Package: rrq (via r-universe)

June 29, 2024

Title Simple Redis Queue

Version 0.7.15

Description Simple Redis queue in R.

License MIT + file LICENSE

LazyData true

URL https://github.com/mrc-ide/rrq

BugReports https://github.com/mrc-ide/rrq/issues

Imports R6, callr (>= 3.7.0), cli, docopt, ids, logwatch (>= 0.1.1), progress, redux (>= 1.0.0), rlang

Suggests knitr, markdown, mockery, processx, rmarkdown, testthat, withr

RoxygenNote 7.3.1

Encoding UTF-8

Roxygen list(markdown = TRUE, r6 = TRUE)

Language en-GB

Config/testthat/edition 3

VignetteBuilder knitr

Remotes reside-ic/logwatch

Repository https://mrc-ide.r-universe.dev

RemoteUrl https://github.com/mrc-ide/rrq

RemoteRef master

RemoteSha 7d3d32153e6b43a0768b05b9505067494de73d24

Contents

`object_store` *rrq object store*

Description

When you create a task with rrq and that task uses local variables these need to be copied over to the worker that will evaluate the task. So, if we had

```
rrq_task_create_expr(f(a, b))
```

that would be the objects `a` and `b` from the context where `rrq_task_create_expr` was called. There are a few considerations here:

- The names `a` and `b` are only useful in the immediate context of the controller at the point the task is sent and so we need to store the *values* referenced by `a` and `b` without reference to the names - we do this by naming the new values after their value. That is, the name becomes the hash of the object, computed by `rlang::hash()`, as a form of content-addressable storage.
- When doing this we note that we might end up using the value referenced by `a` or `b` many times in different tasks so we should not re-save the data more than needed, and we should not necessarily delete it when a task is deleted unless nothing else uses that value.
- The objects might tiny or could be large; if small we tend to care about how quickly they can be resolved (i.e., latency) and if large we need to be careful not to overfull Redis' database as it's a memory-based system.

To make this robust and flexible, we use a `object_store` object, which will allow objects to be stored either directly in Redis, or offloaded onto some "large" data store based on their size. Currently, we provide support only for offloading to disk, but in future hope to expand this.

When we create a value in the store (or reference a value that already exists) we assign a tag into the database; this means that we have for a value with hash `abc123` and tag `def789`

- `prefix:data["abc123"] => [1] f5 26 a5 b7 26 93 b3 41 b7 d0 b0...` (the data stored, serialised into a redis hash by its hash, as a binary object.
- `prefix:tag_hash:def789 => {abc123}` (a set of hashes used by our tag)
- `prefix:hash_tag:abc123 => {def789}` (a set of tags that reference our hash)

If we also used the value with hash `abc123` with tag `fed987` this would look like

- `prefix:data[abc123] => [1] f5 26 a5 b7 26 93 b3 41 b7 d0 b0...` hash, as a binary object.
- `prefix:tag_hash:def789 => {abc123}`
- `prefix:tag_hash:fed987 => {abc123}`
- `prefix:hash_tag:abc123 => {def789, fed987}`

As tags are dropped, then the references are dropped from the set `prefix:hash_tag:abc123` and when that set becomes empty then we can delete `prefix:data[abc123]` as simple form of reference counting.

For `rrq` we will use `task_ids` as a tag.

For dealing with large data, we "offload" large data into a secondary store. This replaces the redis hash of `hash => value` with something else. Currently the only alternative we offer is `object_store_offload_disk` which will save the binary representation of the object at the path `<path>/<hash>` and will allow large values to be shared between controller and worker so long as they share a common filesystem.

Details

Create an object store. Typically this is not used by end-users, and is used internally by rrq_controller

Methods

Public methods:

- `object_store$new()`
- `object_store$list()`
- `object_store$tags()`
- `object_store$get()`
- `object_store$mget()`
- `object_store$set()`
- `object_store$mset()`
- `object_store$location()`
- `object_store$drop()`
- `object_store$destroy()`

Method `new()`**:** Create a new object store (or connect to an existing one)

Usage:

```
object_store$new(con, prefix, max_size = Inf, offload = NULL)
```

Arguments:

`con` A redis connection object

`prefix` A key prefix to use; we will make a number of keys that start with this prefix.

`max_size` The maximum serialised object size, in bytes. If the serialised object is larger than this size it will be placed into the offload storage, as provided by the `offload` argument. By default this is `Inf` so all values will be stored in the redis database.

`offload` An offload storage object. We provide one of these `object_store_offload_disk`, which saves objects to on disk after serialisation). This interface is subject to change. If not given but an object exceeds `max_size` an error will be thrown.

Method `list()`**:** List all hashes of data known to this data store

Usage:

```
object_store$list()
```

Method `tags()`**:** List all tags known to this data store

Usage:

```
object_store$tags()
```

Method `get()`**:** Get a single object by its hash

Usage:

```
object_store$get(hash)
```

Arguments:

`hash` a single hash to use

Method `mget()`**:** Get a number objects by their hashes. Unlike `$get()` this method accepts a vector of hash (length 0, 1, or more than 1) and returns a list of the same length.

Usage:

```
object_store$mget(hash)
```

Arguments:

`hash` A vector of object hashes

Method `set()`**:** Set an object into the object store, returning the hash of that object.

Usage:

```
object_store$set(value, tag, serialize = TRUE)
```

Arguments:

`value` The object to save

`tag` A string used to associate with the object. When all tags that point to a particular object value have been removed, then the object will be deleted from the store.

`serialize` Logical, indicating if the values should be serialised first. Typically this should be `TRUE`, but for advanced use if you already have a serialised object you can pass that in and set to `FALSE`. Note that only objects serialised with `redux::object_to_bin` (or with `serialize(..., xdr = FALSE)`) will be accepted.

Method `mset()`**:** Set a number of objects into the store. Unlike `$set()`, this method sets a list of objects into the store at once, and returns a character vector of hashes the same length as the list of values.

Usage:

```
object_store$mset(value, tag, serialize = TRUE)
```

Arguments:

`value` A list of objects to save

`tag` A string used to associate with the object. When all tags that point to a particular object value have been removed, then the object will be deleted from the store. The same tag is used for all objects.

`serialize` Logical, indicating if the values should be serialised first. Typically this should be `TRUE`, but for advanced use if you already have a serialised object you can pass that in and set to `FALSE`. Note that only objects serialised with `redux::object_to_bin` (or with `serialize(..., xdr = FALSE)`) will be accepted.

Method `location()`**:** Return the storage locations of a set of hashes. Currently the location may be `redis` (stored directly in the redis server), `offload` (stored in the offload storage) or `NA` (if not found, and if `error = FALSE`).

