate with names for the tasks. If names is supplied, then iterate[[names]] must be a character vector.

save_command Logical of length 1, whether to store a text string version of the R command in the output.

error Character of length 1, choice of action if a task has an error. Possible values:

- "stop": throw an error in the main R session instead of returning a value. In case of an error, the results from the last errored map() are in the error field of the controller, e.g. controller_object$error. To reduce memory consumption, set controller_object$error <- NULL after you are finished troubleshooting.
- "warn": throw a warning. This allows the return value with all the error messages and tracebacks to be generated.
- "silent": do nothing special.

warnings Logical of length 1, whether to throw a warning in the interactive session if at least one task encounters an error.

verbose Logical of length 1, whether to print progress messages.

scale Logical, whether to automatically scale workers to meet demand. See also the throttle argument.

throttle TRUE to skip auto-scaling if it already happened within the last seconds_interval seconds. FALSE to auto-scale every time scale() is called. Throttling avoids overburdening the mirai dispatcher and other resources.

controller Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: map() cannot be used unless all prior tasks are completed and popped. You may need to wait and then pop them manually. Alternatively, you can start over: either call terminate() on the current controller object to reset it, or create a new controller object entirely.

Returns: A tibble of results and metadata: one row per task and columns corresponding to the output of pop().

Method `pop()`**:** Pop a completed task from the results data frame.

Usage:

```
crew_class_controller$pop(
  scale = TRUE,
  collect = NULL,
  throttle = TRUE,
  error = "silent",
  controllers = NULL
)
```

Arguments:

`scale` Logical of length 1, whether to automatically call `scale()` to auto-scale workers to meet the demand of the task load. Scaling up on `pop()` may be important for transient or nearly transient workers that tend to drop off quickly after doing little work. See also the `throttle` argument.

`collect` Deprecated in version 0.5.0.9003 (2023-10-02).

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`error` Character of length 1, choice of action if the popped task threw an error. Possible values:

- `"stop"`: throw an error in the main R session instead of returning a value.
- `"warn"`: throw a warning.
- `"silent"`: do nothing special.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: If not task is currently completed, `pop()` will attempt to auto-scale workers as needed.

Returns: If there is no task to collect, return `NULL`. Otherwise, return a one-row `tibble` with the following columns.

- `name`: the task name if given.
- `command`: a character string with the R command if `save_command` was set to `TRUE` in `push()`.
- `result`: a list containing the return value of the R command.
- `seconds`: number of seconds that the task ran.
- `seed`: the single integer originally supplied to `push()`, `NA` otherwise. The pseudo-random number generator state just prior to the task can be restored using `set.seed(seed = seed, kind = algorithm)`, where `seed` and `algorithm` are part of this output.
- `algorithm`: name of the pseudo-random number generator algorithm originally supplied to `push()`, `NA` otherwise. The pseudo-random number generator state just prior to the task can be restored using `set.seed(seed = seed, kind = algorithm)`, where `seed` and `algorithm` are part of this output.
- `error`: the first 2048 characters of the error message if the task threw an error, `NA` otherwise.
- `trace`: the first 2048 characters of the text of the traceback if the task threw an error, `NA` otherwise.
- `warnings`: the first 2048 characters. of the text of warning messages that the task may have generated, `NA` otherwise.
- `launcher`: name of the `crew` launcher where the task ran.

Method `collect()`**:** Pop all available task results and return them in a tidy `tibble`.

Usage:

```
crew_class_controller$collect(
  scale = TRUE,
  throttle = TRUE,
  controllers = NULL
)
```

Arguments:

`scale` Logical of length 1, whether to automatically call `scale()` to auto-scale workers to meet the demand of the task load.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: A `tibble` of results and metadata of all resolved tasks, with one row per task.

Method `promise()`**:** Create a `promises::promise()` object to asynchronously pop or collect one or more tasks.

Usage:

```
crew_class_controller$promise(
  mode = "one",
  seconds_interval = 0.1,
  scale = TRUE,
  throttle = TRUE,
  controllers = NULL
)
```

Arguments:

`mode` Character of length 1, what kind of promise to create. `mode` must be `"one"` or `"all"`. Details:

- If `mode` is `"one"`, then the promise is fulfilled (or rejected) when at least one task is resolved and available to `pop()`. When that happens, `pop()` runs asynchronously, pops a result off the task list, and returns a value. If the task succeeded, then the promise is fulfilled and its value is the result of `pop()` (a one-row `tibble` with the result and metadata). If the task threw an error, the error message of the task is forwarded to any error callbacks registered with the promise.
- If `mode` is `"all"`, then the promise is fulfilled (or rejected) when there are no unresolved tasks left in the controller. (Be careful: this condition is trivially met in the moment if the controller is empty and you have not submitted any tasks, so it is best to create this kind of promise only after you submit tasks.) When there are no unresolved tasks left, `collect()` runs asynchronously, pops all available results off the task list, and returns a value. If the task succeeded, then the promise is fulfilled and its value is the result of `collect()` (a `tibble` with one row per task result). If any of the tasks threw an error, then the first error message detected is forwarded to any error callbacks registered with the promise.

`seconds_interval` Positive numeric of length 1, delay in the `later::later()` polling interval to asynchronously check if the promise can be resolved.

`scale` Logical of length 1, whether to automatically call `scale()` to auto-scale workers to meet the demand of the task load. Scaling up on `pop()` may be important for transient or nearly transient workers that tend to drop off quickly after doing little work. See also the `throttle` argument.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: Please be aware that `pop()` or `collect()` will happen asynchronously at a some unpredictable time after the promise object is created, even if your local R process appears to be doing something completely different. This behavior is highly desirable in a Shiny reactive context, but please be careful as it may be surprising in other situations.

Returns: A `promises::promise()` object whose eventual value will be a `tibble` with results from one or more popped tasks. If `mode = "one"`, only one task is popped and returned (one row). If `mode = "all"`, then all the tasks are returned in a `tibble` with one row per task (or `NULL` is returned if there are no tasks to pop).

Method `wait()`**:** Wait for tasks.

Usage:

```
crew_class_controller$wait(
  mode = "all",
  seconds_interval = 0.5,
  seconds_timeout = Inf,
  scale = TRUE,
  throttle = TRUE,
  controllers = NULL
)
```

Arguments:

`mode` Character of length 1: `"all"` to wait for all tasks to complete, `"one"` to wait for a single task to complete.

`seconds_interval` Number of seconds to interrupt the wait in order to scale up workers as needed.

`seconds_timeout` Timeout length in seconds waiting for tasks.

`scale` Logical, whether to automatically call `scale()` to auto-scale workers to meet the demand of the task load. See also the `throttle` argument.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: The `wait()` method blocks the calling R session and repeatedly auto-scales workers for tasks that need them. The function runs until it either times out or the condition in `mode` is met.

Returns: A logical of length 1, invisibly. `TRUE` if the condition in `mode` was met, `FALSE` otherwise.

Method `push_backlog()`**:** Push the name of a task to the backlog.

Usage:

```
crew_class_controller$push_backlog(name, controller = NULL)
```

Arguments:

`name` Character of length 1 with the task name to push to the backlog.

`controller` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: `pop_backlog()` pops the tasks that can be pushed without saturating the controller.

Returns: `NULL` (invisibly).

Method `pop_backlog()`**:** Pop the task names from the head of the backlog which can be pushed without saturating the controller.

Usage:

```
crew_class_controller$pop_backlog(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: Character vector of task names which can be pushed to the controller without saturating it. If the controller is saturated, `character(0L)` is returned.

Method `summary()`**:** Summarize the workers and tasks of the controller.

Usage:

```
crew_class_controller$summary(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: A data frame of summary statistics on the workers and tasks. It has one row per worker websocket and the following columns:

- `controller`: name of the controller. . * `worker`: integer index of the worker.
- `tasks`: number of tasks which were completed by a worker at the websocket and then returned by calling `pop()` on the controller object.
- `seconds`: total number of runtime and seconds of all the tasks that ran on a worker connected to this websocket and then were retrieved by calling `pop()` on the controller object.
- `errors`: total number of tasks which ran on a worker at the website, encountered an error in R, and then retrieved with `pop()`.
- `warnings`: total number of tasks which ran on a worker at the website, encountered one or more warnings in R, and then retrieved with `pop()`. Note: `warnings` is actually the number of *tasks*, not the number of warnings. (A task could throw more than one warning.

Method `terminate()`**:** Terminate the workers and the `mirai` client.

Usage:

```
crew_class_controller$terminate(controllers = NULL)
```

Arguments:

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Returns: `NULL` (invisibly).

See Also

Other controller: `crew_controller()`

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
launcher <- crew_launcher_local()
controller <- crew_controller(client = client, launcher = launcher)
controller$start()
controller$push(name = "task", command = sqrt(4))
controller$wait()
controller$pop()
controller$terminate()
}

## ------------------------------------------------
## Method `crew_class_controller$new`
## ------------------------------------------------

if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
launcher <- crew_launcher_local()
controller <- crew_controller(client = client, launcher = launcher)
controller$start()
controller$push(name = "task", command = sqrt(4))
controller$wait()
controller$pop()
controller$terminate()
}
```

---

crew_class_controller_group

Controller group class

Description

R6 class for controller groups.

Details

See crew_controller_group().

Active bindings

`controllers` List of R6 controller objects.

`relay` Relay object for event-driven programming on a downstream condition variable.

Methods

Public methods:

- crew_class_controller_group$new()

- crew_class_controller_group$validate()
- crew_class_controller_group$empty()
- crew_class_controller_group$nonempty()
- crew_class_controller_group$resolved()
- crew_class_controller_group$unresolved()
- crew_class_controller_group$unpopped()
- crew_class_controller_group$saturated()
- crew_class_controller_group$start()
- crew_class_controller_group$launch()
- crew_class_controller_group$scale()
- crew_class_controller_group$push()
- crew_class_controller_group$walk()
- crew_class_controller_group$map()
- crew_class_controller_group$pop()
- crew_class_controller_group$collect()
- crew_class_controller_group$promise()
- crew_class_controller_group$wait()
- crew_class_controller_group$push_backlog()
- crew_class_controller_group$pop_backlog()
- crew_class_controller_group$summary()
- crew_class_controller_group$terminate()

Method `new()`**:** Multi-controller constructor.

Usage:

```
crew_class_controller_group$new(controllers = NULL, relay = NULL)
```

Arguments:

`controllers` List of `R6` controller objects.

`relay` Relay object for event-driven programming on a downstream condition variable.

Returns: An `R6` object with the controller group object.

Examples:

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
persistent <- crew_controller_local(name = "persistent")
transient <- crew_controller_local(
  name = "transient",
  tasks_max = 1L
)
group <- crew_controller_group(persistent, transient)
group$start()
group$push(name = "task", command = sqrt(4), controller = "transient")
group$wait()
group$pop()
group$terminate()
}
```

Method `validate()`**:** Validate the client.

