Package 'crew'

June 9, 2025

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 1.2.1

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, later, mirai (>= 2.0.1), nanonext (>= 1.6.0), processx, promises, ps, R6, rlang, stats, tibble, tidyselect, tools, utils

Suggests autometric (>= 0.1.0), 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 7.3.2

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2025-06-09 16:10:02 UTC

Contents

`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

host See crew_client().

port See crew_client().

tls See crew_client().

serialization 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.

url Client websocket URL.

profile Compute profile of the client.

condition Condition variable of the client.

client Process ID of the local process running the client.

dispatcher Process ID of the mirai dispatcher

Methods

Public methods:

- crew_class_client$new()
- crew_class_client$validate()
- crew_class_client$set_started()
- crew_class_client$start()
- crew_class_client$terminate()
- crew_class_client$resolved()
- crew_class_client$status()
- crew_class_client$pids()

Method new(): mirai client constructor.

Usage:

```
crew_class_client$new(
  host = NULL,
  port = NULL,
  tls = NULL,
  serialization = NULL,
  seconds_interval = NULL,
  seconds_timeout = NULL,
  relay = NULL
)
```

Arguments:

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

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

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

`serialization` 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 `set_started()`**:** Register the client as started.

Usage:

```
crew_class_client$set_started()
```

Details: Exported to implement the sequential controller. Only meant to be called manually inside the client or the sequential controller.

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

Usage:

```
crew_class_client$resolved()
```

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

Method `status()`**:** Internal function: return the `mirai` status of the compute profile.

Usage:

```
crew_class_client$status()
```

Details: Should only be called by the launcher, never by the user. The returned `events` field changes on every call and must be interpreted by the launcher before it vanishes.

Returns: A list with status information.

Method `pids()`**:** Get the process IDs of the local process and the `mirai` dispatcher (if started).

Usage:

```
crew_class_client$pids()
```

Returns: An integer vector of process IDs of the local process and the `mirai` dispatcher (if started).

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` Client 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.

`reset_globals` See `crew_controller()`. since the controller was started.

`reset_packages` See `crew_controller()`. since the controller was started.

`reset_options` See `crew_controller()`. since the controller was started.

`garbage_collection` See `crew_controller()`. since the controller was started.

`crashes_max` See `crew_controller()`.

`backup` See `crew_controller()`.

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

`backlog` A `crew_queue()` object tracking explicitly backlogged tasks.

`autoscaling` `TRUE` or `FALSE`, whether async `later`-based auto-scaling is currently running

`queue` Queue of resolved unpopped/uncollected 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$started()
- crew_class_controller$launch()
- crew_class_controller$scale()
- crew_class_controller$autoscale()
- crew_class_controller$descale()
- crew_class_controller$crashes()
- 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$cancel()
- crew_class_controller$pids()
- crew_class_controller$terminate()

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

Usage:

```
crew_class_controller$new(
  client = NULL,
  launcher = NULL,
  reset_globals = NULL,
  reset_packages = NULL,
  reset_options = NULL,
  garbage_collection = NULL,
  crashes_max = NULL,
  backup = NULL
)
```

Arguments:

`client` Client object. See crew_controller().

`launcher` Launcher object. See crew_controller().

`reset_globals` See crew_controller().

`reset_packages` See crew_controller().

`reset_options` See crew_controller().

`garbage_collection` See crew_controller().

`crashes_max` See crew_controller().

`backup` 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 controller.

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 `started()`**:** Check whether the controller is started.

Usage:

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

Arguments:

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

Details: Actually checks whether the client is started.

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

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

Usage:

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

Arguments:

`n` Number of workers to launch.

`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 launches new workers to run tasks if needed.

Returns: Invisibly returns `TRUE` if there was any relevant auto-scaling activity (new worker launches or worker connection/disconnection events) (`FALSE` otherwise).

Method `autoscale()`**:** Run worker auto-scaling in a private `later` loop every `controller$client$seconds_interval` seconds.

Usage:

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

Arguments:

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

Details: Call `controller$descale()` to terminate the auto-scaling loop.

Returns: NULL (invisibly).

Method `descale()`**:** Terminate the auto-scaling loop started by `controller$autoscale()`.

Usage:

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

Arguments:

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

Returns: NULL (invisibly).

Method `crashes()`**:** Report the number of consecutive crashes of a task.

Usage:

```
crew_class_controller$crashes(name, controllers = NULL)
```

Arguments:

`name` Character string, name of the task to check.

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

Details: See the `crashes_max` argument of `crew_controller()`.

Returns: Non-negative integer, number of consecutive times the task crashed.

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 = NULL,
  save_command = NULL,
  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` Character string, name of the task. If `NULL`, then a random name is generated automatically. The name of the task must not conflict with the name of another task pushed to the controller. Any previous task with the same name must first be popped before a new task with that name can be pushed.

`save_command` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004) and no longer used.

`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 = NULL,
  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_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` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004). The command is always saved now.

`verbose` Logical of length 1, whether to print to a progress bar when pushing tasks.

`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 = NULL,
  seconds_timeout = NULL,
  names = NULL,
  save_command = NULL,
  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` Deprecated on 2025-01-17 (`crew` version 0.10.2.9003). Instead, the `seconds_interval` argument passed to `crew_controller_group()` is used as `seconds_max` in a `crew_throttle()` object which orchestrates exponential backoff.

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

`names` Optional character string, 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` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004). The command is always saved now.

`error` Character of length 1, choice of action if a task was not successful. 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. NOTE: the only kinds of errors considered here are errors at the R level. A crashed tasks will return a status of `"crash"` in the output and not trigger an error in `map()` unless `crashes_max` is reached.

`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 to a progress bar as tasks resolve.

`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 = NULL,
  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` `NULL` or 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.
- `NULL` or `"silent"`: do not react to errors. NOTE: the only kinds of errors considered here are errors at the R level. A crashed tasks will return a status of `"crash"` in the output and not trigger an error in `pop()` unless `crashes_max` is reached.