Usage:

```
object_store$location(hash, error = TRUE)
```

Arguments:

`hash` A vector of hashes

`error` A logical, indicating if we should throw an error if a hash is unknown

Method `drop()`**:** Delete tags from the store. This will dissociate the tags from any hashes they references and if that means that no tag points to a hash then the data at that hash will be removed. We return (invisibly) a character vector of any dropped hashes.

Usage:

```
object_store$drop(tag)
```

Arguments:

`tag` Vector of tags to drop

Method `destroy()`**:** Remove all data from the store, and all the stores metadata

Usage:

```
object_store$destroy()
```

`object_store_offload_disk` *Disk-based offload*

Description

Disk-based offload

Disk-based offload

Details

A disk-based offload for `object_store`. This is not intended at all for direct user-use.

Methods

Public methods:

- `object_store_offload_disk$new()`
- `object_store_offload_disk$mset()`
- `object_store_offload_disk$mget()`
- `object_store_offload_disk$mdel()`
- `object_store_offload_disk$list()`
- `object_store_offload_disk$destroy()`

Method `new()`**:** Create the store

Usage:

```
object_store_offload_disk$new(path)
```

Arguments:

`path` A directory name to store objects in. It will be created if it does not yet exist.

Method `mset()`**:** Save a number of values to disk

Usage:

```
object_store_offload_disk$mset(hash, value)
```

Arguments:

`hash` A character vector of object hashes

`value` A list of serialised objects (each of which is a raw vector)

Method `mget()`**:** Retrieve a number of objects from the store

Usage:

```
object_store_offload_disk$mget(hash)
```

Arguments:

`hash` A character vector of hashes of the objects to return. The objects will be deserialised before return.

Method `mdel()`**:** Delete a number of objects from the store

Usage:

```
object_store_offload_disk$mdel(hash)
```

Arguments:

`hash` A character vector of hashes to remove

Method `list()`**:** List hashes stored in this offload store

Usage:

```
object_store_offload_disk$list()
```

Method `destroy()`**:** Completely delete the store (by deleting the directory)

Usage:

```
object_store_offload_disk$destroy()
```

`rrq_configure` *Configure rrq*

Description

Configure rrq options. This function must be called before either a controller or worker connects to a queue, as the options will apply to both. The function may only be called once on a given queue as there is no facility (yet) to update options. Currently the options concern only storage, and specifically how larger objects will be saved (using `object_store`.

Usage

```
rrq_configure(
  queue_id,
  con = redux::hiredis(),
  ...,
  store_max_size = Inf,
  offload_path = NULL
)
```

Arguments

`queue_id`	The queue id; the same as you would pass to rrq_controller
`con`	A redis connection
`...`	Additional arguments - this must be empty. This argument exists so that all additional arguments must be passed by name.
`store_max_size`	The maximum object size, in bytes, before being moved to the offload store. If given, then larger data will be saved in `offload_path` (using `object_store_offload_disk`)
`offload_path`	The path to create an offload store at (passed to `object_store_offload_disk`). The directory will be created if it does not exist. If not given (or `NULL`) but `store_max_size` is finite, then trying to save large objects will throw an error.

Value

Invisibly, a list with processed configuration information

Storage

Every time that a task is saved, or a task is completed, results are saved into the Redis database. Because Redis is an in-memory database, it's not a great idea to save very large objects into it (if you ran 100 jobs in parallel and each saved a 2GB object you'd likely take down your redis server). In addition, `redux` does not support directly saving objects larger than `2^31 - 1` bytes into Redis. So, for some use cases we need to consider where to store larger objects.

The strategy here is to "offload" the larger objects - bigger than some user-given size - onto some other storage system. Currently the only alternative supported is a disk store (`object_store_offload_disk`)

but we hope to expand this later. So if your task returns a 3GB object then we will spill that to disk rather than failing to save that into Redis.

How big is an object? We serialise the object (`redux::object_to_bin` just wraps `serialize`) which creates a vector of bytes and that is saved into the database. To get an idea of how large things are you can do: `length(redux::object_to_bin(your_object))`. At the time this documentation was written, `mtcars` was `3807` bytes, and a million random numbers was `8,000,031` bytes. It's unlikely that a `store_max_size` of less than 1MB will be sensible.

`rrq_controller` *Create rrq controller*

Description

Create a new controller. This is the new interface that will replace rrq_controller soon, at which point it will rename back to `rrq_controller`.

Usage

```
rrq_controller(
  queue_id,
  con = redux::hiredis(),
  timeout_task_wait = NULL,
  follow = NULL,
  check_version = TRUE
)
```

Arguments

`queue_id`	An identifier for the queue. This will prefix all keys in redis, so a prefix might be useful here depending on your use case (e.g. `rrq:<user>:<id>`)
`con`	A redis connection. The default tries to create a redis connection using default ports, or environment variables set as in `redux::hiredis()`
`timeout_task_wait`	An optional default timeout to use when waiting for tasks with rrq_task_wait. If not given, then we fall back on the global option `rrq.timeout_task_wait`, and if that is not set, we wait forever (i.e., `timeout_task_wait = Inf`).
`follow`	An optional default logical to use for tasks that may (or may not) be retried. If not given we fall back on the global option `rrq.follow`, and if that is not set then `TRUE` (i.e., we do follow). The value `follow = TRUE` is potentially slower than `follow = FALSE` for some operations because we need to dereference every task id. If you never use rrq_task_retry then this dereference never has an effect and we can skip it. See `vignette("fault-tolerance")` for more information.
`check_version`	Logical, indicating if we should check the schema version. You can pass `FALSE` here to continue even where the schema version is incompatible, though any subsequent actions may lead to corruption.

Value

An `rrq_controller` object, which is opaque.

Task lifecycle

- A task is queued with `$enqueue()`, at which point it becomes `PENDING`
- Once a worker selects the task to run, it becomes `RUNNING`
- If the task completes successfully without error it becomes `COMPLETE`
- If the task throws an error, it becomes `ERROR`
- If the task was cancelled (e.g., via `$task_cancel()`) it becomes `CANCELLED`
- If the task is killed by an external process, crashes or the worker dies (and is running a heartbeat) then the task becomes `DIED`.
- The status of an unknown task is `MISSING`
- Tasks in any terminal state (except `IMPOSSIBLE`) may be retried with `task_retry` at which point they become `MOVED`, see `vignette("fault-tolerance")` for details

Worker lifecycle

- A worker appears and is `IDLE`
- When running a task it is `BUSY`
- If it receives a `PAUSE` message it becomes `PAUSED` until it receives a `RESUME` message
- If it exits cleanly (e.g., via a `STOP` message or a timeout) it becomes `EXITED`
- If it crashes and was running a heartbeat, it becomes `LOST`

Messages

Most of the time workers process tasks, but you can also send them "messages". Messages take priority over tasks, so if a worker becomes idle (by coming online or by finishing a task) it will consume all available messages before starting on a new task, even if both are available.