Usage:

```
crew_class_controller_group$validate()
```

Returns: `NULL` (invisibly).

Method `empty()`**:** See if the controllers are empty.

Usage:

```
crew_class_controller_group$empty(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Details: A controller is empty if it has no running tasks or completed tasks waiting to be retrieved with `push()`.

Returns: `TRUE` if all the selected controllers are empty, `FALSE` otherwise.

Method `nonempty()`**:** Check if the controller group is nonempty.

Usage:

```
crew_class_controller_group$nonempty(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Details: A controller is empty if it has no running tasks or completed tasks waiting to be retrieved with `push()`.

Returns: `TRUE` if the controller is empty, `FALSE` otherwise.

Method `resolved()`**:** Number of resolved `mirai()` tasks.

Usage:

```
crew_class_controller_group$resolved(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Details: `resolved()` is cumulative: it counts all the resolved tasks over the entire lifetime of the controller session.

Returns: Non-negative integer of length 1, number of resolved `mirai()` tasks. The return value is 0 if the condition variable does not exist (i.e. if the client is not running).

Method `unresolved()`**:** Number of unresolved `mirai()` tasks.

Usage:

```
crew_class_controller_group$unresolved(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: Non-negative integer of length 1, number of unresolved `mirai()` tasks.

Method `unpopped()`**:** Number of resolved `mirai()` tasks available via `pop()`.

Usage:

```
crew_class_controller_group$unpopped(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: Non-negative integer of length 1, number of resolved `mirai()` tasks available via `pop()`.

Method `saturated()`**:** Check if a controller is saturated.

Usage:

```
crew_class_controller_group$saturated(
  collect = NULL,
  throttle = NULL,
  controller = NULL
)
```

Arguments:

`collect` Deprecated in version 0.5.0.9003 (2023-10-02). Not used.

`throttle` Deprecated in version 0.5.0.9003 (2023-10-02). Not used.

`controller` Character vector of length 1 with the controller name. Set to `NULL` to select the default controller that `push()` would choose.

Details: A controller is saturated if the number of unresolved tasks is greater than or equal to the maximum number of workers. In other words, in a saturated controller, every available worker has a task. You can still push tasks to a saturated controller, but tools that use `crew` such as `targets` may choose not to.

Returns: `TRUE` if all the selected controllers are saturated, `FALSE` otherwise.

Method `start()`**:** Start one or more controllers.

Usage:

```
crew_class_controller_group$start(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: `NULL` (invisibly).

Method `launch()`**:** Launch one or more workers on one or more controllers.

Usage:

```
crew_class_controller_group$launch(n = 1L, controllers = NULL)
```

Arguments:

`n` Number of workers to launch in each controller selected.

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: `NULL` (invisibly).

Method `scale()`**:** Automatically scale up the number of workers if needed in one or more controller objects.

Usage:

```
crew_class_controller_group$scale(throttle = TRUE, controllers = NULL)
```

Arguments:

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Details: See the `scale()` method in individual controller classes.

Returns: `NULL` (invisibly).

Method `push()`**:** Push a task to the head of the task list.

Usage:

```
crew_class_controller_group$push(
  command,
  data = list(),
  globals = list(),
  substitute = TRUE,
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  seconds_timeout = NULL,
  scale = TRUE,
  throttle = TRUE,
  name = NA_character_,
  save_command = FALSE,
  controller = NULL
)
```

Arguments:

`command` Language object with R code to run.

`data` Named list of local data objects in the evaluation environment.

`globals` Named list of objects to temporarily assign to the global environment for the task. See the `reset_globals` argument of `crew_controller_local()`.

`substitute` Logical of length 1, whether to call `base::substitute()` on the supplied value of the `command` argument. If `TRUE` (default) then `command` is quoted literally as you write it, e.g. `push(command = your_function_call())`. If `FALSE`, then `crew` assumes `command` is a language object and you are passing its value, e.g. `push(command = quote(your_function_call()))`. `substitute = TRUE` is appropriate for interactive use, whereas `substitute = FALSE` is meant for automated R programs that invoke `crew` controllers.

`seed` Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the `seed` argument of `set.seed()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`algorithm` Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the `kind` argument of `RNGkind()` if not `NULL`. If

algorithm and seed are both NULL, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by mirai::nextstream(). See vignette("parallel", package = "parallel") for details.

packages Character vector of packages to load for the task.

library Library path to load the packages. See the lib.loc argument of require().

seconds_timeout Optional task timeout passed to the .timeout argument of mirai::mirai() (after converting to milliseconds).

scale Logical, whether to automatically scale workers to meet demand. See the scale argument of the push() method of ordinary single controllers.

throttle TRUE to skip auto-scaling if it already happened within the last seconds_interval seconds. FALSE to auto-scale every time scale() is called. Throttling avoids overburdening the mirai dispatcher and other resources.

name Optional name of the task. Replaced with a random name if NULL or in conflict with an existing name in the task list.

save_command Logical of length 1. If TRUE, the controller deparses the command and returns it with the output on pop(). If FALSE (default), the controller skips this step to increase speed.

controller Character of length 1, name of the controller to submit the task. If NULL, the controller defaults to the first controller in the list.

Returns: Invisibly return the mirai object of the pushed task. This allows you to interact with the task directly, e.g. to create a promise object with promises::as.promise().

Method `walk()`**:** Apply a single command to multiple inputs, and return control to the user without waiting for any task to complete.

Usage:

```
crew_class_controller_group$walk(
  command,
  iterate,
  data = list(),
  globals = list(),
  substitute = TRUE,
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  seconds_timeout = NULL,
  names = NULL,
  save_command = FALSE,
  scale = TRUE,
  throttle = TRUE,
  controller = NULL
)
```

Arguments:

command Language object with R code to run.

iterate Named list of vectors or lists to iterate over. For example, to run function calls f(x = 1, y = "a") and f(x = 2, y = "b"), set command to f(x, y), and set iterate to list(x = c(1, 2), y = c("a", "b")). The individual function calls are evaluated as f(x =

iterate$x[[1]], y = iterate$y[[1]]) and f(x = iterate$x[[2]], y = iterate$y[[2]]). All the elements of iterate must have the same length. If there are any name conflicts between iterate and data, iterate takes precedence.

data Named list of constant local data objects in the evaluation environment. Objects in this list are treated as single values and are held constant for each iteration of the map.

globals Named list of constant objects to temporarily assign to the global environment for each task. This list should include any functions you previously defined in the global environment which are required to run tasks. See the reset_globals argument of crew_controller_local(). Objects in this list are treated as single values and are held constant for each iteration of the map.

substitute Logical of length 1, whether to call base::substitute() on the supplied value of the command argument. If TRUE (default) then command is quoted literally as you write it, e.g. push(command = your_function_call()). If FALSE, then crew assumes command is a language object and you are passing its value, e.g. push(command = quote(your_function_call())). substitute = TRUE is appropriate for interactive use, whereas substitute = FALSE is meant for automated R programs that invoke crew controllers.

seed Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the seed argument of set.seed() if not NULL. If algorithm and seed are both NULL, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by mirai::nextstream(). See vignette("parallel", package = "parallel") for details.

algorithm Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the kind argument of RNGkind() if not NULL. If algorithm and seed are both NULL, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by mirai::nextstream(). See vignette("parallel", package = "parallel") for details.

packages Character vector of packages to load for the task.

library Library path to load the packages. See the lib.loc argument of require().

seconds_timeout Optional task timeout passed to the .timeout argument of mirai::mirai() (after converting to milliseconds).

names Optional character of length 1, name of the element of iterate with names for the tasks. If names is supplied, then iterate[[names]] must be a character vector.

save_command Logical of length 1, whether to store a text string version of the R command in the output.

scale Logical, whether to automatically scale workers to meet demand. See also the throttle argument.

throttle TRUE to skip auto-scaling if it already happened within the last seconds_interval seconds. FALSE to auto-scale every time scale() is called. Throttling avoids overburdening the mirai dispatcher and other resources.

controller Character of length 1, name of the controller to submit the tasks. If NULL, the controller defaults to the first controller in the list.

Details: In contrast to walk(), map() blocks the local R session and waits for all tasks to complete.

Returns: Invisibly returns a list of mirai task objects for the newly created tasks. The order of tasks in the list matches the order of data in the iterate argument.

Method map(): Apply a single command to multiple inputs.

Usage:

```
crew_class_controller_group$map(
  command,
  iterate,
  data = list(),
  globals = list(),
  substitute = TRUE,
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  seconds_interval = 0.5,
  seconds_timeout = NULL,
  names = NULL,
  save_command = FALSE,
  error = "stop",
  warnings = TRUE,
  verbose = interactive(),
  scale = TRUE,
  throttle = TRUE,
  controller = NULL
)
```

Arguments:

`command` Language object with R code to run.