`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.

- command: a character string with the R command.
- result: a list containing the return value of the R command. NA if the task failed.
- status: a character string. "success" if the task succeeded, "cancel" if the task was canceled with the cancel() controller method, "crash" if the worker running the task exited before it could complete the task, or "error" for any other kind of error.
- error: the first 2048 characters of the error message if the task status is not "success", NA otherwise. Messages for crashes and cancellations are captured here alongside ordinary R-level errors.
- code: an integer code denoting the specific exit status: 0 for successful tasks, -1 for tasks with an error in the R command of the task, and another positive integer with an NNG status code if there is an error at the NNG/nanonext level. nanonext::nng_error() can interpret these codes.
- 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.
- 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.
- controller: name of the crew controller where the task ran.
- worker: name of the crew worker that ran the task.

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

Usage:

```
crew_class_controller$collect(
  scale = TRUE,
  throttle = TRUE,
  error = NULL,
  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.

error NULL or 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. * NULL or "silent": do not react to errors. NOTE: the only kinds of errors considered here are errors at the R level. A crashed tasks will return a status of "crash" in the output and not trigger an error in collect() unless crashes_max is reached.

`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. Returns `NULL` if there are no tasks to collect. See `pop()` for details on the columns of the returned `tibble`.

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 = 1,
  scale = NULL,
  throttle = NULL,
  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` Deprecated on 2024-04-10 (version 0.9.1.9003) and no longer used. Now, `promise()` always turns on auto-scaling in a private `later` loop (if not already activated).

`throttle` Deprecated on 2024-04-10 (version 0.9.1.9003) and no longer used. Now, `promise()` always turns on auto-scaling in a private `later` loop (if not already activated).

`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 = NULL,
  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` Deprecated on 2025-01-17 (`crew` version 0.10.2.9003). Instead, the `seconds_interval` argument passed to `crew_controller_group()` is used as `seconds_max` in a `crew_throttle()` object which orchestrates exponential backoff.

`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 tasks that ran on a worker and then were returned by `pop()` or `collect()`. It has one row and the following columns:

- `controller`: name of the controller.
- `tasks`: number of tasks.
- `seconds`: total number of runtime in seconds.
- `success`: total number of successful tasks.
- `error`: total number of tasks with errors, either in the R code of the task or an NNG-level error that is not a cancellation or crash.
- `crash`: total number of crashed tasks (where the worker exited unexpectedly while it was running the task).
- `cancel`: total number of tasks interrupted with the `cancel()` controller method.
- `warnings`: total number of tasks with one or more warnings.

Method `cancel()`**:** Cancel one or more tasks.

Usage:

```
crew_class_controller$cancel(names = character(0L), all = FALSE)
```

Arguments:

`names` Character vector of names of tasks to cancel. Those names must have been manually supplied by `push()`.

`all` `TRUE` to cancel all tasks, `FALSE` otherwise. `all = TRUE` supersedes the `names` argument.

Returns: `NULL` (invisibly).

Method `pids()`**:** Get the process IDs of the local process and the `mirai` dispatcher (if started).

Usage:

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

Arguments:

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

Returns: An integer vector of process IDs of the local process and the `mirai` dispatcher (if started).

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.

`throttle` `crew_throttle()` object to orchestrate exponential backoff in the relay and auto-scaling.

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$start()`
- `crew_class_controller_group$started()`
- `crew_class_controller_group$launch()`
- `crew_class_controller_group$scale()`
- `crew_class_controller_group$autoscale()`
- `crew_class_controller_group$descale()`
- `crew_class_controller_group$crashes()`
- `crew_class_controller_group$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$summary()
- crew_class_controller_group$pids()
- crew_class_controller_group$terminate()

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

Usage:

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

Arguments:

`controllers` List of `R6` controller objects.

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

`throttle` crew_throttle() object to orchestrate exponential backoff in the relay and auto-scaling.

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 `started()`**:** Check whether all the given controllers are started.

Usage:

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

Arguments:

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

Details: Actually checks whether all the given clients are started.

Returns: `TRUE` if the controllers are started, `FALSE` if any are not.

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: Invisibly returns `TRUE` if there was any relevant auto-scaling activity (new worker launches or worker connection/disconnection events) (`FALSE` otherwise).

Method `autoscale()`**:** Run worker auto-scaling in a private `later` loop every `controller$client$seconds_interval` seconds.

Usage:

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

Arguments:

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

Returns: `NULL` (invisibly).

Method `descale()`**:** Terminate the auto-scaling loop started by `controller$autoscale()`.

Usage:

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

Arguments:

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

Returns: `NULL` (invisibly).

Method `crashes()`**:** Report the number of consecutive crashes of a task, summed over all selected controllers in the group.

Usage:

```
crew_class_controller_group$crashes(name, controllers = NULL)
```

Arguments:

`name` Character string, name of the task to check.

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

Details: See the `crashes_max` argument of `crew_controller()`.

Returns: Number of consecutive crashes of the named task, summed over all the controllers in the group.

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 = NULL,
  save_command = NULL,
  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` Character string, name of the task. If `NULL`, a random name is automatically generated. The task name must not conflict with an existing task in the controller where it is submitted. To reuse the name, wait for the existing task to finish, then either `pop()` or `collect()` it to remove it from its controller.

`save_command` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004).

`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 = NULL,
  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_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` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004).

`verbose` Logical of length 1, whether to print to a progress bar when pushing tasks.

`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 = NULL,
  seconds_timeout = NULL,
  names = NULL,
  save_command = NULL,
  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` Deprecated on 2025-01-17 (`crew` version 0.10.2.9003). Instead, the `seconds_interval` argument passed to `crew_controller_group()` is used as `seconds_max` in a `crew_throttle()` object which orchestrates exponential backoff.

`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` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004).

`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 = NULL,
  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` `NULL` or 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.
- `NULL` or `"silent"`: do not react to errors.

`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,
  error = NULL,
  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.

`error` `NULL` or 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.
- `NULL` or `"silent"`: do not react to errors.

`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 = NULL,
  throttle = NULL,
  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` Deprecated on 2024-04-10 (version 0.9.1.9003) and no longer used. Now, `promise()` always turns on auto-scaling in a private `later` loop (if not already activated).

`throttle` Deprecated on 2024-04-10 (version 0.9.1.9003) and no longer used. Now, `promise()` always turns on auto-scaling in a private `later` loop (if not already activated).

`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 = NULL,
  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` Deprecated on 2025-01-17 (`crew` version 0.10.2.9003). Instead, the `seconds_interval` argument passed to `crew_controller_group()` is used as `seconds_max` in a `crew_throttle()` object which orchestrates exponential backoff.

`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 `pids()`**:** Get the process IDs of the local process and the `mirai` dispatchers (if started).

Usage:

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

Arguments:

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

Returns: An integer vector of process IDs of the local process and the `mirai` dispatcher (if started).

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_controller_sequential` *Sequential controller class*

Description

R6 class for sequential controllers.

Details

See `crew_controller_sequential()`.

Super class

`crew::crew_class_controller` -> `crew_class_controller_sequential`

Methods

Public methods:

- `crew_class_controller_sequential$start()`
- `crew_class_controller_sequential$launch()`
- `crew_class_controller_sequential$scale()`
- `crew_class_controller_sequential$autoscale()`
- `crew_class_controller_sequential$descale()`
- `crew_class_controller_sequential$push()`
- `crew_class_controller_sequential$wait()`
- `crew_class_controller_sequential$push_backlog()`
- `crew_class_controller_sequential$pop_backlog()`
- `crew_class_controller_sequential$cancel()`

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

Usage:

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

Arguments:

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

Details: For the sequential controller, there is nothing to do except register the client as started.

Returns: `NULL` (invisibly).

Method `launch()`**:** Does nothing for the sequential controller.

Usage:

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

Arguments:

`n` Number of workers to launch.

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

Returns: `NULL` (invisibly).

Method `scale()`**:** Does nothing for the sequential controller.

Usage:

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

Arguments:

`throttle` Not applicable to the sequential controller.

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

Returns: Invisibly returns `FALSE`.

Method `autoscale()`**:** Not applicable to the sequential controller.

Usage:

```
crew_class_controller_sequential$autoscale(controllers = NULL)
```

Arguments:

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

Returns: `NULL` (invisibly).

Method `descale()`**:** Not applicable to the sequential controller.

Usage:

```
crew_class_controller_sequential$descale(controllers = NULL)
```

Arguments:

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

Returns: `NULL` (invisibly).

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

Usage:

```
crew_class_controller_sequential$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 = NULL,
  save_command = NULL,
  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` for the sequential controller, then the random number generator defaults to the current RNG of the local R session where the sequential controller lives.

`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` for the sequential controller, then the random number generator defaults to the current RNG of the local R session where the sequential controller lives.

`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` Not used in the sequential controller..

`scale` Not used in the sequential controller.

`throttle` Not used in the sequential controller.

`name` Character string, name of the task. If `NULL`, then a random name is generated automatically. The name of the task must not conflict with the name of another task pushed to the controller. Any previous task with the same name must first be popped before a new task with that name can be pushed.

`save_command` Deprecated on 2025-01-22 (`crew` version 0.10.2.9004) and no longer used.

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

Returns: Invisibly returns a `mirai`-like list where the `data` element is the result of the task.

Method `wait()`**:** Not applicable to the sequential controller.

Usage:

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

Arguments:

`mode` Not applicable to the sequential controller.

`seconds_interval` Not applicable to the sequential controller.

`seconds_timeout` Not applicable to the sequential controller.

`scale` Not applicable to the sequential controller.

`throttle` Not applicable to the sequential controller.

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

Returns: Always returns `TRUE` (invisibly) for the sequential controller.

Method `push_backlog()`**:** Not applicable to the sequential controller.

Usage:

```
crew_class_controller_sequential$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.

Returns: `NULL` (invisibly).

Method `pop_backlog()`**:** Not applicable to the sequential controller.

Usage:

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

Arguments:

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

Returns: Always `character(0L)` for the sequential controller.

Method `cancel()`**:** Not applicable to the sequential controller.

Usage:

```
crew_class_controller_sequential$cancel(names = character(0L), all = FALSE)
```

Arguments:

`names` Not applicable to the sequential controller.

`all` Not applicable to the sequential controller.

See Also

Other sequential controllers: `crew_controller_sequential()`

Examples

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

---

crew_class_launcher *Launcher abstract class*

Description

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

Active bindings

`name` See `crew_launcher()`.

`workers` 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()`.

`tasks_max` See `crew_launcher()`.

`tasks_timers` See `crew_launcher()`.

`tls` See `crew_launcher()`.

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

`r_arguments` See `crew_launcher()`.

`options_metrics` See `crew_launcher()`.

`url` Websocket URL for worker connections.

`profile` `mirai` compute profile of the launcher.

`instances` Data frame of worker instance information.

`id` Integer worker ID from the last call to `settings()`.

`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$poll()
- crew_class_launcher$settings()
- crew_class_launcher$call()
- crew_class_launcher$start()
- crew_class_launcher$terminate()
- crew_class_launcher$resolve()
- crew_class_launcher$update()
- crew_class_launcher$launch()
- crew_class_launcher$scale()
- crew_class_launcher$launch_worker()
- crew_class_launcher$terminate_worker()
- crew_class_launcher$terminate_workers()
- crew_class_launcher$crashes()
- crew_class_launcher$set_name()

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

Usage:

```
crew_class_launcher$new(
  name = NULL,
  workers = 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,
  crashes_error = NULL,
  launch_max = NULL,
  tls = NULL,
  processes = NULL,
  r_arguments = NULL,
  options_metrics = NULL
)
```

Arguments:

`name` See `crew_launcher()`.

workers 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 Deprecated. See crew_launcher().
reset_packages Deprecated. See crew_launcher().
reset_options Deprecated. See crew_launcher().
garbage_collection Deprecated. See crew_launcher().
crashes_error See crew_launcher().
launch_max Deprecated.
tls See crew_launcher().
processes See crew_launcher().
r_arguments See crew_launcher().
options_metrics See crew_launcher().

Returns: An R6 object with the launcher.

Examples:

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local()
launcher$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
mirai::call_mirai(task)
task$data
client$terminate()
}
```

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

Usage:

```
crew_class_launcher$validate()
```

Returns: NULL (invisibly).