Each message has a "command" and may have "arguments" to that command. The supported messages are:

- `PING` (no args): "ping" the worker, if alive it will respond with "PONG"
- `ECHO` (accepts an argument of a string): Print a string to the terminal and log of the worker. Will respond with `OK` once the message has been printed.
- `EVAL` (accepts a string or a quoted expression): Evaluate an arbitrary R expression on the worker. Responds with the value of this expression.
- `STOP` (accepts a string to print as the worker exits, defaults to "BYE"): Tells the worker to stop.
- `INFO` (no args): Returns information about the worker (versions of packages, hostname, pid, etc).
- `PAUSE` (no args): Tells the worker to stop accepting tasks (until it receives a `RESUME` message). Messages are processed as normal.

- RESUME (no args): Tells a paused worker to resume accepting tasks.
- REFRESH (no args): Tells the worker to rebuild their environment with the create method.
- TIMEOUT_SET (accepts a number, representing seconds): Updates the worker timeout - the length of time after which it will exit if it has not processed a task.
- TIMEOUT_GET (no args): Tells the worker to respond with its current timeout.

`rrq_default_controller_set` *Register default controller*

Description

Set or clear a default controller for use with rrq functions. You will want to use this to avoid passing `controller` in as a named argument to every function.

Usage

```
rrq_default_controller_set(controller)

rrq_default_controller_clear()
```

Arguments

`controller`	An rrq_controller object

`rrq_deferred_list` *List deferred tasks*

Description

Return deferred tasks and what they are waiting on. Note this is in an arbitrary order, tasks will be added to the queue as their dependencies are satisfied.

Usage

```
rrq_deferred_list(controller = NULL)
```

Arguments

`controller`	The controller to use. If not given (or NULL) we'll use the controller registered with `rrq_default_controller_set()`.

rrq_destroy *Destroy queue*

Description

Entirely destroy a queue, by deleting all keys associated with it from the Redis database. This is a very destructive action and cannot be undone.

Usage

```
rrq_destroy(
  delete = TRUE,
  worker_stop_type = "message",
  timeout_worker_stop = 0,
  controller = NULL
)
```

Arguments

delete	Either TRUE (the default) indicating that the keys should be immediately deleted. Alternatively, provide an integer value and the keys will instead be marked for future deletion by "expiring" after this many seconds, using Redis' EXPIRE command.
worker_stop_type	Passed to rrq_worker_stop(); Can be one of "message", "kill" or "kill_local". The "kill" method requires that the workers are using a heartbeat, and "kill_local" requires that the workers are on the same machine as the controller. However, these may be faster to stop workers than "message", which will wait until any task is finished.
timeout_worker_stop	A timeout to pass to the worker to respond the request to stop. See worker_stop's timeout argument for details.
controller	The controller to destroy

rrq_envir *Create simple worker environments*

Description

Helper function for creating a worker environment. This function exists to create a function suitable for passing to rrq_worker_envir_set for the common case where the worker should source some R scripts and/or load some packages on startup. This is a convenience wrapper around defining your own function, covering the most simple case. If you need more flexibility you should write your own function.

Usage

```
rrq_envir(packages = NULL, sources = NULL)
```

Arguments

packages	An optional character vector of
sources	An optional character vector of scripts to read. Typically these will contain just function definitions but you might read large data objects here too.

Value

A function suitable for passing to rrq_worker_envir_set, which can set (or update) the environment for your workers.

---

rrq_heartbeat *Create a heartbeat instance*

Description

Create a heartbeat instance

Create a heartbeat instance

Details

Create a heartbeat instance. This can be used by running `obj$start()` which will reset the TTL (Time To Live) on `key` every `period` seconds (don't set this too high). If the R process dies, then the key will expire after `3 * period` seconds (or set `expire`) and another application can tell that this R instance has died.

Methods

Public methods:

- `rrq_heartbeat$new()`
- `rrq_heartbeat$is_running()`
- `rrq_heartbeat$start()`
- `rrq_heartbeat$stop()`
- `rrq_heartbeat$format()`

Method `new()`**:** Create a heartbeat object

Usage:

```
rrq_heartbeat$new(
  key,
  period,
  expire = 3 * period,
  value = expire,
```

```
  config = NULL,
  start = TRUE,
  timeout = 10
)
```

Arguments:

`key` Key to use. Once the heartbeat starts it will create this key and set it to expire after `expiry` seconds.

`period` Timeout period (in seconds)

`expire` Key expiry time (in seconds)

`value` Value to store in the key. By default it stores the expiry time, so the time since last heartbeat can be computed. This will be converted to character with `as.character` before saving into Redis

`config` Configuration parameters passed through to `redux::redis_config`. Provide as either a named list or a `redis_config` object. This allows host, port, password, db, etc all to be set.

`start` Should the heartbeat be started immediately?

`timeout` Time, in seconds, to wait for the heartbeat to appear. It should generally appear very quickly (within a second unless your connection is very slow) so this can be generally left alone.

Method `is_running()`**:** Report if heartbeat process is running. This will be `TRUE` if the process has been started and has not stopped.

Usage:

```
rrq_heartbeat$is_running()
```

Method `start()`**:** Start the heartbeat process. An error will be thrown if it is already running.

Usage:

```
rrq_heartbeat$start()
```

Method `stop()`**:** Stop the heartbeat process

Usage:

```
rrq_heartbeat$stop(wait = TRUE)
```

Arguments:

`wait` Logical, indicating if we should wait until the heartbeat process terminates (should take only a fraction of a second)

Method `format()`**:** Format method, used by R6 to nicely print the object

Usage:

```
rrq_heartbeat$format(...)
```

Arguments:

`...` Additional arguments, currently ignored

Examples

```
if (redux::redis_available()) {
  rand_str <- function() {
    paste(sample(letters, 20, TRUE), collapse = "")
  }
  key <- sprintf("rrq:heartbeat:%s", rand_str())
  h <- rrq::rrq_heartbeat$new(key, 1, expire = 2)
  con <- redux::hiredis()

  # The heartbeat key exists
  con$EXISTS(key)

  # And has an expiry of less than 2000ms
  con$PTTL(key)

  # We can manually stop the heartbeat, and 2s later the key will
  # stop existing
  h$stop()

  Sys.sleep(2)
  con$EXISTS(key) # 0

  # This is required to close any processes opened by this
  # example, normally you would not need this.
  processx:::supervisor_kill()
}
```

`rrq_heartbeat_kill` *Kill a process running a heartbeat*

Description

Send a kill signal (typically `SIGTERM`) to terminate a process that is running a heartbeat. This is used by `rrq_controller` in order to tear down workers, even if they are processing a task. When a heartbeat process is created, in its main loop it will listen for requests to kill via this function and will forward them to the worker. This is primarily useful where workers are on a different physical machine to the controller where `tools::pskill()` cannot be used.