`iterate` Named list of vectors or lists to iterate over. For example, to run function calls `f(x = 1, y = "a")` and `f(x = 2, y = "b")`, set `command` to `f(x, y)`, and set `iterate` to `list(x = c(1, 2), y = c("a", "b"))`. The individual function calls are evaluated as `f(x = iterate$x[[1]], y = iterate$y[[1]])` and `f(x = iterate$x[[2]], y = iterate$y[[2]])`. All the elements of `iterate` must have the same length. If there are any name conflicts between `iterate` and `data`, `iterate` takes precedence.

`data` Named list of constant local data objects in the evaluation environment. Objects in this list are treated as single values and are held constant for each iteration of the map.

`globals` Named list of constant objects to temporarily assign to the global environment for each task. This list should include any functions you previously defined in the global environment which are required to run tasks. See the `reset_globals` argument of `crew_controller_local()`. Objects in this list are treated as single values and are held constant for each iteration of the map.

`substitute` Logical of length 1, whether to call `base::substitute()` on the supplied value of the `command` argument. If `TRUE` (default) then `command` is quoted literally as you write it, e.g. `push(command = your_function_call())`. If `FALSE`, then `crew` assumes `command` is a language object and you are passing its value, e.g. `push(command = quote(your_function_call()))`. `substitute = TRUE` is appropriate for interactive use, whereas `substitute = FALSE` is meant for automated R programs that invoke `crew` controllers.

`seed` Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the `seed` argument of `set.seed()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`algorithm` Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the `kind` argument of `RNGkind()` if not `NULL`. If `algorithm` and `seed` are both `NULL`, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by `mirai::nextstream()`. See `vignette("parallel", package = "parallel")` for details.

`packages` Character vector of packages to load for the task.

`library` Library path to load the packages. See the `lib.loc` argument of `require()`.

`seconds_interval` Number of seconds to wait between auto-scaling operations while waiting for tasks to complete.

`seconds_timeout` Optional task timeout passed to the `.timeout` argument of `mirai::mirai()` (after converting to milliseconds).

`names` Optional character of length 1, name of the element of `iterate` with names for the tasks. If `names` is supplied, then `iterate[[names]]` must be a character vector.

`save_command` Logical of length 1, whether to store a text string version of the R command in the output.

`error` Character vector of length 1, choice of action if a task has an error. Possible values:

- `"stop"`: throw an error in the main R session instead of returning a value. In case of an error, the results from the last errored `map()` are in the `error` field of the controller, e.g. `controller_object$error`. To reduce memory consumption, set `controller_object$error <- NULL` after you are finished troubleshooting.
- `"warn"`: throw a warning. This allows the return value with all the error messages and tracebacks to be generated.
- `"silent"`: do nothing special.

`warnings` Logical of length 1, whether to throw a warning in the interactive session if at least one task encounters an error.

`verbose` Logical of length 1, whether to print progress messages.

`scale` Logical, whether to automatically scale workers to meet demand. See also the `throttle` argument.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controller` Character of length 1, name of the controller to submit the tasks. If `NULL`, the controller defaults to the first controller in the list.

Details: The idea comes from functional programming: for example, the `map()` function from the `purrr` package.

Returns: A `tibble` of results and metadata: one row per task and columns corresponding to the output of `pop()`.

Method `pop()`**:** Pop a completed task from the results data frame.

Usage:

```
crew_class_controller_group$pop(
  scale = TRUE,
  collect = NULL,
  throttle = TRUE,
  error = "silent",
```

```
  controllers = NULL
)
```

Arguments:

`scale` Logical, whether to automatically scale workers to meet demand. See the `scale` argument of the `pop()` method of ordinary single controllers.

`collect` Deprecated in version 0.5.0.9003 (2023-10-02). Not used.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`error` Character of length 1, choice of action if the popped task threw an error. Possible values:

- `"stop"`: throw an error in the main R session instead of returning a value.
- `"warn"`: throw a warning.
- `"silent"`: do nothing special.

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: If there is no task to collect, return `NULL`. Otherwise, return a one-row `tibble` with the same columns as `pop()` for ordinary controllers.

Method `collect()`**:** Pop all available task results and return them in a tidy `tibble`.

Usage:

```
crew_class_controller_group$collect(
  scale = TRUE,
  throttle = TRUE,
  controllers = NULL
)
```

Arguments:

`scale` Logical of length 1, whether to automatically call `scale()` to auto-scale workers to meet the demand of the task load.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: A `tibble` of results and metadata of all resolved tasks, with one row per task. Returns `NULL` if there are no available results.

Method `promise()`**:** Create a `promises::promise()` object to asynchronously pop or collect one or more tasks.

Usage:

```
crew_class_controller_group$promise(
  mode = "one",
  seconds_interval = 0.1,
  scale = TRUE,
  throttle = TRUE,
  controllers = NULL
)
```

Arguments:

`mode` Character of length 1, what kind of promise to create. `mode` must be `"one"` or `"all"`. Details:

- If `mode` is `"one"`, then the promise is fulfilled (or rejected) when at least one task is resolved and available to `pop()`. When that happens, `pop()` runs asynchronously, pops a result off the task list, and returns a value. If the task succeeded, then the promise is fulfilled and its value is the result of `pop()` (a one-row `tibble` with the result and metadata). If the task threw an error, the error message of the task is forwarded to any error callbacks registered with the promise.
- If `mode` is `"all"`, then the promise is fulfilled (or rejected) when there are no unresolved tasks left in the controller. (Be careful: this condition is trivially met in the moment if the controller is empty and you have not submitted any tasks, so it is best to create this kind of promise only after you submit tasks.) When there are no unresolved tasks left, `collect()` runs asynchronously, pops all available results off the task list, and returns a value. If the task succeeded, then the promise is fulfilled and its value is the result of `collect()` (a `tibble` with one row per task result). If any of the tasks threw an error, then the first error message detected is forwarded to any error callbacks registered with the promise.

`seconds_interval` Positive numeric of length 1, delay in the `later::later()` polling interval to asynchronously check if the promise can be resolved.

`scale` Logical of length 1, whether to automatically call `scale()` to auto-scale workers to meet the demand of the task load. Scaling up on `pop()` may be important for transient or nearly transient workers that tend to drop off quickly after doing little work. See also the `throttle` argument.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Not used. Included to ensure the signature is compatible with the analogous method of controller groups.

Details: Please be aware that `pop()` or `collect()` will happen asynchronously at a some unpredictable time after the promise object is created, even if your local R process appears to be doing something completely different. This behavior is highly desirable in a Shiny reactive context, but please be careful as it may be surprising in other situations.

Returns: A `promises::promise()` object whose eventual value will be a `tibble` with results from one or more popped tasks. If `mode = "one"`, only one task is popped and returned (one row). If `mode = "all"`, then all the tasks are returned in a `tibble` with one row per task (or `NULL` is returned if there are no tasks to pop).

Method `wait()`**:** Wait for tasks.

Usage:

```
crew_class_controller_group$wait(
  mode = "all",
  seconds_interval = 0.5,
  seconds_timeout = Inf,
  scale = TRUE,
  throttle = TRUE,
```

```
  controllers = NULL
)
```

Arguments:

`mode` Character of length 1: `"all"` to wait for all tasks in all controllers to complete, `"one"` to wait for a single task in a single controller to complete. In this scheme, the timeout limit is applied to each controller sequentially, and a timeout is treated the same as a completed controller.

`seconds_interval` Number of seconds to interrupt the wait in order to scale up workers as needed.

`seconds_timeout` Timeout length in seconds waiting for results to become available.

`scale` Logical of length 1, whether to call `scale_later()` on each selected controller to schedule auto-scaling. See the `scale` argument of the `wait()` method of ordinary single controllers.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Details: The `wait()` method blocks the calling R session and repeatedly auto-scales workers for tasks that need them. The function runs until it either times out or the condition in `mode` is met.

Returns: A logical of length 1, invisibly. `TRUE` if the condition in `mode` was met, `FALSE` otherwise.

Method `push_backlog()`**:** Push the name of a task to the backlog.

Usage:

```
crew_class_controller_group$push_backlog(name, controller = NULL)
```

Arguments:

`name` Character of length 1 with the task name to push to the backlog.

`controller` Character vector of length 1 with the controller name. Set to `NULL` to select the default controller that `push_backlog()` would choose.

Details: `pop_backlog()` pops the tasks that can be pushed without saturating the controller.

Returns: `NULL` (invisibly).

Method `pop_backlog()`**:** Pop the task names from the head of the backlog which can be pushed without saturating the controller.

Usage:

```
crew_class_controller_group$pop_backlog(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: Character vector of task names which can be pushed to the controller without saturating it. If the controller is saturated, `character(0L)` is returned.

Method `summary()`**:** Summarize the workers of one or more controllers.

Usage:

```
crew_class_controller_group$summary(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: A data frame of aggregated worker summary statistics of all the selected controllers. It has one row per worker, and the rows are grouped by controller. See the documentation of the `summary()` method of the controller class for specific information about the columns in the output.

Method `terminate()`**:** Terminate the workers and disconnect the client for one or more controllers.

Usage:

```
crew_class_controller_group$terminate(controllers = NULL)
```

Arguments:

`controllers` Character vector of controller names. Set to `NULL` to select all controllers.

Returns: `NULL` (invisibly).