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

Usage:

```
crew_class_launcher$poll()
```

Returns: TRUE to run whatever work comes next, FALSE to skip until the appropriate time.

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

Usage:

```
crew_class_launcher$settings()
```

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(
  worker,
  socket = NULL,
  launcher = NULL,
  instance = NULL
)
```

Arguments:

`worker` Character string, name of the worker.

`socket` Deprecated on 2025-01-28 (`crew` version 1.0.0).

`launcher` Deprecated on 2025-01-28 (`crew` version 1.0.0).

`instance` Deprecated on 2025-01-28 (`crew` version 1.0.0).

Returns: Character string with a call to `crew_worker()`.

Examples:

```
launcher <- crew_launcher_local()
launcher$start(url = "tcp://127.0.0.1:57000", profile = "profile")
launcher$call(worker = "worker_name")
launcher$terminate()
```

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

Usage:

```
crew_class_launcher$start(url = NULL, profile = NULL, sockets = NULL)
```

Arguments:

`url` Character string, websocket URL for worker connections.

`profile` Character string, `mirai` compute profile.

`sockets` Deprecated on 2025-01-28 (`crew` version 1.0.0).

Returns: `NULL` (invisibly).

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

Usage:

```
crew_class_launcher$terminate()
```

Returns: `NULL` (invisibly).

Method `resolve()`**:** Resolve asynchronous worker submissions.

Usage:

```
crew_class_launcher$resolve()
```

Returns: `NULL` (invisibly). Throw an error if there were any asynchronous worker submission errors.'

Method `update()`**:** Update worker metadata, resolve asynchronous worker submissions, and terminate lost workers.

Usage:

```
crew_class_launcher$update(status)
```

Arguments:

`status` A `mirai` status list.

Returns: `NULL` (invisibly).

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

Usage:

```
crew_class_launcher$launch()
```

Returns: Handle of the launched worker.

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

Usage:

```
crew_class_launcher$scale(status, throttle = NULL)
```

Arguments:

`status` A `mirai` status list with worker and task information.

`throttle` Deprecated, only used in the controller as of 2025-01-16 (`crew` version 0.10.2.9003).

Returns: Invisibly returns `TRUE` if there was any relevant auto-scaling activity (new worker launches or worker connection/disconnection events) (`FALSE` otherwise).

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

Usage:

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

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.

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 all workers.

Usage:

```
crew_class_launcher$terminate_workers()
```

Returns: `NULL` (invisibly).

Method `crashes()`**:** Deprecated on 2025-01-28 (`crew` version 1.0.0).

Usage:

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

Arguments:

`index` Unused argument.

Returns: The integer 1, for compatibility.

Method `set_name()`**:** Deprecated on 2025-01-28 (`crew` version 1.0.0).

Usage:

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

Arguments:

`name` Name to set for the launcher.

Returns: `NULL` (invisibly).

See Also

Other launcher: `crew_launcher()`

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local()
launcher$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
mirai::call_mirai(task)
task$data
client$terminate()
}

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

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

```
launcher <- crew_launcher_local()
launcher$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
mirai::call_mirai(task)
task$data
client$terminate()
}

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

launcher <- crew_launcher_local()
launcher$start(url = "tcp://127.0.0.1:57000", profile = "profile")
launcher$call(worker = "worker_name")
launcher$terminate()
```

`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

`options_local` 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,
  workers = 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,
  crashes_error = NULL,
  tls = NULL,
  processes = NULL,
  r_arguments = NULL,
  options_metrics = NULL,
  options_local = NULL
)
```

Arguments:

`name` See `crew_launcher()`.

`workers` 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()`.

`crashes_error` See `crew_launcher()`.

`tls` See `crew_launcher()`.

`processes` See `crew_launcher()`.

`r_arguments` See `crew_launcher()`.

`options_metrics` See `crew_launcher_local()`.

`options_local` See `crew_launcher_local()`.

`reset_globals` Deprecated. See `crew_launcher()`.

`reset_packages` Deprecated. See `crew_launcher()`.

`reset_options` Deprecated. See `crew_launcher()`.

`garbage_collection` Deprecated. See `crew_launcher()`.

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(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
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)
```

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` and `worker` arguments described below.

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

`worker` Character string, name of the worker within 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$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
mirai::call_mirai(task)
task$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$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
mirai::call_mirai(task)
task$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_queue R6 *queue class*

Description

R6 class for a queue.

Details

See the Details section of crew_queue(). The R6 crew queue class is not portable (for efficiency), so other packages should not inherit from it. The reason for non-portability is efficiency: elements can be directly accessed without self$ or private$, and they can be directly modified with <<-. This is especially important for push() because envir$vector[slice] <- x copies the entire vector in memory, which has O(n^2) complexity and is extremely slow for large vectors.

Active bindings

data See crew_queue().

head Non-negative integer pointing to the location of the next element to pop.

tail Non-negative integer pointing to the tail of the queue.

step See crew_queue().

Methods

Public methods:

- crew_class_queue$new()
- crew_class_queue$validate()
- crew_class_queue$empty()
- crew_class_queue$nonempty()
- crew_class_queue$list()
- crew_class_queue$reset()
- crew_class_queue$clean()
- crew_class_queue$set()
- crew_class_queue$extend()
- crew_class_queue$push()
- crew_class_queue$pop()
- crew_class_queue$collect()

Method new(): Create a queue object.

Usage:

```
crew_class_queue$new(data = NULL, step = NULL)
```

Arguments:

data See crew_queue().

`step` See `crew_queue()`.

Returns: A queue object.

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

Usage:

```
crew_class_queue$validate()
```

Returns: `NULL` (invisibly). Called for its side effects.

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

Usage:

```
crew_class_queue$empty()
```

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

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

Usage:

```
crew_class_queue$nonempty()
```

Returns: `TRUE` if the queue is nonempty, `FALSE` otherwise.

Method `list()`**:** List available data.

Usage:

```
crew_class_queue$list()
```

Returns: Character vector of available data.

Method `reset()`**:** Reset the queue.

Usage:

```
crew_class_queue$reset()
```

Returns: `NULL` (invisibly). Called for its side effects.

Method `clean()`**:** Remove popped elements from the data in the queue.

Usage:

```
crew_class_queue$clean()
```

Returns: `NULL` (invisibly).

Method `set()`**:** Set the data in the queue.

Usage:

```
crew_class_queue$set(data = character(0L))
```

Arguments:

`data` Character vector of data to set.

Returns: `NULL` (invisibly). Called for its side effects.

Method `extend()`**:** Extend the queue data by `step` elements.

Usage:

```
crew_class_queue$extend(n)
```

Arguments:

`n` Positive integer, number of elements to extend the queue data.

Returns: `NULL` (invisibly).

Method `push()`**:** Append new elements to the queue.

Usage:

```
crew_class_queue$push(x)
```

Arguments:

`x` Character vector of new data to append.

Details: `push()` is the reason the queue class is not portable. According to R6 documentation, members of non-portable classes can be accessed without `self$` or `private$`, and assignment can be done with `<<-`. In the case of `push()`, this prevents each assignment from deep-copying the entire contents of the vector.

Returns: `NULL` (invisibly).

Method `pop()`**:** Pop one or more elements off the queue.

Usage:

```
crew_class_queue$pop(n = 1L)
```

Arguments:

`n` Positive integer, maximum number of elements to pop. Fewer than `n` are popped if fewer than `n` are available.

Returns: Character vector of elements popped off the queue. `NULL` if there are no more elements available to pop.

Method `collect()`**:** Remove and return all available elements off the queue.

Usage:

```
crew_class_queue$collect()
```

Returns: Character vector, elements collected from the queue. `NULL` if there are no more elements available to collect.

See Also

Other queue: `crew_queue()`

Examples

```
crew_queue()
```

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.

throttle A crew_throttle() object for wait().

Methods

Public methods:

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

Method new(): Relay constructor.

Usage:

```
crew_class_relay$new(throttle)
```

Arguments:

throttle A crew_throttle() object.

Returns: A crew_relay() object.

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. Use the throttle to determine the waiting time.

Usage:

```
crew_class_relay$wait()
```

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_max See crew_throttle().

seconds_min See crew_throttle().

seconds_start See crew_throttle().

base See crew_throttle().

seconds_interval Current wait time interval.

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$poll()
- crew_class_throttle$accelerate()
- crew_class_throttle$decelerate()
- crew_class_throttle$reset()
- crew_class_throttle$update()

Method new(): Throttle constructor.

Usage:

```
crew_class_throttle$new(
  seconds_max = NULL,
  seconds_min = NULL,
  seconds_start = NULL,
  base = NULL
)
```

Arguments:

seconds_max See crew_throttle().

seconds_min See crew_throttle().

`seconds_start` See `crew_throttle()`.

`base` See `crew_throttle()`.

Returns: An `R6` object with throttle configuration.

Examples:

```
throttle <- crew_throttle(seconds_max = 1)
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 `max` seconds, `FALSE` otherwise.

Method `accelerate()`**:** Divide `seconds_interval` by `base`.

Usage:

```
crew_class_throttle$accelerate()
```

Returns: `NULL` (invisibly). Called for its side effects.

Method `decelerate()`**:** Multiply `seconds_interval` by `base`.

Usage:

```
crew_class_throttle$decelerate()
```

Returns: `NULL` (invisibly). Called for its side effects.

Method `reset()`**:** Reset the throttle object so the next `poll()` returns `TRUE`, and reset the wait time interval to its initial value.

Usage:

```
crew_class_throttle$reset()
```

Returns: `NULL` (invisibly).

Method `update()`**:** Reset the throttle when there is activity and decelerate it gradually when there is no activity.

Usage:

```
crew_class_throttle$update(activity)
```

Arguments:

`activity` `TRUE` if there is activity, `FALSE` otherwise.

Returns: `NULL` (invisibly).

See Also

Other throttle: `crew_throttle()`

Examples

```
throttle <- crew_throttle(seconds_max = 1)
throttle$poll()
throttle$poll()

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

throttle <- crew_throttle(seconds_max = 1)
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()`
- `crew_class_tls$url()`

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(profile)
```

Arguments:

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

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

Method `url()`**:** Form the URL for `crew` worker connections.

Usage:

```
crew_class_tls$url(host, port)
```

Arguments:

`host` Character string with the host name or IP address.

`port` Non-negative integer with the port number (0 to let NNG select a random ephemeral port).

Returns: Character string with the URL.