Usage

```
rrq_heartbeat_kill(con, key, signal = tools::SIGTERM)
```

Arguments

`con`	A hiredis object
`key`	The heartbeat key
`signal`	A signal to send (typically `tools::SIGTERM` for a "polite" shutdown)

Examples

```
if (redux::redis_available()) {
  rand_str <- function() {
    paste(sample(letters, 20, TRUE), collapse = "")
  }
  # Suppose we have a process that exposes a heartbeat running on
  # this key:
  key <- sprintf("rrq:heartbeat:%s", rand_str())

  # We can send it a SIGTERM signal over redis using:
  con <- redux::hiredis()
  rrq::rrq_heartbeat_kill(con, key, tools::SIGTERM)
}
```

`rrq_message_get_response` *Get message response*

Description

Get response to messages, waiting until the message has been responded to.

Usage

```
rrq_message_get_response(
  message_id,
  worker_ids = NULL,
  named = TRUE,
  delete = FALSE,
  timeout = 0,
  time_poll = 0.5,
  progress = NULL,
  controller = NULL
)
```

Arguments

`message_id`	The message id
`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers are used (note that this may differ to the set of workers that the message was sent to!)
`named`	Logical, indicating if the return value should be named by worker id.
`delete`	Logical, indicating if messages should be deleted after retrieval
`timeout`	Integer, representing seconds to wait until the response has been received. An error will be thrown if a response has not been received in this time.
`time_poll`	If `timeout` is greater than zero, this is the polling interval used between redis calls. Increasing this reduces network load but increases the time that may be waited for.

progress	Optional logical indicating if a progress bar should be displayed. If NULL we fall back on the value of the global option rrq.progress, and if that is unset display a progress bar if in an interactive session.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

---

rrq_message_has_response *Detect if message has response*

Description

Detect if a response is available for a message

Usage

```
rrq_message_has_response(
  message_id,
  worker_ids = NULL,
  named = TRUE,
  controller = NULL
)
```

Arguments

message_id	The message id
worker_ids	Optional vector of worker ids. If NULL then all active workers are used (note that this may differ to the set of workers that the message was sent to!)
named	Logical, indicating if the return vector should be named
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A logical vector, possibly named (depending on the named argument)

rrq_message_response_ids *Return ids for messages with responses for a particular worker.*

Description

Return ids for messages with responses for a particular worker.

Usage

```
rrq_message_response_ids(worker_id, controller = NULL)
```

Arguments

worker_id	The worker id
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A character vector of ids

rrq_message_send *Send message to workers*

Description

Send a message to workers. Sending a message returns a message id, which can be used to poll for a response with the other rrq_message_* functions.

Usage

```
rrq_message_send(command, args = NULL, worker_ids = NULL, controller = NULL)
```

Arguments

command	A command, such as PING, PAUSE; see the Messages section of the Details for al messages.
args	Arguments to the command, if supported
worker_ids	Optional vector of worker ids to send the message to. If NULL then the message will be sent to all active workers.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

Invisibly, a single identifier

`rrq_message_send_and_wait` *Send a message and wait for response*

Description

Send a message and wait for responses. This is a helper function around `rrq_message_send()` and `rrq_message_get_response()`.

Usage

```
rrq_message_send_and_wait(
  command,
  args = NULL,
  worker_ids = NULL,
  named = TRUE,
  delete = TRUE,
  timeout = 600,
  time_poll = 0.05,
  progress = NULL,
  controller = NULL
)
```

Arguments

`command`	A command, such as `PING`, `PAUSE`; see the Messages section of the Details for al messages.
`args`	Arguments to the command, if supported
`worker_ids`	Optional vector of worker ids to send the message to. If `NULL` then the message will be sent to all active workers.
`named`	Logical, indicating if the return value should be named by worker id.
`delete`	Logical, indicating if messages should be deleted after retrieval
`timeout`	Integer, representing seconds to wait until the response has been received. An error will be thrown if a response has not been received in this time.
`time_poll`	If `timeout` is greater than zero, this is the polling interval used between redis calls. Increasing this reduces network load but increases the time that may be waited for.
`progress`	Optional logical indicating if a progress bar should be displayed. If `NULL` we fall back on the value of the global option `rrq.progress`, and if that is unset display a progress bar if in an interactive session.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

The message response

rrq_queue_length *Queue length*

Description

Returns the length of the queue (the number of tasks waiting to run). This is the same as the length of the value returned by rrq_queue_list.

Usage

```
rrq_queue_length(queue = NULL, controller = NULL)
```

Arguments

queue	The name of the queue to query (defaults to the "default" queue).
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A number

rrq_queue_list *List queue contents*

Description

Returns the keys in the task queue.

Usage

```
rrq_queue_list(queue = NULL, controller = NULL)
```

Arguments

queue	The name of the queue to query (defaults to the "default" queue).
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

rrq_queue_remove *Remove task ids from a queue*

Description

Remove task ids from a queue.

Usage

```
rrq_queue_remove(task_ids, queue = NULL, controller = NULL)
```

Arguments

task_ids	Task ids to remove
queue	The name of the queue to query (defaults to the "default" queue).
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

rrq_task_cancel *Cancel a task*

Description

Cancel a single task. If the task is PENDING it will be unqueued and the status set to CANCELED. If RUNNING then the task will be stopped if it was set to run in a separate process (i.e., queued with separate_process = TRUE). Dependent tasks will be marked as impossible.

Usage

```
rrq_task_cancel(task_id, wait = TRUE, timeout_wait = 10, controller = NULL)
```

Arguments

task_id	Id of the task to cancel
wait	Wait for the task to be stopped, if it was running.
timeout_wait	Maximum time, in seconds, to wait for the task to be cancelled by the worker.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

Nothing if successfully cancelled, otherwise throws an error with task_id and status e.g. Task 123 is not running (MISSING)

`rrq_task_create_bulk_call` *Create bulk tasks from a call*

Description

Create a bulk set of tasks based on applying a function over a vector or data.frame. This is the bulk equivalent of rrq_task_create_call, in the same way that rrq_task_create_bulk_expr is a bulk version of rrq_task_create_expr.