See Also

Other controller_group: `crew_controller_group()`

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
persistent <- crew_controller_local(name = "persistent")
transient <- crew_controller_local(
  name = "transient",
  tasks_max = 1L
)
group <- crew_controller_group(persistent, transient)
group$start()
group$push(name = "task", command = sqrt(4), controller = "transient")
group$wait()
group$pop()
group$terminate()
}

## ------------------------------------------------
## Method `crew_class_controller_group$new`
## ------------------------------------------------

if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
persistent <- crew_controller_local(name = "persistent")
transient <- crew_controller_local(
  name = "transient",
  tasks_max = 1L
)
group <- crew_controller_group(persistent, transient)
group$start()
```

```
group$push(name = "task", command = sqrt(4), controller = "transient")
group$wait()
group$pop()
group$terminate()
}
```

`crew_class_launcher` *Launcher abstract class*

Description

R6 abstract class to build other subclasses which launch and manage workers.

Active bindings

`workers` Data frame of worker information.

`name` Name of the launcher.

`seconds_interval` See `crew_launcher()`.

`seconds_timeout` See `crew_launcher()`.

`seconds_launch` See `crew_launcher()`.

`seconds_idle` See `crew_launcher()`.

`seconds_wall` See `crew_launcher()`.

`tasks_max` See `crew_launcher()`.

`tasks_timers` See `crew_launcher()`.

`reset_globals` See `crew_launcher()`.

`reset_packages` See `crew_launcher()`.

`reset_options` See `crew_launcher()`.

`garbage_collection` See `crew_launcher()`.

`launch_max` See `crew_launcher()`.

`tls` See `crew_launcher()`.

`processes` See `crew_launcher()`. asynchronously.

`async` A `crew_async()` object to run low-level launcher tasks asynchronously.

`throttle` A `crew_throttle()` object to throttle scaling.

Methods

Public methods:

- `crew_class_launcher$new()`
- `crew_class_launcher$validate()`
- `crew_class_launcher$set_name()`
- `crew_class_launcher$settings()`

- crew_class_launcher$call()
- crew_class_launcher$start()
- crew_class_launcher$terminate()
- crew_class_launcher$summary()
- crew_class_launcher$tally()
- crew_class_launcher$unlaunched()
- crew_class_launcher$booting()
- crew_class_launcher$active()
- crew_class_launcher$done()
- crew_class_launcher$rotate()
- crew_class_launcher$launch()
- crew_class_launcher$forward()
- crew_class_launcher$errors()
- crew_class_launcher$wait()
- crew_class_launcher$scale()
- crew_class_launcher$launch_worker()
- crew_class_launcher$terminate_worker()
- crew_class_launcher$terminate_workers()

Method `new()`**:** Launcher constructor.

Usage:

```
crew_class_launcher$new(
  name = NULL,
  seconds_interval = NULL,
  seconds_timeout = NULL,
  seconds_launch = NULL,
  seconds_idle = NULL,
  seconds_wall = NULL,
  seconds_exit = NULL,
  tasks_max = NULL,
  tasks_timers = NULL,
  reset_globals = NULL,
  reset_packages = NULL,
  reset_options = NULL,
  garbage_collection = NULL,
  launch_max = NULL,
  tls = NULL,
  processes = NULL
)
```

Arguments:

`name` See crew_launcher().

`seconds_interval` See crew_launcher().

`seconds_timeout` See crew_launcher().

`seconds_launch` See crew_launcher().

`seconds_idle` See crew_launcher().

`seconds_wall` See `crew_launcher()`.

`seconds_exit` See `crew_launcher()`.

`tasks_max` See `crew_launcher()`.

`tasks_timers` See `crew_launcher()`.

`reset_globals` See `crew_launcher()`.

`reset_packages` See `crew_launcher()`.

`reset_options` See `crew_launcher()`.

`garbage_collection` See `crew_launcher()`.

`launch_max` See `crew_launcher()`.

`tls` See `crew_launcher()`.

`processes` See `crew_launcher()`.

Returns: An `R6` object with the launcher.

Examples:

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(workers = client$workers)
launcher$launch(index = 1L)
m <- mirai::mirai("result", .compute = client$name)
Sys.sleep(0.25)
m$data
client$terminate()
}
```

Method `validate()`**:** Validate the launcher.

Usage:

```
crew_class_launcher$validate()
```

Returns: `NULL` (invisibly).

Method `set_name()`**:** Set the name of the launcher.

Usage:

```
crew_class_launcher$set_name(name)
```

Arguments:

`name` Character of length 1, name to set for the launcher.

Returns: `NULL` (invisibly).

Method `settings()`**:** List of arguments for `mirai::daemon()`.

Usage:

```
crew_class_launcher$settings(socket)
```

Arguments:

`socket` Character of length 1, websocket address of the worker to launch.

Returns: List of arguments for `mirai::daemon()`.

Method `call()`**:** Create a call to `crew_worker()` to help create custom launchers.

Usage:

```
crew_class_launcher$call(socket, launcher, worker, instance)
```

Arguments:

`socket` Socket where the worker will receive tasks.

`launcher` Character of length 1, name of the launcher.

`worker` Positive integer of length 1, index of the worker. This worker index remains the same even when the current instance of the worker exits and a new instance launches.

`instance` Character of length 1 to uniquely identify the instance of the worker.

Returns: Character of length 1 with a call to `crew_worker()`.

Examples:

```
launcher <- crew_launcher_local()
launcher$call(
  socket = "ws://127.0.0.1:5000/3/cba033e58",
  launcher = "launcher_a",
  worker = 3L,
  instance = "cba033e58"
)
```

Method `start()`**:** Start the launcher.

Usage:

```
crew_class_launcher$start(sockets = NULL)
```

Arguments:

`sockets` For testing purposes only.

Details: Creates the workers data frame. Meant to be called once at the beginning of the launcher life cycle, after the client has started.

Returns: `NULL` (invisibly).

Method `terminate()`**:** Terminate the whole launcher, including all workers.

Usage:

```
crew_class_launcher$terminate()
```

Returns: `NULL` (invisibly).

Method `summary()`**:** Summarize the workers.

Usage:

```
crew_class_launcher$summary()
```

Returns: `NULL` if the launcher is not started. Otherwise, a `tibble` with one row per `crew` worker and the following columns:

- `worker`: integer index of the worker.
- `launches`: number of times the worker was launched. Each launch occurs at a different websocket because the token at the end of the URL is rotated before each new launch.
- `online`: logical vector, whether the current instance of each worker was actively connected to its NNG socket during the time of the last call to `tally()`.

- `discovered`: logical vector, whether the current instance of each worker had connected to its NNG socket at some point (and then possibly disconnected) during the time of the last call to `tally()`.
- `assigned`: cumulative number of tasks assigned, reported by `mirai::daemons()` and summed over all completed instances of the worker. Does not reflect the activity of the currently running instance of the worker.
- `complete`: cumulative number of tasks completed, reported by `mirai::daemons()` and summed over all completed instances of the worker. Does not reflect the activity of the currently running instance of the worker.
- `socket`: current websocket URL of the worker.

Method `tally()`**:** Update the `daemons`-related columns of the internal `workers` data frame.

Usage:

```
crew_class_launcher$tally(daemons = NULL)
```

Arguments:

`daemons` `mirai` daemons matrix. For testing only. Users should not set this.

Returns: `NULL` (invisibly).

Method `unlaunched()`**:** Get indexes of unlaunched workers.

Usage:

```
crew_class_launcher$unlaunched(n = Inf)
```

Arguments:

`n` Maximum number of worker indexes to return.

Details: A worker is "unlaunched" if it has never connected to the current instance of its websocket. Once a worker launches with the `launch()` method, it is considered "launched" until it disconnects and its websocket is rotated with `rotate()`.

Returns: Integer index of workers available for launch. The backlogged workers are listed first. A worker is backlogged if it is assigned more tasks than it completed.

Method `booting()`**:** Get workers that may still be booting up.

Usage:

```
crew_class_launcher$booting()
```

Details: A worker is "booting" if its launch time is within the last `seconds_launch` seconds. `seconds_launch` is a configurable grace period when `crew` allows a worker to start up and connect to the `mirai` dispatcher. The `booting()` function does not know about the actual worker connection status, it just knows about launch times, so it may return `TRUE` for workers that have already connected and started doing tasks.

Method `active()`**:** Get active workers.

Usage:

```
crew_class_launcher$active()
```

Details: A worker is "active" if its current instance is online and connected, or if it is within its booting time window and has never connected. In other words, "active" means `online | (!discovered & booting)`.

Returns: Logical vector with `TRUE` for active workers and `FALSE` for inactive ones.

Method `done()`**:** Get done workers.

Usage:

```
crew_class_launcher$done()
```

Details: A worker is "done" if it is launched and inactive. A worker is "launched" if `launch()` was called and the worker websocket has not been rotated since.

Returns: Integer index of inactive workers.

Method `rotate()`**:**

Usage:

```
crew_class_launcher$rotate()
```

Details: Rotate websockets at all unlaunched workers.

Returns: `NULL` (invisibly).

Method `launch()`**:** Launch a worker.

Usage:

```
crew_class_launcher$launch(index)
```

Arguments:

`index` Positive integer of length 1, index of the worker to launch.

Returns: `NULL` (invisibly).

Method `forward()`**:** Forward an asynchronous launch/termination error condition of a worker.

Usage:

```
crew_class_launcher$forward(index, condition = "error")
```

Arguments:

`index` Integer of length 1, index of the worker to inspect.

`condition` Character of length 1 indicating what to do with an error if found. `"error"` to throw an error, `"warning"` to throw a warning, `"message"` to print a message, and `"character"` to return a character vector of specific task-level error messages. The return value is `NULL` if no error is found.

Returns: Throw an error, throw a warning, or return a character string, depending on the `condition` argument.

Method `errors()`**:** Collect and return the most recent error messages from asynchronous worker launching and termination.

Usage:

```
crew_class_launcher$errors()
```

Returns: Character vector of all the most recent error messages from asynchronous worker launching and termination. `NULL` if there are no errors.

Method `wait()`**:** Wait for any local asynchronous launch or termination tasks to complete.

Usage:

`crew_class_launcher$wait()`

Details: Only relevant if `processes` is a positive integer.

Returns: `NULL` (invisibly).

Method `scale()`**:** Auto-scale workers out to meet the demand of tasks.

Usage:

```
crew_class_launcher$scale(demand, throttle = TRUE)
```

Arguments:

`demand` Number of unresolved tasks.

`throttle` `TRUE` to skip auto-scaling if it already happened within the last `seconds_interval` seconds. `FALSE` to auto-scale every time `scale()` is called. Throttling avoids overburdening the `mirai` dispatcher and other resources.