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 = 1,
  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 = NULL,
  host = NULL,
  port = NULL,
  tls = crew::crew_tls(),
  tls_enable = NULL,
  tls_config = NULL,
  serialization = NULL,
  seconds_interval = 1,
  seconds_timeout = 60,
  retry_tasks = NULL
)
```

Arguments

name	Deprecated on 2025-01-14 (crew version 0.10.2.9002).
workers	Deprecated on 2025-01-13 (crew version 0.10.2.9002).
host	IP address of the mirai client to send and receive tasks. If NULL, the host defaults to nanonext::ip_addr()[1].
port	TCP port to listen for the workers. If NULL, then an available ephemeral port is automatically chosen. Controllers running simultaneously on the same computer (as in a controller group) must not share the same TCP port.
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.
serialization	Either NULL (default) or an object produced by mirai::serial_config() to control the serialization of data sent to workers. This can help with either more efficient data transfers or to preserve attributes of otherwise non-exportable objects (such as torch tensors or arrow tables). See ?mirai::serial_config for details.
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().
retry_tasks	Deprecated on 2025-01-13 (crew version 0.10.2.9002).

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,
  reset_globals = TRUE,
  reset_packages = FALSE,
  reset_options = FALSE,
  garbage_collection = FALSE,
  crashes_max = 5L,
  backup = NULL,
  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().
reset_globals	TRUE to reset global environment variables between tasks, FALSE to leave them alone.
reset_packages	TRUE to detach any packages loaded during a task (runs between each task), FALSE to leave packages alone. In either case, the namespaces are not detached.
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. for this and other reasons, reset_options only resets options that were nonempty at the beginning of the task. If your task sets an entirely new option not already in options(), then reset_options = TRUE does not delete the option.
garbage_collection	TRUE to run garbage collection after each task task, FALSE to skip.
crashes_max	In rare cases, a worker may exit unexpectedly before it completes its current task. If this happens, pop() returns a status of "crash" instead of "error" for the task. The controller does not automatically retry the task, but you can retry it manually by calling push() again and using the same task name as before. (However, targets pipelines running crew do automatically retry tasks whose workers crashed.) crashes_max is a non-negative integer, and it sets the maximum number of allowable consecutive crashes for a given task. If a task's worker crashes more than crashes_max times in a row, then pop() throws an error when it tries to return the results of the task.
backup	An optional crew controller object, or NULL to omit. If supplied, the backup controller runs any pushed tasks that have already reached crashes_max consecutive crashes. Using backup, you can create a chain of controllers with different levels of resources (such as worker memory and CPUs) so that a task that fails on one controller can retry using incrementally more powerful workers.

All controllers in a backup chain should be part of the same controller group (see crew_controller_group()) so you can call the group-level pop() and collect() methods to make sure you get results regardless of which controller actually ended up running the task.

Limitations of backup: * crashes_max needs to be positive in order for backup to be used. Otherwise, every task would always skip the current controller and go to backup. * backup cannot be a controller group. It must be an ordinary controller.

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(..., seconds_interval = 1)
```

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.

seconds_interval — Number of seconds between polling intervals waiting for certain internal synchronous operations to complete, such as checking mirai::status()

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,
  serialization = NULL,
  seconds_interval = 1,
  seconds_timeout = 60,
  seconds_launch = 30,
  seconds_idle = 300,
  seconds_wall = Inf,
  seconds_exit = NULL,
  retry_tasks = NULL,
  tasks_max = Inf,
  tasks_timers = 0L,
  reset_globals = TRUE,
  reset_packages = FALSE,
  reset_options = FALSE,
  garbage_collection = FALSE,
  crashes_error = NULL,
```

```
  launch_max = NULL,
  r_arguments = c("--no-save", "--no-restore"),
  crashes_max = 5L,
  backup = NULL,
  options_metrics = crew::crew_options_metrics(),
  options_local = crew::crew_options_local(),
  local_log_directory = NULL,
  local_log_join = NULL
)
```

Arguments

name	Character string, name of the launcher. If the name is NULL, then a name is automatically generated when the launcher starts.
workers	Maximum number of workers to run concurrently when auto-scaling, excluding task retries and manual calls to launch(). Special workers allocated for task retries do not count towards this limit, so the number of workers running at a given time may exceed this maximum. A smaller number of workers may run if the number of executing tasks is smaller than the supplied value of the workers argument.
host	IP address of the mirai client to send and receive tasks. If NULL, the host defaults to nanonext::ip_addr()[1].
port	TCP port to listen for the workers. If NULL, then an available ephemeral port is automatically chosen. Controllers running simultaneously on the same computer (as in a controller group) must not share the same TCP port.
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.
serialization	Either NULL (default) or an object produced by mirai::serial_config() to control the serialization of data sent to workers. This can help with either more efficient data transfers or to preserve attributes of otherwise non-exportable objects (such as torch tensors or arrow tables). See ?mirai::serial_config for details.
seconds_interval	Number of seconds between polling intervals waiting for certain internal synchronous operations to complete. In certain cases, exponential backoff is used with this argument passed to seconds_max in a crew_throttle() object.
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.

retry_tasks
: Deprecated on 2025-01-13 (crew version 0.10.2.9002).

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 detach any packages loaded during a task (runs between each task), FALSE to leave packages alone. In either case, the namespaces are not detached.

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. for this and other reasons, reset_options only resets options that were nonempty at the beginning of the task. If your task sets an entirely new option not already in options(), then reset_options = TRUE does not delete the option.

garbage_collection
: TRUE to run garbage collection after each task task, FALSE to skip.

crashes_error
: Deprecated on 2025-01-13 (crew version 0.10.2.9002).

launch_max
: Deprecated on 2024-11-04 (crew version 0.10.2.9002).

r_arguments
: Optional character vector of command line arguments to pass to Rscript (non-Windows) or Rscript.exe (Windows) when starting a worker. Example: r_arguments = c("--vanilla", "--max-connections=32").

crashes_max
: In rare cases, a worker may exit unexpectedly before it completes its current task. If this happens, pop() returns a status of "crash" instead of "error" for the task. The controller does not automatically retry the task, but you can retry it manually by calling push() again and using the same task name as before. (However, targets pipelines running crew do automatically retry tasks whose workers crashed.)

  crashes_max is a non-negative integer, and it sets the maximum number of allowable consecutive crashes for a given task. If a task's worker crashes more than crashes_max times in a row, then pop() throws an error when it tries to return the results of the task.

backup
: An optional crew controller object, or NULL to omit. If supplied, the backup controller runs any pushed tasks that have already reached crashes_max consecutive crashes. Using backup, you can create a chain of controllers with different levels of resources (such as worker memory and CPUs) so that a task that

fails on one controller can retry using incrementally more powerful workers. All controllers in a backup chain should be part of the same controller group (see crew_controller_group()) so you can call the group-level pop() and collect() methods to make sure you get results regardless of which controller actually ended up running the task.

Limitations of backup: * crashes_max needs to be positive in order for backup to be used. Otherwise, every task would always skip the current controller and go to backup. * backup cannot be a controller group. It must be an ordinary controller.

options_metrics

Either NULL to opt out of resource metric logging for workers, or an object from crew_options_metrics() to enable and configure resource metric logging for workers. For resource logging to run, the autometric R package version 0.1.0 or higher must be installed.

options_local An object generated by crew_options_local() with options specific to the local controller.

local_log_directory

Deprecated on 2024-10-08. Use options_local instead.

local_log_join Deprecated on 2024-10-08. Use options_local instead.

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_controller_sequential

Create a sequential controller.

Description

The sequential controller runs tasks on the same R process where the controller object exists. Tasks run sequentially rather than in parallel.

Usage