Usage

```
rrq_task_create_bulk_call(
  fn,
  data,
  args = NULL,
  queue = NULL,
  separate_process = FALSE,
  timeout_task_run = NULL,
  depends_on = NULL,
  controller = NULL
)
```

Arguments

`fn`	The function to call
`data`	The data to apply the function over. This can be a vector or list, in which case we act like `lapply` and apply `fn` to each element in turn. Alternatively, this can be a data.frame, in which case each row is taken as a set of arguments to `fn`. Note that if `data` is a `data.frame` then all arguments to `fn` are named.
`args`	Additional arguments to `fn`, shared across all calls. These must be named. If you are using a `data.frame` for `data`, you'd probably be better off adding additional columns that don't vary across rows, but the end result is the same.
`queue`	The queue to add the task to; if not specified the "default" queue (which all workers listen to) will be used. If you have configured workers to listen to more than one queue you can specify that here. Be warned that if you push jobs onto a queue with no worker, it will queue forever.
`separate_process`	Logical, indicating if the task should be run in a separate process on the worker. If `TRUE`, then the worker runs the task in a separate process using the `callr` package. This means that the worker environment is completely clean, subsequent runs are not affected by preceding ones. The downside of this approach is a considerable overhead in starting the external process and transferring data back.

`timeout_task_run`
: Optionally, a maximum allowed running time, in seconds. This parameter only has an effect if `separate_process` is `TRUE`. If given, then if the task takes longer than this time it will be stopped and the task status set to `TIMEOUT`.

`depends_on`
: Vector or list of IDs of tasks which must have completed before this job can be run. Once all dependent tasks have been successfully run, this task will get added to the queue. If the dependent task fails then this task will be removed from the queue.

`controller`
: The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A vector of task identfiers; this will have the length as `data` has rows if it is a `data.frame`, otherwise it has the same length as `data`

`rrq_task_create_bulk_expr`
Create bulk tasks from an expression

Description

Create a bulk set of tasks. Variables in `data` take precedence over variables in the environment in which `expr` was created. There is no "pronoun" support yet (see rlang docs). Use `!!` to pull a variable from the environment if you need to, but be careful not to inject something really large (e.g., any vector really) or you'll end up with a revolting expression and poor backtraces.

Usage

```
rrq_task_create_bulk_expr(
  expr,
  data,
  queue = NULL,
  separate_process = FALSE,
  timeout_task_run = NULL,
  depends_on = NULL,
  controller = NULL
)
```

Arguments

`expr`
: An expression, as for rrq_task_create_expr

`data`
: Data that you wish to inject *row-wise* into the expression

`queue`
: The queue to add the task to; if not specified the "default" queue (which all workers listen to) will be used. If you have configured workers to listen to more than one queue you can specify that here. Be warned that if you push jobs onto a queue with no worker, it will queue forever.

separate_process
: Logical, indicating if the task should be run in a separate process on the worker. If TRUE, then the worker runs the task in a separate process using the callr package. This means that the worker environment is completely clean, subsequent runs are not affected by preceding ones. The downside of this approach is a considerable overhead in starting the external process and transferring data back.

timeout_task_run
: Optionally, a maximum allowed running time, in seconds. This parameter only has an effect if separate_process is TRUE. If given, then if the task takes longer than this time it will be stopped and the task status set to TIMEOUT.

depends_on
: Vector or list of IDs of tasks which must have completed before this job can be run. Once all dependent tasks have been successfully run, this task will get added to the queue. If the dependent task fails then this task will be removed from the queue.

controller
: The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A character vector with task identifiers; this will have a length equal to the number of row in data

---

`rrq_task_create_call` *Create a task from a call*

Description

Create a task based on a function call. This is fairly similar to callr::r, and forms the basis of lapply()-like task submission. Sending a call may have slightly different semantics than you expect if you send a closure (a function that binds data), and we may change behaviour here until we find a happy set of compromises. See Details for more on this. The expression rrq_task_create_call(f, list(a, b, c)) is similar to rrq_task_create_expr(f(a, b, c)), use whichever you prefer.

Usage

```
rrq_task_create_call(
  fn,
  args,
  queue = NULL,
  separate_process = FALSE,
  timeout_task_run = NULL,
  depends_on = NULL,
  controller = NULL
)
```

Arguments

fn	The function to call
args	A list of arguments to pass to the function
queue	The queue to add the task to; if not specified the "default" queue (which all workers listen to) will be used. If you have configured workers to listen to more than one queue you can specify that here. Be warned that if you push jobs onto a queue with no worker, it will queue forever.
separate_process	Logical, indicating if the task should be run in a separate process on the worker. If TRUE, then the worker runs the task in a separate process using the callr package. This means that the worker environment is completely clean, subsequent runs are not affected by preceding ones. The downside of this approach is a considerable overhead in starting the external process and transferring data back.
timeout_task_run	Optionally, a maximum allowed running time, in seconds. This parameter only has an effect if separate_process is TRUE. If given, then if the task takes longer than this time it will be stopped and the task status set to TIMEOUT.
depends_on	Vector or list of IDs of tasks which must have completed before this job can be run. Once all dependent tasks have been successfully run, this task will get added to the queue. If the dependent task fails then this task will be removed from the queue.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Details

Things are pretty unambiguous when you pass in a function from a package, especially when you refer to that package with its namespace (e.g. pkg::fn).

If you pass in the name *without a namespace* from a package that you have loaded with library() locally but you have not loaded with library within your worker environment, we may not do the right thing and you may see your task fail, or find a different function with the same name.

If you pass in an anonymous function (e.g., function(x) x + 1) we may or may not do the right thing with respect to environment capture. We never capture the global environment so if your function is a closure that tries to bind a symbol from the global environment it will not work. Like with callr::r, anonymous functions will be easiest to think about where they are fully self contained (i.e., all inputs to the functions come through args). If you have bound a *local* environment, we may do slightly better, but semantics here are undefined and subject to change.

R does some fancy things with function calls that we don't try to replicate. In particular you may have noticed that this works:

```
c <- "x"
c(c, c) # a vector of two "x"'s
```

You can end up in this situation locally with:

```
f <- function(x) x + 1
local({
  f <- 1
  f(f) # 2
})
```

this is because when R looks for the symbol for the call it skips over non-function objects. We don't reconstruct environment chains in exactly the same way as you would have locally so this is not possible.

Value

A task identifier (a 32 character hex string) that you can pass in to other rrq functions, notably `rrq_task_status()` and `rrq_task_result()`

`rrq_task_create_expr` *Create a task based on an expression*

Description

Create a task based on an expression. The expression passed as `expr` will typically be a function call (e.g., `f(x)`). We will analyse the expression and find all variables that you reference (in the case of `f(x)` this is `x`) and combine this with the function name to run on the worker. If `x` cannot be found in your calling environment we will error.

Usage

```
rrq_task_create_expr(
  expr,
  queue = NULL,
  separate_process = FALSE,
  timeout_task_run = NULL,
  depends_on = NULL,
  controller = NULL
)
```

Arguments

`expr`	The expression, does not need quoting. See Details.
`queue`	The queue to add the task to; if not specified the "default" queue (which all workers listen to) will be used. If you have configured workers to listen to more than one queue you can specify that here. Be warned that if you push jobs onto a queue with no worker, it will queue forever.
`separate_process`	Logical, indicating if the task should be run in a separate process on the worker. If `TRUE`, then the worker runs the task in a separate process using the `callr`

package. This means that the worker environment is completely clean, subsequent runs are not affected by preceding ones. The downside of this approach is a considerable overhead in starting the external process and transferring data back.