Returns: `NULL` (invisibly)

Method `launch_worker()`**:** Abstract worker launch method.

Usage:

```
crew_class_launcher$launch_worker(call, name, launcher, worker, instance)
```

Arguments:

`call` Character of length 1 with a namespaced call to `crew_worker()` which will run in the worker and accept tasks.

`name` Character of length 1 with an informative worker name.

`launcher` Character of length 1, name of the launcher.

`worker` Positive integer of length 1, index of the worker. This worker index remains the same even when the current instance of the worker exits and a new instance launches. It is always between 1 and the maximum number of concurrent workers.

`instance` Character of length 1 to uniquely identify the current instance of the worker a the index in the launcher.

Details: Launcher plugins will overwrite this method.

Returns: A handle to mock the worker launch.

Method `terminate_worker()`**:** Abstract worker termination method.

Usage:

```
crew_class_launcher$terminate_worker(handle)
```

Arguments:

`handle` A handle object previously returned by `launch_worker()` which allows the termination of the worker.

Details: Launcher plugins will overwrite this method.

Returns: A handle to mock worker termination.

Method `terminate_workers()`**:** Terminate one or more workers.

Usage:

```
crew_class_launcher$terminate_workers(index = NULL)
```

Arguments:

`index` Integer vector of the indexes of the workers to terminate. If `NULL`, all current workers are terminated.

Returns: `NULL` (invisibly).

See Also

Other launcher: crew_launcher()

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(workers = client$workers)
launcher$launch(index = 1L)
m <- mirai::mirai("result", .compute = client$name)
Sys.sleep(0.25)
m$data
client$terminate()
}

## ------------------------------------------------
## Method `crew_class_launcher$new`
## ------------------------------------------------

if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(workers = client$workers)
launcher$launch(index = 1L)
m <- mirai::mirai("result", .compute = client$name)
Sys.sleep(0.25)
m$data
client$terminate()
}

## ------------------------------------------------
## Method `crew_class_launcher$call`
## ------------------------------------------------

launcher <- crew_launcher_local()
launcher$call(
  socket = "ws://127.0.0.1:5000/3/cba033e58",
  launcher = "launcher_a",
  worker = 3L,
  instance = "cba033e58"
)
```

---

crew_class_launcher_local

Local process launcher class

Description

R6 class to launch and manage local process workers.

Details

See crew_launcher_local().

Super class

crew::crew_class_launcher -> crew_class_launcher_local

Active bindings

local_log_directory See crew_launcher_local().

local_log_join See crew_launcher_local().

Methods

Public methods:

- crew_class_launcher_local$new()
- crew_class_launcher_local$validate()
- crew_class_launcher_local$launch_worker()
- crew_class_launcher_local$terminate_worker()

Method new(): Local launcher constructor.

Usage:

```
crew_class_launcher_local$new(
  name = NULL,
  seconds_interval = NULL,
  seconds_timeout = NULL,
  seconds_launch = NULL,
  seconds_idle = NULL,
  seconds_wall = NULL,
  seconds_exit = NULL,
  tasks_max = NULL,
  tasks_timers = NULL,
  reset_globals = NULL,
  reset_packages = NULL,
  reset_options = NULL,
  garbage_collection = NULL,
  launch_max = NULL,
  tls = NULL,
  processes = NULL,
  local_log_directory = NULL,
  local_log_join = NULL
)
```

Arguments:

name See crew_launcher().

seconds_interval See crew_launcher().

seconds_timeout See crew_launcher().

seconds_launch See crew_launcher().

seconds_idle See crew_launcher().

seconds_wall See crew_launcher().

seconds_exit See crew_launcher().

tasks_max See crew_launcher().

tasks_timers See crew_launcher().

reset_globals See crew_launcher().

reset_packages See crew_launcher().

reset_options See crew_launcher().

garbage_collection See crew_launcher().

launch_max See crew_launcher().

tls See crew_launcher().

processes See crew_launcher().

local_log_directory See crew_launcher_local().

local_log_join See crew_launcher_local().

Returns: An R6 object with the local launcher.

Examples:

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(sockets = client$summary()$socket)
launcher$launch(index = 1L)
task <- mirai::mirai("result", .compute = client$name)
mirai::call_mirai_(task)
task$data
client$terminate()
}
```

Method `validate()`**:** Validate the local launcher.

Usage:

```
crew_class_launcher_local$validate()
```

Returns: NULL (invisibly).

Method `launch_worker()`**:** Launch a local process worker which will dial into a socket.

Usage:

```
crew_class_launcher_local$launch_worker(call, name, launcher, worker, instance)
```

Arguments:

call Character of length 1 with a namespaced call to crew_worker() which will run in the worker and accept tasks.

`name` Character of length 1 with a long informative worker name which contains the `launcher`, `worker`, and `instance` arguments described below.

`launcher` Character of length 1, name of the launcher.

`worker` Positive integer of length 1, index of the worker. This worker index remains the same even when the current instance of the worker exits and a new instance launches. It is always between 1 and the maximum number of concurrent workers.

`instance` Character of length 1 to uniquely identify the current instance of the worker a the index in the launcher.

Details: The `call` argument is R code that will run to initiate the worker. Together, the `launcher`, `worker`, and `instance` arguments are useful for constructing informative job names.

Returns: A handle object to allow the termination of the worker later on.

Method `terminate_worker()`**:** Terminate a local process worker.

Usage:

```
crew_class_launcher_local$terminate_worker(handle)
```

Arguments:

`handle` A process handle object previously returned by `launch_worker()`.

Returns: A list with the process ID of the worker.

See Also

Other plugin_local: `crew_controller_local()`, `crew_launcher_local()`

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(sockets = client$summary()$socket)
launcher$launch(index = 1L)
task <- mirai::mirai("result", .compute = client$name)
mirai::call_mirai_(task)
task$data
client$terminate()
}

## ------------------------------------------------
## Method `crew_class_launcher_local$new`
## ------------------------------------------------

if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(sockets = client$summary()$socket)
launcher$launch(index = 1L)
task <- mirai::mirai("result", .compute = client$name)
```

```
mirai::call_mirai_(task)
task$data
client$terminate()
}
```

`crew_class_monitor_local` *Local monitor class*

Description

Local monitor `R6` class

Details

See `crew_monitor_local()`.

Methods

Public methods:

- `crew_class_monitor_local$dispatchers()`
- `crew_class_monitor_local$daemons()`
- `crew_class_monitor_local$workers()`
- `crew_class_monitor_local$terminate()`

Method `dispatchers()`**:** List the process IDs of the running `mirai` dispatcher processes.

Usage:

```
crew_class_monitor_local$dispatchers(user = ps::ps_username())
```

Arguments:

`user` Character of length 1, user ID to filter on. `NULL` to list processes of all users (not recommended).

Returns: Integer vector of process IDs of the running `mirai` dispatcher processes.

Method `daemons()`**:** List the process IDs of the locally running `mirai` daemon processes which are not `crew` workers. The `crew_async()` object can launch such processes: for example, when a positive integer is supplied to the `processes` argument of e.g. `crew.aws.batch::crew_controller_aws_batch()`.

Usage:

```
crew_class_monitor_local$daemons(user = ps::ps_username())
```

Arguments:

`user` Character of length 1, user ID to filter on. `NULL` to list processes of all users (not recommended).

Returns: Integer vector of process IDs of the locally running `mirai` daemon processes which are not `crew` workers.

Method `workers()`**:** List the process IDs of locally running `crew` workers launched by the local controller (`crew_controller_local()`).

Usage:

```
crew_class_monitor_local$workers(user = ps::ps_username())
```

Arguments:

`user` Character of length 1, user ID to filter on. `NULL` to list processes of all users (not recommended).

Details: Only the workers running on your local computer are listed. Workers that are not listed include jobs on job schedulers like SLURM or jobs on cloud services like AWS Batch. To monitor those worker processes, please consult the monitor objects in the relevant third-party launcher plugins such as `crew.cluster` and `crew.aws.batch`.

Returns: Integer vector of process IDs of locally running `crew` workers launched by the local controller (`crew_controller_local()`).

Method `terminate()`**:** Terminate the given process IDs.

Usage:

```
crew_class_monitor_local$terminate(pids)
```

Arguments:

`pids` Integer vector of process IDs of local processes to terminate.

Details: Termination happens with the operating system signal given by `crew_terminate_signal()`.

Returns: `NULL` (invisibly).

See Also

Other monitor: `crew_monitor_local()`

`crew_class_relay` `R6` *relay class.*

Description

`R6` class for relay configuration.

Details

See `crew_relay()`.

Active bindings

`condition` Main condition variable.

`from` Condition variable to relay from.

`to` Condition variable to relay to.

Methods

Public methods:

- crew_class_relay$validate()
- crew_class_relay$start()
- crew_class_relay$terminate()
- crew_class_relay$set_from()
- crew_class_relay$set_to()
- crew_class_relay$wait()

Method `validate()`**:** Validate the object.

Usage:

```
crew_class_relay$validate()
```

Returns: `NULL` (invisibly).

Method `start()`**:** Start the relay object.

Usage:

```
crew_class_relay$start()
```

Returns: `NULL` (invisibly).

Method `terminate()`**:** Terminate the relay object.

Usage:

```
crew_class_relay$terminate()
```

Returns: `NULL` (invisibly).

Method `set_from()`**:** Set the condition variable to relay from.

Usage:

```
crew_class_relay$set_from(from)
```

Arguments:

`from` Condition variable to relay from.

Returns: `NULL` (invisibly).

Method `set_to()`**:** Set the condition variable to relay to.

Usage:

```
crew_class_relay$set_to(to)
```

Arguments:

`to` Condition variable to relay to.

Returns: `NULL` (invisibly).

Method `wait()`**:** Wait until an unobserved task resolves or the timeout is reached.

Usage:

```
crew_class_relay$wait(seconds_timeout = 1000)
```

Arguments:

`seconds_timeout` Positive numeric of length 1, Number of seconds to wait before timing out.

Returns: `NULL` (invisibly).

See Also

Other relay: crew_relay()

Examples

```
crew_relay()
```

---

crew_class_throttle R6 *throttle class.*

Description

R6 class for throttle configuration.

Details

See crew_throttle().

Active bindings

seconds_interval Positive numeric of length 1, throttling interval in seconds.

polled Positive numeric of length 1, millisecond timestamp of the last time poll() returned TRUE. NULL if poll() was never called on the current object.

Methods

Public methods:

- crew_class_throttle$new()
- crew_class_throttle$validate()
- crew_class_throttle$poll()
- crew_class_throttle$reset()

Method new(): Throttle constructor.

Usage:

```
crew_class_throttle$new(seconds_interval = NULL)
```

Arguments:

seconds_interval Throttling interval in seconds.

Returns: An R6 object with throttle configuration.

Examples:

```
throttle <- crew_throttle(seconds_interval = 0.5)
throttle$poll()
throttle$poll()
```

Method validate(): Validate the object.

Usage:

```
crew_class_throttle$validate()
```

Returns: NULL (invisibly).

Method `poll()`**:** Poll the throttler.

Usage:

```
crew_class_throttle$poll()
```

Returns: TRUE if `poll()` did not return TRUE in the last `seconds_interval` seconds, FALSE otherwise.

Method `reset()`**:** Reset the throttle object so the next `poll()` returns TRUE.

Usage:

```
crew_class_throttle$reset()
```

Returns: NULL (invisibly).

See Also

Other throttle: `crew_throttle()`

Examples

```
throttle <- crew_throttle(seconds_interval = 0.5)
throttle$poll()
throttle$poll()

## ------------------------------------------------
## Method `crew_class_throttle$new`
## ------------------------------------------------

throttle <- crew_throttle(seconds_interval = 0.5)
throttle$poll()
throttle$poll()
```

`crew_class_tls` R6 *TLS class.*

Description

R6 class for TLS configuration.

Details

See `crew_tls()`.

Active bindings

`mode` See `crew_tls()`.

`key` See `crew_tls()`.

`password` See `crew_tls()`.

`certificates` See `crew_tls()`.

Methods

Public methods:

- `crew_class_tls$new()`
- `crew_class_tls$validate()`
- `crew_class_tls$client()`
- `crew_class_tls$worker()`

Method `new()`**:** TLS configuration constructor.

Usage:

```
crew_class_tls$new(
  mode = NULL,
  key = NULL,
  password = NULL,
  certificates = NULL
)
```

Arguments:

`mode` Argument passed from `crew_tls()`.

`key` Argument passed from `crew_tls()`.

`password` Argument passed from `crew_tls()`.

`certificates` Argument passed from `crew_tls()`.

Returns: An `R6` object with TLS configuration.

Examples:

```
crew_tls(mode = "automatic")
```

Method `validate()`**:** Validate the object.

Usage:

```
crew_class_tls$validate(test = TRUE)
```

Arguments:

`test` Logical of length 1, whether to test the TLS configuration with `nanonext::tls_config()`.

Returns: `NULL` (invisibly).

Method `client()`**:** TLS credentials for the `crew` client.

Usage:

```
crew_class_tls$client()
```

Returns: `NULL` or character vector, depending on the mode.

Method `worker()`**:** TLS credentials for `crew` workers.

Usage:

```
crew_class_tls$worker(name)
```

Arguments:

`name` Character of length 1 with the `mirai` compute profile.

Returns: `NULL` or character vector, depending on the mode.

See Also

Other tls: `crew_tls()`

Examples

```
crew_tls(mode = "automatic")

## ------------------------------------------------
## Method `crew_class_tls$new`
## ------------------------------------------------

crew_tls(mode = "automatic")
```

`crew_clean` *Terminate dispatchers and/or workers*

Description

Terminate `mirai` dispatchers and/or `crew` workers which may be lingering from previous workloads.

Usage

```
crew_clean(
  dispatchers = TRUE,
  workers = TRUE,
  user = ps::ps_username(),
  seconds_interval = 0.5,
  seconds_timeout = 60,
  verbose = TRUE
)
```

Arguments

`dispatchers`	Logical of length 1, whether to terminate dispatchers.
`workers`	Logical of length 1, whether to terminate workers.
`user`	Character of length 1. Terminate dispatchers and/or workers associated with this user name.

seconds_interval
: Seconds to between polling intervals waiting for a process to exit.

seconds_timeout
: Seconds to wait for a process to exit.

verbose
: Logical of length 1, whether to print an informative message every time a process is terminated.

Details

Behind the scenes, mirai uses an external R process called a "dispatcher" to send tasks to crew workers. This dispatcher usually shuts down when you terminate the controller or quit your R session, but sometimes it lingers. Likewise, sometimes crew workers do not shut down on their own. The crew_clean() function searches the process table on your local machine and manually terminates any mirai dispatchers and crew workers associated with your user name (or the user name you select in the user argument. Unfortunately, it cannot reach remote workers such as those launched by a crew.cluster controller.

Value

NULL (invisibly). If verbose is TRUE, it does print out a message for every terminated process.

See Also

Other utility: crew_assert(), crew_deprecate(), crew_eval(), crew_random_name(), crew_retry(), crew_terminate_process(), crew_terminate_signal(), crew_worker()

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
crew_clean()
}
```

---

crew_client *Create a client object.*

Description

Create an R6 wrapper object to manage the mirai client.

Usage

```
crew_client(
  name = NULL,
  workers = 1L,
  host = NULL,
  port = NULL,
  tls = crew::crew_tls(),
  tls_enable = NULL,
```

```
  tls_config = NULL,
  seconds_interval = 0.5,
  seconds_timeout = 5
)
```

Arguments

name	Name of the client object. If NULL, a name is automatically generated.
workers	Integer, maximum number of parallel workers to run.
host	IP address of the mirai client to send and receive tasks. If NULL, the host defaults to the local IP address.
port	TCP port to listen for the workers. If NULL, then an available ephemeral port is automatically chosen.
tls	A TLS configuration object from crew_tls().
tls_enable	Deprecated on 2023-09-15 in version 0.4.1. Use argument tls instead.
tls_config	Deprecated on 2023-09-15 in version 0.4.1. Use argument tls instead.
seconds_interval	Number of seconds between polling intervals waiting for certain internal synchronous operations to complete, such as checking mirai::status()
seconds_timeout	Number of seconds until timing out while waiting for certain synchronous operations to complete, such as checking mirai::status().

See Also

Other client: crew_class_client

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
client$summary()
client$terminate()
}
```

---

crew_controller *Create a controller object from a client and launcher.*

Description

This function is for developers of crew launcher plugins. Users should use a specific controller helper such as crew_controller_local().

Usage

```
crew_controller(client, launcher, auto_scale = NULL)
```

Arguments

client	An R6 client object created by crew_client().
launcher	An R6 launcher object created by one of the crew_launcher_*() functions such as crew_launcher_local().
auto_scale	Deprecated. Use the scale argument of push(), pop(), and wait() instead.

See Also

Other controller: crew_class_controller

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
launcher <- crew_launcher_local()
controller <- crew_controller(client = client, launcher = launcher)
controller$start()
controller$push(name = "task", command = sqrt(4))
controller$wait()
controller$pop()
controller$terminate()
}
```

---

crew_controller_group *Create a controller group.*

Description

Create an R6 object to submit tasks and launch workers through multiple crew controllers.

Usage

```
crew_controller_group(...)
```

Arguments

...	R6 controller objects or lists of R6 controller objects. Nested lists are allowed, but each element must be a control object or another list.

See Also

Other controller_group: crew_class_controller_group

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
persistent <- crew_controller_local(name = "persistent")
transient <- crew_controller_local(
  name = "transient",
  tasks_max = 1L
)
group <- crew_controller_group(persistent, transient)
group$start()
group$push(name = "task", command = sqrt(4), controller = "transient")
group$wait()
group$pop()
group$terminate()
}
```

`crew_controller_local` *Create a controller with a local process launcher.*

Description

Create an R6 object to submit tasks and launch workers on local processes.

Usage