```
crew_controller_sequential()
```

See Also

Other sequential controllers: crew_class_controller_sequential

Examples

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

---

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,
  data = list(),
  globals = list(),
  seed = NULL,
  algorithm = NULL,
  packages = character(0),
  library = NULL,
  reset_globals = TRUE,
  reset_packages = FALSE,
  reset_options = FALSE,
  garbage_collection = FALSE
)
```

Arguments

command	Language object with R code to run.
name	Character of length 1, name of the task.
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().
reset_globals	TRUE to reset global environment variables between tasks, FALSE to leave them alone.
reset_packages	TRUE to detach any packages loaded during a task (runs between each task), FALSE to leave packages alone. In either case, the namespaces are not detached.
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. for this and other reasons, reset_options only resets options that were nonempty at the beginning of the task. If your task sets an entirely new option not already in options(), then reset_options = TRUE does not delete the option.
garbage_collection	TRUE to run garbage collection after each task task, FALSE to skip.

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), name = "task_name")
```

---

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,
  workers = 1L,
  seconds_interval = 1,
  seconds_timeout = 60,
  seconds_launch = 30,
  seconds_idle = 300,
  seconds_wall = Inf,
  seconds_exit = NULL,
  tasks_max = Inf,
  tasks_timers = 0L,
  reset_globals = NULL,
  reset_packages = NULL,
  reset_options = NULL,
  garbage_collection = NULL,
  crashes_error = NULL,
  launch_max = NULL,
  tls = crew::crew_tls(),
  processes = NULL,
  r_arguments = c("--no-save", "--no-restore"),
  options_metrics = crew::crew_options_metrics()
)
```

Arguments

name	Character string, name of the launcher. If the name is NULL, then a name is automatically generated when the launcher starts.

`workers` Maximum number of workers to run concurrently when auto-scaling, excluding task retries and manual calls to `launch()`. Special workers allocated for task retries do not count towards this limit, so the number of workers running at a given time may exceed this maximum. A smaller number of workers may run if the number of executing tasks is smaller than the supplied value of the `workers` argument.

`seconds_interval`
Number of seconds between polling intervals waiting for certain internal synchronous operations to complete. In certain cases, exponential backoff is used with this argument passed to `seconds_max` in a `crew_throttle()` object.

`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` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `reset_globals` option of `crew_controller()` instead.

`reset_packages` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `reset_packages` option of `crew_controller()` instead.

`reset_options` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `reset_options` option of `crew_controller()` instead.

`garbage_collection`
Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `garbage_collection` option of `crew_controller()` instead.

`crashes_error` Deprecated on 2025-01-13 (`crew` version 0.10.2.9002).

`launch_max` Deprecated on 2024-11-04 (`crew` version 0.10.2.9002).

`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.

r_arguments Optional character vector of command line arguments to pass to Rscript (non-Windows) or Rscript.exe (Windows) when starting a worker. Example: r_arguments = c("--vanilla", "--max-connections=32").

options_metrics Either NULL to opt out of resource metric logging for workers, or an object from crew_options_metrics() to enable and configure resource metric logging for workers. For resource logging to run, the autometric R package version 0.1.0 or higher must be installed.

See Also

Other launcher: crew_class_launcher

Examples

```
if (identical(Sys.getenv("CREW_EXAMPLES"), "true")) {
client <- crew_client()
client$start()
launcher <- crew_launcher_local()
launcher$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
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,
  workers = 1L,
  seconds_interval = 1,
  seconds_timeout = 60,
  seconds_launch = 30,
```

```
  seconds_idle = Inf,
  seconds_wall = Inf,
  seconds_exit = NULL,
  tasks_max = Inf,
  tasks_timers = 0L,
  reset_globals = NULL,
  reset_packages = NULL,
  reset_options = NULL,
  garbage_collection = NULL,
  crashes_error = NULL,
  launch_max = NULL,
  tls = crew::crew_tls(),
  r_arguments = c("--no-save", "--no-restore"),
  options_metrics = crew::crew_options_metrics(),
  options_local = crew::crew_options_local(),
  local_log_directory = NULL,
  local_log_join = NULL
)
```

Arguments

name	Character string, name of the launcher. If the name is NULL, then a name is automatically generated when the launcher starts.
workers	Maximum number of workers to run concurrently when auto-scaling, excluding task retries and manual calls to launch(). Special workers allocated for task retries do not count towards this limit, so the number of workers running at a given time may exceed this maximum. A smaller number of workers may run if the number of executing tasks is smaller than the supplied value of the workers argument.
seconds_interval	Number of seconds between polling intervals waiting for certain internal synchronous operations to complete. In certain cases, exponential backoff is used with this argument passed to seconds_max in a crew_throttle() object.
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` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `reset_globals` option of `crew_controller()` instead.

`reset_packages` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `reset_packages` option of `crew_controller()` instead.