`timeout_task_run`
: Optionally, a maximum allowed running time, in seconds. This parameter only has an effect if `separate_process` is `TRUE`. If given, then if the task takes longer than this time it will be stopped and the task status set to `TIMEOUT`.

`depends_on`
: Vector or list of IDs of tasks which must have completed before this job can be run. Once all dependent tasks have been successfully run, this task will get added to the queue. If the dependent task fails then this task will be removed from the queue.

`controller`
: The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Details

Alternatively you may provide a multiline statement by using `{}` to surround multiple lines, such as:

```
task_create_expr({
  x <- runif(1)
  f(x)
}, ...)
```

in this case, we apply a simple heuristic to work out that `x` is locally assigned and should not be saved with the expression.

`rrq_task_data` *Fetch internal task data*

Description

Fetch internal data about a task (expert use only)

Usage

```
rrq_task_data(task_id, controller = NULL)
```

Arguments

`task_id`
: A single task identifier

`controller`
: The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

Internal data, structures subject to change

rrq_task_delete *Delete tasks*

Description

Delete one or more tasks

Usage

```
rrq_task_delete(task_ids, check = TRUE, controller = NULL)
```

Arguments

task_ids	Vector of task ids to delete
check	Logical indicating if we should check that the tasks are not running. Deleting running tasks is unlikely to result in desirable behaviour.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

Nothing, called for side effects only

rrq_task_exists *Test if tasks exist*

Description

Test if task ids exist (i.e., are known to this controller). Nonexistant tasks may be deleted, known to a different controller or just never have existed.

Usage

```
rrq_task_exists(task_ids, named = FALSE, controller = NULL)
```

Arguments

task_ids	Vector of task ids to check
named	Logical, indicating if the return value should be named with the task ids; as these are quite long this can make the value a little awkward to work with.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A logical vector the same length as task_ids; TRUE where the task exists, FALSE otherwise.

rrq_task_info *Fetch task information*

Description

Fetch information about a task. This currently includes information about where a task is (or was) running and information about any retry chain, but will expand in future. The format of the output here is subject to change (and will probably get a nice print method) but the values present in the output will be included in any future update.

Usage

```
rrq_task_info(task_id, controller = NULL)
```

Arguments

task_id	A single task identifier
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A list, format currently subject to change

rrq_task_list *List all tasks*

Description

List all tasks. This may be a lot of tasks, and so can be quite slow to execute.

Usage

```
rrq_task_list(controller = NULL)
```

Arguments

controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A character vector

rrq_task_overview *High level task overciew*

Description

Provide a high level overview of task statuses for a set of task ids, being the count in major categories of PENDING, RUNNING, COMPLETE and ERROR.

Usage

```
rrq_task_overview(task_ids = NULL, controller = NULL)
```

Arguments

task_ids	Optional character vector of task ids for which you would like the overview. If not given (or NULL) then the status of all task ids known to this rrq controller is used (this might be fairly costly).
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A list with names corresponding to possible task status levels and values being the number of tasks in that state.

rrq_task_position *Find task position in queue*

Description

Find the position of one or more tasks in the queue.

Usage

```
rrq_task_position(
  task_ids,
  missing = 0L,
  queue = NULL,
  follow = NULL,
  controller = NULL
)
```

Arguments

task_ids	Character vector of tasks to find the position for.
missing	Value to return if the task is not found in the queue. A task will take value missing if it is running, complete, errored, deferred etc and a positive integer if it is in the queue, indicating its position (with 1) being the next task to run.
queue	The name of the queue to query (defaults to the "default" queue).
follow	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option rrq.follow, and finally TRUE. Set to FALSE if you want to return information about the original task, even if it has been subsequently retried.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

An integer vector, the same length as task_ids

---

rrq_task_preceeding *List tasks ahead of a task*

Description

List the tasks in front of task_id in the queue. If the task is missing from the queue this will return NULL. If the task is next in the queue this will return an empty character vector.

Usage

```
rrq_task_preceeding(task_id, queue = NULL, follow = NULL, controller = NULL)
```

Arguments

task_id	Task to find the position for.
queue	The name of the queue to query (defaults to the "default" queue).
follow	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option rrq.follow, and finally TRUE. Set to FALSE if you want to return information about the original task, even if it has been subsequently retried.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

rrq_task_progress *Fetch task progress information*

Description

Retrieve task progress, if set. This will be NULL if progress has never been registered, otherwise whatever value was set - can be an arbitrary R object.

Usage

```
rrq_task_progress(task_id, controller = NULL)
```

Arguments

task_id	A single task id for which the progress is wanted.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

Any set progress object

rrq_task_progress_update *Post task update*

Description

Post a task progress update. The progress system in rrq is agnostic about how you are going to render your progress, and so it just a convention - see Details below. Any R object can be sent as a progress value (e.g., a string, a list, etc).

Usage

```
rrq_task_progress_update(value, error = FALSE)
```

Arguments

value	An R object with the contents of the update. This will overwrite any previous progress value, and can be retrieved by calling rrq_task_progress. A value of NULL will appear to clear the status, as NULL will also be returned if no status is found for a task.
error	Logical, indicating if we should throw an error if not running as an rrq task. Set this to FALSE if you want code to work without modification within and outside of an rrq job, or to TRUE if you want to be sure that progress messages have made it to the server.

Details

In order to report on progress, a task may, in it's code, write

```
rrq::rrq_task_progress_update("task is 90% done")
```

and this information will be fetchable by calling rrq_task_progress with the `task_id`.

It is also possible to register progress *without* acquiring a dependency on `rrq`. If your package/script includes code like:

```
progress <- function(message) {
  signalCondition(structure(list(message = message),
                            class = c("progress", "condition")))
}
```

(this function can be called anything - the important bit is the body function body - you must return an object with a `message` element and the two class attributes `progress` and `condition`).

then you can use this in the same way as `rrq::rrq_task_progress_update` above in your code. When run without using rrq, this function will appear to do nothing.

`rrq_task_result` *Fetch single task result*

Description

Get the result for a single task (see rrq_task_results for a method for efficiently getting multiple results at once). Returns the value of running the task if it is complete, and an error otherwise.

Usage

```
rrq_task_result(task_id, error = FALSE, follow = NULL, controller = NULL)
```

Arguments

`task_id`	The single id for which the result is wanted.
`error`	Logical, indicating if we should throw an error if a task was not successful. By default (`error = FALSE`), in the case of the task result returning an error we return an object of class `rrq_task_error`, which contains information about the error. Passing `error = TRUE` calls `stop()` on this error if it is returned.
`follow`	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option `rrq.follow`, and finally `TRUE`. Set to `FALSE` if you want to return information about the original task, even if it has been subsequently retried.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

The result of your task

rrq_task_results	*Get the results of a group of tasks, returning them as a list. See rrq_task_result for getting the result of a single task.*

Description

Get the results of a group of tasks, returning them as a list. See rrq_task_result for getting the result of a single task.