```
crew_controller_local(
  name = NULL,
  workers = 1L,
  host = "127.0.0.1",
  port = NULL,
  tls = crew::crew_tls(),
  tls_enable = NULL,
  tls_config = NULL,
  seconds_interval = 0.5,
  seconds_timeout = 60,
  seconds_launch = 30,
  seconds_idle = Inf,
  seconds_wall = Inf,
  seconds_exit = NULL,
  tasks_max = Inf,
  tasks_timers = 0L,
  reset_globals = TRUE,
  reset_packages = FALSE,
  reset_options = FALSE,
  garbage_collection = FALSE,
  launch_max = 5L,
  local_log_directory = NULL,
  local_log_join = TRUE
)
```

Arguments

name	Name of the client object. If NULL, a name is automatically generated.
workers	Integer, maximum number of parallel workers to run.
host	IP address of the mirai client to send and receive tasks. If NULL, the host defaults to the local IP address.
port	TCP port to listen for the workers. If NULL, then an available ephemeral port is automatically chosen.
tls	A TLS configuration object from crew_tls().
tls_enable	Deprecated on 2023-09-15 in version 0.4.1. Use argument tls instead.
tls_config	Deprecated on 2023-09-15 in version 0.4.1. Use argument tls instead.
seconds_interval	Number of seconds between polling intervals waiting for certain internal synchronous operations to complete, such as checking mirai::status()
seconds_timeout	Number of seconds until timing out while waiting for certain synchronous operations to complete, such as checking mirai::status().
seconds_launch	Seconds of startup time to allow. A worker is unconditionally assumed to be alive from the moment of its launch until seconds_launch seconds later. After seconds_launch seconds, the worker is only considered alive if it is actively connected to its assign websocket.
seconds_idle	Maximum number of seconds that a worker can idle since the completion of the last task. If exceeded, the worker exits. But the timer does not launch until tasks_timers tasks have completed. See the idletime argument of mirai::daemon(). crew does not excel with perfectly transient workers because it does not micromanage the assignment of tasks to workers, so please allow enough idle time for a new worker to be delegated a new task.
seconds_wall	Soft wall time in seconds. The timer does not launch until tasks_timers tasks have completed. See the walltime argument of mirai::daemon().
seconds_exit	Deprecated on 2023-09-21 in version 0.5.0.9002. No longer necessary.
tasks_max	Maximum number of tasks that a worker will do before exiting. See the maxtasks argument of mirai::daemon(). crew does not excel with perfectly transient workers because it does not micromanage the assignment of tasks to workers, it is recommended to set tasks_max to a value greater than 1.
tasks_timers	Number of tasks to do before activating the timers for seconds_idle and seconds_wall. See the timerstart argument of mirai::daemon().
reset_globals	TRUE to reset global environment variables between tasks, FALSE to leave them alone.
reset_packages	TRUE to unload any packages loaded during a task (runs between each task), FALSE to leave packages alone.
reset_options	TRUE to reset global options to their original state between each task, FALSE otherwise. It is recommended to only set reset_options = TRUE if reset_packages is also TRUE because packages sometimes rely on options they set at loading time.

garbage_collection
: TRUE to run garbage collection between tasks, FALSE to skip.

launch_max
: Positive integer of length 1, maximum allowed consecutive launch attempts which do not complete any tasks. Enforced on a worker-by-worker basis. The futile launch count resets to back 0 for each worker that completes a task. It is recommended to set launch_max above 0 because sometimes workers are unproductive under perfectly ordinary circumstances. But launch_max should still be small enough to detect errors in the underlying platform.

local_log_directory
: Either NULL or a character of length 1 with the file path to a directory to write worker-specific log files with standard output and standard error messages. Each log file represents a single *instance* of a running worker, so there will be more log files if a given worker starts and terminates a lot. Set to NULL to suppress log files (default).

local_log_join
: Logical of length 1. If TRUE, crew will write standard output and standard error to the same log file for each worker instance. If FALSE, then they these two streams will go to different log files with informative suffixes.

See Also

Other plugin_local: crew_class_launcher_local, crew_launcher_local()

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
controller <- crew_controller_local()
controller$start()
controller$push(name = "task", command = sqrt(4))
controller$wait()
controller$pop()
controller$terminate()
}
```

---

crew_deprecate *Deprecate a* crew *feature.*

Description

Show an informative warning when a crew feature is deprecated.

Usage

```
crew_deprecate(
  name,
  date,
  version,
  alternative,
```

```
  condition = "warning",
  value = "x",
  skip_cran = FALSE,
  frequency = "always"
)
```

Arguments

name	Name of the feature (function or argument) to deprecate.
date	Date of deprecation.
version	Package version when deprecation was instated.
alternative	Message about an alternative.
condition	Either "warning" or "message" to indicate the type of condition thrown on deprecation.
value	Value of the object. Deprecation is skipped if `value` is `NULL`.
skip_cran	Logical of length 1, whether to skip the deprecation warning or message on CRAN.
frequency	Character of length 1, passed to the `.frequency` argument of `rlang::warn()`.

Value

`NULL` (invisibly). Throws a warning if a feature is deprecated.

See Also

Other utility: `crew_assert()`, `crew_clean()`, `crew_eval()`, `crew_random_name()`, `crew_retry()`, `crew_terminate_process()`, `crew_terminate_signal()`, `crew_worker()`

Examples

```
suppressWarnings(
  crew_deprecate(
    name = "auto_scale",
    date = "2023-05-18",
    version = "0.2.0",
    alternative = "use the scale argument of push(), pop(), and wait()."
  )
)
```

crew_eval *Evaluate an R command and return results as a monad.*

Description

Not a user-side function. Do not call directly.

Usage

```
crew_eval(
  command,
  name = NA_character_,
  string = NA_character_,
  data = list(),
  globals = list(),
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL
)
```

Arguments

command	Language object with R code to run.
name	Character of length 1, name of the task.
string	Character of length 1, string representation of the command.
data	Named list of local data objects in the evaluation environment.
globals	Named list of objects to temporarily assign to the global environment for the task.
seed	Integer of length 1 with the pseudo-random number generator seed to set for the evaluation of the task. Passed to the seed argument of set.seed() if not NULL. If algorithm and seed are both NULL, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by mirai::nextstream(). See vignette("parallel", package = "parallel") for details.
algorithm	Integer of length 1 with the pseudo-random number generator algorithm to set for the evaluation of the task. Passed to the kind argument of RNGkind() if not NULL. If algorithm and seed are both NULL, then the random number generator defaults to the recommended widely spaced worker-specific L'Ecuyer streams as supported by mirai::nextstream(). See vignette("parallel", package = "parallel") for details.
packages	Character vector of packages to load for the task.
library	Library path to load the packages. See the lib.loc argument of require().

Details

The `crew_eval()` function evaluates an R expression in an encapsulated environment and returns a monad with the results, including warnings and error messages if applicable. The random number generator seed, `globals`, and global options are restored to their original values on exit.

Value

A monad object with results and metadata.

See Also

Other utility: `crew_assert()`, `crew_clean()`, `crew_deprecate()`, `crew_random_name()`, `crew_retry()`, `crew_terminate_process()`, `crew_terminate_signal()`, `crew_worker()`

Examples

```
crew_eval(quote(1 + 1))
```

---

crew_launcher *Create an abstract launcher.*

Description

This function is useful for inheriting argument documentation in functions that create custom third-party launchers. See `@inheritParams crew::crew_launcher` in the source code file of `crew_launcher_local()`.

Usage

```
crew_launcher(
  name = NULL,
  seconds_interval = 0.5,
  seconds_timeout = 60,
  seconds_launch = 30,
  seconds_idle = Inf,
  seconds_wall = Inf,
  seconds_exit = NULL,
  tasks_max = Inf,
  tasks_timers = 0L,
  reset_globals = TRUE,
  reset_packages = FALSE,
  reset_options = FALSE,
  garbage_collection = FALSE,
  launch_max = 5L,
  tls = crew::crew_tls(),
  processes = NULL
)
```

Arguments

name	Name of the launcher.
seconds_interval	Number of seconds between polling intervals waiting for certain internal synchronous operations to complete, such as checking mirai::status().
seconds_timeout	Number of seconds until timing out while waiting for certain synchronous operations to complete, such as checking mirai::status().
seconds_launch	Seconds of startup time to allow. A worker is unconditionally assumed to be alive from the moment of its launch until seconds_launch seconds later. After seconds_launch seconds, the worker is only considered alive if it is actively connected to its assign websocket.
seconds_idle	Maximum number of seconds that a worker can idle since the completion of the last task. If exceeded, the worker exits. But the timer does not launch until tasks_timers tasks have completed. See the idletime argument of mirai::daemon(). crew does not excel with perfectly transient workers because it does not micromanage the assignment of tasks to workers, so please allow enough idle time for a new worker to be delegated a new task.
seconds_wall	Soft wall time in seconds. The timer does not launch until tasks_timers tasks have completed. See the walltime argument of mirai::daemon().
seconds_exit	Deprecated on 2023-09-21 in version 0.5.0.9002. No longer necessary.
tasks_max	Maximum number of tasks that a worker will do before exiting. See the maxtasks argument of mirai::daemon(). crew does not excel with perfectly transient workers because it does not micromanage the assignment of tasks to workers, it is recommended to set tasks_max to a value greater than 1.
tasks_timers	Number of tasks to do before activating the timers for seconds_idle and seconds_wall. See the timerstart argument of mirai::daemon().
reset_globals	TRUE to reset global environment variables between tasks, FALSE to leave them alone.
reset_packages	TRUE to unload any packages loaded during a task (runs between each task), FALSE to leave packages alone.
reset_options	TRUE to reset global options to their original state between each task, FALSE otherwise. It is recommended to only set reset_options = TRUE if reset_packages is also TRUE because packages sometimes rely on options they set at loading time.
garbage_collection	TRUE to run garbage collection between tasks, FALSE to skip.
launch_max	Positive integer of length 1, maximum allowed consecutive launch attempts which do not complete any tasks. Enforced on a worker-by-worker basis. The futile launch count resets to back 0 for each worker that completes a task. It is recommended to set launch_max above 0 because sometimes workers are unproductive under perfectly ordinary circumstances. But launch_max should still be small enough to detect errors in the underlying platform.
tls	A TLS configuration object from crew_tls().

processes NULL or positive integer of length 1, number of local processes to launch to allow worker launches to happen asynchronously. If NULL, then no local processes are launched. If 1 or greater, then the launcher starts the processes on start() and ends them on terminate(). Plugins that may use these processes should run asynchronous calls using launcher$async$eval() and expect a mirai task object as the return value.