`reset_options` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `reset_options` option of `crew_controller()` instead.

`garbage_collection` Deprecated on 2025-05-30 (`crew` version 1.1.2.9004). Please use the `garbage_collection` option of `crew_controller()` instead.

`crashes_error` Deprecated on 2025-01-13 (`crew` version 0.10.2.9002).

`launch_max` Deprecated on 2024-11-04 (`crew` version 0.10.2.9002).

`tls` A TLS configuration object from `crew_tls()`.

`r_arguments` Optional character vector of command line arguments to pass to `Rscript` (non-Windows) or `Rscript.exe` (Windows) when starting a worker. Example: `r_arguments = c("--vanilla", "--max-connections=32")`.

`options_metrics` Either `NULL` to opt out of resource metric logging for workers, or an object from `crew_options_metrics()` to enable and configure resource metric logging for workers. For resource logging to run, the `autometric` R package version 0.1.0 or higher must be installed.

`options_local` An object generated by `crew_options_local()` with options specific to the local controller.

`local_log_directory` Deprecated on 2024-10-08. Use `options_local` instead.

`local_log_join` Deprecated on 2024-10-08. Use `options_local` instead.

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$start(url = client$url, profile = client$profile)
launcher$launch()
task <- mirai::mirai("result", .compute = client$profile)
```

```
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_options_local` *Local* `crew` *launcher options.*

Description

Options for the local `crew` launcher.

Usage

```
crew_options_local(log_directory = NULL, log_join = TRUE)
```

Arguments

`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).
`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.

Value

A classed list of options for the local launcher.

See Also

Other options: `crew_options_metrics()`

Examples

```
crew_options_local()
```

`crew_options_metrics` *Options for logging resource usage metrics.*

Description

`crew_options_metrics()` configures the `crew` to record resource usage metrics (such as CPU and memory usage) for each running worker. To be activate resource usage logging, the `autometric` R package version 0.1.0 or higher must be installed.

Logging happens in the background (through a detached POSIX) so as not to disrupt the R session. On Unix-like systems, `crew_options_metrics()` can specify `/dev/stdout` or `/dev/stderr` as the log files, which will redirect output to existing logs you are already using. `autometric::log_read()` and `autometric::log_plot()` can read and visualize resource usage data from multiple log files, even if those files are mixed with other messages.

Usage

```
crew_options_metrics(path = NULL, seconds_interval = 5)
```

Arguments

`path`	Where to write resource metric log entries for workers. `path = NULL` disables logging. `path` equal to `"/dev/stdout"` (or `"/dev/stderr"`) sends log messages to the standard output (or standard error) streams, which is recommended on Unix-like systems because then output will go to the existing log files already configured for the controller, e.g. through `crew_options_local()` in the case of `crew_controller_local()`. If `path` is not `NULL`, `"/dev/stdout"`, or `"/dev/stderr"`, it should be a directory path, in which case each worker instance will write to a new file in that directory. After running enough tasks in `crew`, you can call `autometric::log_read(path)` to read all the data from all the log files in the files or directories at `path`, even if the logs files are mixed with other kinds of messages. Pass that data into `autometric::log_plot()` to visualize it.
`seconds_interval`	Positive number, seconds between resource metric log entries written to `path`.

Value

A classed list of options for logging resource usage metrics.

See Also

Other options: crew_options_local()

Examples

```
crew_options_metrics()
```

---

crew_queue *Create a* crew *queue object.*

Description

Create an R6 crew queue object.

Usage

```
crew_queue(data = character(0L), step = 1000L)
```

Arguments

data	Character vector of initial queue data.
step	Positive integer with the number of elements to extend the queue on each call to the extend() method.

Details

A crew queue is a classical first-in-first-out data structure that extends itself in chunks (of size step) to avoid overhead. crew uses queues to efficiently track the names of resolved tasks and backlogged tasks.

Value

A queue object.

See Also

Other queue: crew_class_queue

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(throttle = crew_throttle())
```

Arguments

throttle	A crew_throttle() object.

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(),
  assertions = TRUE
)
```

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.
assertions	TRUE to run assertions to check if arguments are valid, FALSE otherwise. TRUE is recommended for users.

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_max = 1,
  seconds_min = 0.001,
  seconds_start = seconds_min,
  base = 2
)
```

Arguments

seconds_max	Positive numeric scalar, maximum throttling interval
seconds_min	Positive numeric scalar, minimum throttling interval.
seconds_start	Positive numeric scalar, the initial wait time interval in seconds. The default is min because there is almost always auto-scaling to be done when the controller is created. reset() always sets the current wait interval back to seconds_start.
base	Numeric scalar greater than 1, base of the exponential backoff algorithm. increment() multiplies the waiting interval by base and decrement() divides the waiting interval by base. The default base is 2, which specifies a binary exponential backoff algorithm.

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 does not induce superfluous overhead. The throttle uses deterministic exponential backoff algorithm (https://en.wikipedia.org/wiki/Exponential_backoff) which increases wait times when there is nothing to do and decreases wait times when there is something to do. The controller decreases or increases the wait time with methods accelerate() and decelerate() in the throttle object, respectively, by dividing or multiplying by base (but keeping the wait time between seconds_min and seconds_max). In practice, crew calls reset() instead of update() in order to respond quicker to surges of activity (see the update() method).

Value

An R6 object with throttle configuration settings and methods.

See Also

Other throttle: crew_class_throttle

Examples

```
throttle <- crew_throttle(seconds_max = 1)
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,
  options_metrics = crew::crew_options_metrics()
)
```

Arguments

`settings`	Named list of arguments to `mirai::daemon()`.
`launcher`	Character string, name of the launcher
`worker`	Character of length 1 to uniquely identify the current worker.
`options_metrics`	Either `NULL` to opt out of resource metric logging for workers, or an object from `crew_options_metrics()` to enable and configure resource metric logging for workers. For resource logging to run, the `autometric` R package version 0.1.0 or higher must be installed.

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