Usage

```
rrq_task_results(
  task_ids,
  error = FALSE,
  named = FALSE,
  follow = NULL,
  controller = NULL
)
```

Arguments

task_ids	A vector of task ids for which the task result is wanted.
error	Logical, indicating if we should throw an error if the task was not successful. See `rrq_task_result()` for details.
named	Logical, indicating if the return value should be named with the task ids; as these are quite long this can make the value a little awkward to work with.
follow	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option `rrq.follow`, and finally `TRUE`. Set to `FALSE` if you want to return information about the original task, even if it has been subsequently retried.
controller	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

An unnamed list, one entry per result. This function errors if any task is not available.

rrq_task_retry *Retry tasks*

Description

Retry a task (or set of tasks). Typically this is after failure (e.g., ERROR, DIED or similar) but you can retry even successfully completed tasks. Once retried, functions that retrieve information about a task (e.g., rrq_task_status(), [rrq_task_result()]) will behave differently depending on the value of their lowargument. Seevignette("fault-tolerance")‘ for more details.

Usage

```
rrq_task_retry(task_ids, controller = NULL)
```

Arguments

task_ids	Task ids to retry.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

New task ids

rrq_task_status *Fetch task statuses*

Description

Return a character vector of task statuses. The name of each element corresponds to a task id, and the value will be one of the possible statuses ("PENDING", "COMPLETE", etc).

Usage

```
rrq_task_status(task_ids, named = FALSE, follow = NULL, controller = NULL)
```

Arguments

task_ids	Optional character vector of task ids for which you would like statuses.
named	Logical, indicating if the return value should be named with the task ids; as these are quite long this can make the value a little awkward to work with.
follow	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option rrq.follow, and finally TRUE. Set to FALSE if you want to return information about the original task, even if it has been subsequently retried.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A character vector the same length as `task_ids`

`rrq_task_times` *Fetch task times*

Description

Fetch times for tasks at points in their life cycle. For each task returns the time of submission, starting and completion (not necessarily successfully; this includes errors and interruptions). If a task has not reached a point yet (e.g., submitted but not run, or running but not finished) the time will be `NA`). Times are returned in unix timestamp format in UTC; you can use redux::redis_time_to_r to convert them to a POSIXt object.

Usage

```
rrq_task_times(task_ids, follow = NULL, controller = NULL)
```

Arguments

`task_ids`	A vector of task ids
`follow`	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option `rrq.follow`, and finally `TRUE`. Set to `FALSE` if you want to return information about the original task, even if it has been subsequently retried.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A matrix of times, but we might change this to a data.frame at some point in the future.

`rrq_task_wait` *Wait for group of tasks*

Description

Wait for a task, or set of tasks, to complete. If you have used `rrq` prior to version 0.8.0, you might expect this function to return the result, but we now return a logical value which indicates success or not. You can fetch the task result with rrq_task_result.

Usage

```
rrq_task_wait(
  task_id,
  timeout = NULL,
  time_poll = 1,
  progress = NULL,
  follow = NULL,
  controller = NULL
)
```

Arguments

task_id	A vector of task ids to poll for (can be one task or many)
timeout	Optional timeout, in seconds, after which an error will be thrown if the task has not completed. If not given, falls back on the controller's timeout_task_wait (see rrq_controller)
time_poll	Optional time with which to "poll" for completion. By default this will be 1 second; this is the time that each request for a completed task may block for (however, if the task is finished before this, the actual time waited for will be less). Increasing this will reduce the responsiveness of your R session to interrupting, but will cause slightly less network load. Values less than 1s are only suppored with Redis server version 6.0.0 or greater (released September 2020).
progress	Optional logical indicating if a progress bar should be displayed. If NULL we fall back on the value of the global option rrq.progress, and if that is unset display a progress bar if in an interactive session.
follow	Optional logical, indicating if we should follow any redirects set up by doing rrq_task_retry. If not given, falls back on the value passed into the controller, the global option rrq.follow, and finally TRUE. Set to FALSE if you want to return information about the original task, even if it has been subsequently retried.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A scalar logical value; TRUE if *all* tasks complete successfully and FALSE otherwise

---

rrq_worker *rrq queue worker*

Description

rrq queue worker

rrq queue worker

Details

A rrq queue worker. These are not typically for interacting with but will sit and poll a queue for jobs.

Public fields

`id` The id of the worker

`config` The name of the configuration used by this worker

`controller` An rrq controller object

Methods

Public methods:

- `rrq_worker$new()`
- `rrq_worker$info()`
- `rrq_worker$log()`
- `rrq_worker$load_envir()`
- `rrq_worker$poll()`
- `rrq_worker$step()`
- `rrq_worker$loop()`
- `rrq_worker$format()`
- `rrq_worker$timer_start()`
- `rrq_worker$progress()`
- `rrq_worker$task_eval()`
- `rrq_worker$shutdown()`

Method `new()`: Constructor

Usage:

```
rrq_worker$new(
  queue_id,
  name_config = "localhost",
  worker_id = NULL,
  timeout_config = 0,
  is_child = FALSE,
  con = redux::hiredis()
)
```

Arguments:

`queue_id` The queue id

`name_config` Optional name of the configuration. The default "localhost" configuration always exists. Create new configurations using rrq_worker_config_save.

`worker_id` Optional worker id. If omitted, a random id will be created.

`timeout_config` How long to try and read the worker configuration for. Will attempt to read once a second and throw an error if config cannot be located after `timeout` seconds. Use this to create workers before their configurations are available. The default (0) is to assume that the configuration is immediately available.

`is_child` Logical, used to indicate that this is a child of the real worker. If `is_child` is `TRUE`, then most other arguments here have no effect (e.g., queue all the timeout / idle / polling arguments) as they come from the parent. Not for general use.

`con` A redis connection

Method `info()`**:** Return information about this worker, a list of key-value pairs.

Usage:

```
rrq_worker$info()
```

Method `log()`**:** Create a log entry. This will print a human readable format to screen and a machine-readable format to the redis database.

Usage:

```
rrq_worker$log(label, value = NULL)
```

Arguments:

`label` Scalar character, the title of the log entry

`value` Character vector (or null) with log values

Method `load_envir()`**:** Load the worker environment by creating a new environment object and running the create hook (if configured). See `rrq_worker_envir_set()` for details.

Usage:

```
rrq_worker$load_envir()
```

Method `poll()`**:** Poll for work

Usage:

```
rrq_worker$poll(immediate = FALSE)
```

Arguments:

`immediate` Logical, indicating if we should *not* do a blocking wait on the queue but instead reducing the timeout to zero. Intended primarily for use in the tests.