See Also

Other launcher: crew_class_launcher

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(sockets = client$summary()$socket)
launcher$launch(index = 1L)
task <- mirai::mirai("result", .compute = client$name)
mirai::call_mirai_(task)
task$data
client$terminate()
}
```

---

crew_launcher_local *Create a launcher with local process workers.*

Description

Create an R6 object to launch and maintain local process workers.

Usage

```
crew_launcher_local(
  name = NULL,
  seconds_interval = 0.5,
  seconds_timeout = 60,
  seconds_launch = 30,
  seconds_idle = Inf,
  seconds_wall = Inf,
  seconds_exit = NULL,
  tasks_max = Inf,
  tasks_timers = 0L,
  reset_globals = TRUE,
  reset_packages = FALSE,
  reset_options = FALSE,
  garbage_collection = FALSE,
```

```
  launch_max = 5L,
  tls = crew::crew_tls(),
  local_log_directory = NULL,
  local_log_join = TRUE
)
```

Arguments

name	Name of the launcher.
seconds_interval	Number of seconds between polling intervals waiting for certain internal synchronous operations to complete, such as checking mirai::status().
seconds_timeout	Number of seconds until timing out while waiting for certain synchronous operations to complete, such as checking mirai::status().
seconds_launch	Seconds of startup time to allow. A worker is unconditionally assumed to be alive from the moment of its launch until seconds_launch seconds later. After seconds_launch seconds, the worker is only considered alive if it is actively connected to its assign websocket.
seconds_idle	Maximum number of seconds that a worker can idle since the completion of the last task. If exceeded, the worker exits. But the timer does not launch until tasks_timers tasks have completed. See the idletime argument of mirai::daemon(). crew does not excel with perfectly transient workers because it does not micromanage the assignment of tasks to workers, so please allow enough idle time for a new worker to be delegated a new task.
seconds_wall	Soft wall time in seconds. The timer does not launch until tasks_timers tasks have completed. See the walltime argument of mirai::daemon().
seconds_exit	Deprecated on 2023-09-21 in version 0.5.0.9002. No longer necessary.
tasks_max	Maximum number of tasks that a worker will do before exiting. See the maxtasks argument of mirai::daemon(). crew does not excel with perfectly transient workers because it does not micromanage the assignment of tasks to workers, it is recommended to set tasks_max to a value greater than 1.
tasks_timers	Number of tasks to do before activating the timers for seconds_idle and seconds_wall. See the timerstart argument of mirai::daemon().
reset_globals	TRUE to reset global environment variables between tasks, FALSE to leave them alone.
reset_packages	TRUE to unload any packages loaded during a task (runs between each task), FALSE to leave packages alone.
reset_options	TRUE to reset global options to their original state between each task, FALSE otherwise. It is recommended to only set reset_options = TRUE if reset_packages is also TRUE because packages sometimes rely on options they set at loading time.
garbage_collection	TRUE to run garbage collection between tasks, FALSE to skip.

launch_max — Positive integer of length 1, maximum allowed consecutive launch attempts which do not complete any tasks. Enforced on a worker-by-worker basis. The futile launch count resets to back 0 for each worker that completes a task. It is recommended to set launch_max above 0 because sometimes workers are unproductive under perfectly ordinary circumstances. But launch_max should still be small enough to detect errors in the underlying platform.

tls — A TLS configuration object from crew_tls().

local_log_directory — Either NULL or a character of length 1 with the file path to a directory to write worker-specific log files with standard output and standard error messages. Each log file represents a single *instance* of a running worker, so there will be more log files if a given worker starts and terminates a lot. Set to NULL to suppress log files (default).

local_log_join — Logical of length 1. If TRUE, crew will write standard output and standard error to the same log file for each worker instance. If FALSE, then they these two streams will go to different log files with informative suffixes.

See Also

Other plugin_local: crew_class_launcher_local, crew_controller_local()

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local(name = client$name)
launcher$start(sockets = client$summary()$socket)
launcher$launch(index = 1L)
task <- mirai::mirai("result", .compute = client$name)
mirai::call_mirai_(task)
task$data
client$terminate()
}
```

---

crew_monitor_local *Create a local monitor object.*

Description

Create an R6 object to monitor local processes created by crew and mirai.

Usage

```
crew_monitor_local()
```

See Also

Other monitor: crew_class_monitor_local

crew_random_name *Random name*

Description

Generate a random string that can be used as a name for a worker or task.

Usage

```
crew_random_name(n = 12L)
```

Arguments

n	Number of bytes of information in the random string hashed to generate the name. Larger n is more likely to generate unique names, but it may be slower to compute.

Details

The randomness is not reproducible and cannot be set with e.g. `set.seed()` in R.

Value

A random character string.

See Also

Other utility: `crew_assert()`, `crew_clean()`, `crew_deprecate()`, `crew_eval()`, `crew_retry()`, `crew_terminate_process()`, `crew_terminate_signal()`, `crew_worker()`

Examples

```
crew_random_name()
```

crew_relay *Create a* crew *relay object.*

Description

Create an R6 crew relay object.

Usage

```
crew_relay()
```

Details

A crew relay object keeps the signaling relationships among condition variables.

Value

An R6 crew relay object.

See Also

Other relay: crew_class_relay

Examples

```
crew_relay()
```

---

crew_retry *Retry code.*

Description

Repeatedly retry a function while it keeps returning FALSE and exit the loop when it returns TRUE

Usage

```
crew_retry(
  fun,
  args = list(),
  seconds_interval = 1,
  seconds_timeout = 60,
  max_tries = Inf,
  error = TRUE,
  message = character(0),
  envir = parent.frame()
)
```

Arguments

fun	Function that returns FALSE to keep waiting or TRUE to stop waiting.
args	A named list of arguments to fun.
seconds_interval	Nonnegative numeric of length 1, number of seconds to wait between calls to fun.
seconds_timeout	Nonnegative numeric of length 1, number of seconds to loop before timing out.
max_tries	Maximum number of calls to fun to try before giving up.
error	Whether to throw an error on a timeout or max tries.
message	Character of length 1, optional error message if the wait times out.
envir	Environment to evaluate fun.

Value

NULL (invisibly).

See Also

Other utility: crew_assert(), crew_clean(), crew_deprecate(), crew_eval(), crew_random_name(), crew_terminate_process(), crew_terminate_signal(), crew_worker()

Examples

```
crew_retry(fun = function() TRUE)
```

---

crew_terminate_process

Manually terminate a local process.

Description

Manually terminate a local process.

Usage

```
crew_terminate_process(pid)
```

Arguments

pid	Integer of length 1, process ID to terminate.

Value

NULL (invisibly).

See Also

Other utility: crew_assert(), crew_clean(), crew_deprecate(), crew_eval(), crew_random_name(), crew_retry(), crew_terminate_signal(), crew_worker()

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
process <- processx::process$new("sleep", "60")
process$is_alive()
crew_terminate_process(pid = process$get_pid())
process$is_alive()
}
```

crew_terminate_signal *Get the termination signal.*

Description

Get a supported operating system signal for terminating a local process.

Usage

```
crew_terminate_signal()
```

Value

An integer of length 1: `tools::SIGTERM` if your platform supports `SIGTERM`. If not, then `crew_crew_terminate_signal()()` checks `SIGQUIT`, then `SIGINT`, then `SIGKILL`, and then returns the first signal it finds that your operating system can use.

See Also

Other utility: `crew_assert()`, `crew_clean()`, `crew_deprecate()`, `crew_eval()`, `crew_random_name()`, `crew_retry()`, `crew_terminate_process()`, `crew_worker()`

Examples

```
crew_terminate_signal()
```

crew_throttle *Create a stateful throttling object.*

Description

Create an `R6` object for throttling.

Usage

```
crew_throttle(seconds_interval = 0.5)
```

Arguments

`seconds_interval`
: Positive numeric of length 1, throttling interval. The `poll()` method returns `TRUE` if and only if it was not called in the last `seconds_interval` seconds.

Details

Throttling is a technique that limits how often a function is called in a given period of time. crew_throttle() objects support the throttle argument of controller methods, which ensures auto-scaling only happen every seconds_interval seconds. This helps avoid overburdening the mirai dispatcher and other resources.

Value

An R6 object with throttle configuration settings and methods.

See Also

Other throttle: crew_class_throttle

Examples

```
throttle <- crew_throttle(seconds_interval = 0.5)
throttle$poll()
throttle$poll()
```

---

crew_tls *Configure TLS.*

Description

Create an R6 object with transport layer security (TLS) configuration for crew.

Usage

```
crew_tls(
  mode = "none",
  key = NULL,
  password = NULL,
  certificates = NULL,
  validate = TRUE
)
```

Arguments

mode	Character of length 1. Must be one of the following: • "none": disable TLS configuration. • "automatic": let mirai create a one-time key pair with a self-signed certificate. • "custom": manually supply a private key pair, an optional password for the private key, a certificate, an optional revocation list.
key	If mode is "none" or "automatic", then key is NULL. If mode is "custom", then key is a character of length 1 with the file path to the private key file.

password — If mode is "none" or "automatic", then password is NULL. If mode is "custom" and the private key is not encrypted, then password is still NULL. If mode is "custom" and the private key is encrypted, then password is a character of length 1 the the password of the private key. In this case, DO NOT SAVE THE PASSWORD IN YOUR R CODE FILES. See the keyring R package for solutions.

certificates — If mode is "none" or "automatic", then certificates is NULL. If mode is "custom", then certificates is a character vector of file paths to certificate files (signed public keys). If the certificate is self-signed or if it is directly signed by a certificate authority (CA), then only the certificate of the CA is needed. But if you have a whole certificate chain which begins at your own certificate and ends with the CA, then you can supply the whole certificate chain as a character vector which begins at your own certificate and ends with the certificate of the CA.

validate — Logical of length 1, whether to validate the configuration object on creation. If FALSE, then validate() can be called later on.

Details

crew_tls() objects are input to the tls argument of crew_client(), crew_controller_local(), etc. See https://wlandau.github.io/crew/articles/risks.html for details.

Value

An R6 object with TLS configuration settings and methods.

See Also

Other tls: crew_class_tls

Examples

```
crew_tls(mode = "automatic")
```

---

crew_worker *Crew worker.*

Description

Launches a crew worker which runs a mirai daemon. Not a user-side function. Users should not call crew_worker() directly. See launcher plugins like crew_launcher_local() for examples.

Usage

```
crew_worker(settings, launcher, worker, instance)
```

Arguments

settings	Named list of arguments to mirai::daemon().
launcher	Character of length 1, name of the launcher.
worker	Positive integer of length 1, index of the worker. This worker index remains the same even when the current instance of the worker exits and a new instance launches.
instance	Character of length 1 to uniquely identify the current instance of the worker.

Value

NULL (invisibly)

See Also

Other utility: crew_assert(), crew_clean(), crew_deprecate(), crew_eval(), crew_random_name(), crew_retry(), crew_terminate_process(), crew_terminate_signal()

Index