Method `step()`**:** Take a single "step". This consists of

1. Poll for work (`$poll()`)
2. If work found, run it (either a task or a message)
3. If work not found, check the timeout

Usage:

```
rrq_worker$step(immediate = FALSE)
```

Arguments:

`immediate` Logical, indicating if we should *not* do a blocking wait on the queue but instead reducing the timeout to zero. Intended primarily for use in the tests.

Method `loop()`**:** The main worker loop. Use this to set up the main worker event loop, which will continue until exiting (via a timeout or message).

Usage:

```
rrq_worker$loop(immediate = FALSE)
```

Arguments:

`immediate` Logical, indicating if we should *not* do a blocking wait on the queue but instead reducing the timeout to zero. Intended primarily for use in the tests.

Method `format()`**:** Create a nice string representation of the worker. Used automatically to print the worker by R6.

Usage:

```
rrq_worker$format()
```

Method `timer_start()`**:** Start the timer

Usage:

```
rrq_worker$timer_start()
```

Method `progress()`**:** Submit a progress message. See `rrq_task_progress_update()` for details of this mechanism.

Usage:

```
rrq_worker$progress(value, error = TRUE)
```

Arguments:

`value` An R object with the contents of the update. This will overwrite any previous progress value, and can be retrieved with rrq_task_progress. A value of `NULL` will appear to clear the status, as `NULL` will also be returned if no status is found for a task.

`error` Logical, indicating if we should throw an error if not running as an `rrq` task. Set this to `FALSE` if you want code to work without modification within and outside of an `rrq` job, or to `TRUE` if you want to be sure that progress messages have made it to the server.

Method `task_eval()`**:** Evaluate a task. When running a task on a separate process, we will always set two environment variables: * `RRQ_WORKER_ID` this is the id field * `RRQ_TASK_ID` this is the task id

Usage:

```
rrq_worker$task_eval(task_id)
```

Arguments:

`task_id` A task identifier. It is undefined what happens if this identifier does not exist.

Method `shutdown()`**:** Stop the worker

Usage:

```
rrq_worker$shutdown(status = "OK", graceful = TRUE)
```

Arguments:

`status` the worker status; typically be one of `OK` or `ERROR` but can be any string

`graceful` Logical, indicating if we should request a graceful shutdown of the heartbeat, if running.

`rrq_worker_config` *Create worker configuration*

Description

Create a worker configuration, suitable to pass into rrq_worker_config_save. The results of this function should not be modified.

Usage

```
rrq_worker_config(
  queue = NULL,
  verbose = TRUE,
  poll_queue = NULL,
  timeout_idle = Inf,
  poll_process = 1,
  timeout_process_die = 2,
  heartbeat_period = NULL
)
```

Arguments

`queue`	Optional character vector of queues to listen on for tasks. There is a default queue which is always listened on (called 'default'). You can specify additional names here and tasks put onto these queues with `rrq_task_create_expr()` (or other functions) will have *higher* priority than the default. You can explicitly list the "default" queue (e.g., `queue = c("high", "default", "low")`) to set the position of the default queue.
`verbose`	Logical, indicating if the worker should print logging output to the screen. Logging to screen has a small but measurable performance cost, and if you will not collect system logs from the worker then it is wasted time. Logging to the redis server is always enabled.
`poll_queue`	Polling time for new tasks on the queue or messages. Longer values here will reduce the impact on the database but make workers less responsive to being killed with an interrupt (control-C or Escape). The default should be good for most uses, but shorter values are used for debugging. Importantly, longer times here do not increase the time taken for a worker to detect new tasks.
`timeout_idle`	Optional timeout that sets the length of time after which the worker will exit if it has not processed a task. This is (roughly) equivalent to issuing a `TIMEOUT_SET` message after initialising the worker, except that it's guaranteed to be run by all workers.
`poll_process`	Polling time indicating how long to wait for a background process to produce stdout or stderr. Only used for tasks queued with `separate_process TRUE`.
`timeout_process_die`	Timeout indicating how long to wait wait for the background process to respond to SIGTERM, either as we stop a worker or cancel a task. Only used for tasks

queued with separate_process TRUE. If your tasks may take several seconds to stop, you may want to increase this to ensure a clean exit.

heartbeat_period
: Optional period for the heartbeat. If non-NULL then a heartbeat process will be started (using rrq_heartbeat) which can be used to build fault tolerant queues. See vignette("fault-tolerance") for details. If NULL (the default), then no heartbeat is configured.

Value

A list of values with class rrq_worker_config; these should be considered read-only, and contain only the validated input parameters.

Examples

```
rrq::rrq_worker_config()
```

---

rrq_worker_config_list

List worker configurations

Description

Return names of worker configurations saved by rrq_worker_config_save()

Usage

```
rrq_worker_config_list(controller = NULL)
```

Arguments

controller
: The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A character vector of names; these can be passed as the name argument to rrq_worker_config_read().

rrq_worker_config_read

Read worker configuration

Description

Return the value of a of worker configuration saved by `rrq_worker_config_save()`

Usage

```
rrq_worker_config_read(name, timeout = 0, controller = NULL)
```

Arguments

`name`	Name of the configuration (see `rrq_worker_config_list()`)
`timeout`	Optionally, a timeout to wait for a worker configuration to appear. Generally you won't want to set this, but it can be used to block until a configuration becomes available.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

rrq_worker_config_save

Save worker configuration

Description

Save a worker configuration, which can be used to start workers with a set of options with the cli. These correspond to arguments to rrq_worker. **This function will be renamed soon**

Usage

```
rrq_worker_config_save(name, config, overwrite = TRUE, controller = NULL)
```

Arguments

`name`	Name for this configuration
`config`	A worker configuration, created by `rrq_worker_config()`
`overwrite`	Logical, indicating if an existing configuration with this `name` should be overwritten if it exists. If `FALSE`, then the configuration is not updated, even if it differs from the version currently saved.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

Invisibly, a boolean indicating if the configuration was updated.

`rrq_worker_delete_exited` *Clean up exited workers*

Description

Cleans up workers known to have exited

Usage

```
rrq_worker_delete_exited(worker_ids = NULL, controller = NULL)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then rrq looks for exited workers using `rrq_worker_list_exited()`. If given, we check that the workers are known and have exited.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A character vector of workers that were deleted

`rrq_worker_detect_exited` *Detect exited workers*

Description

Detects exited workers through a lapsed heartbeat. This differs from `rrq_worker_list_exited()` which lists workers that have definitely exited by checking to see if any worker that runs a heartbeat process has not reported back in time, then marks that worker as exited. See vignette("fault-tolerance") for details.

Usage

```
rrq_worker_detect_exited(controller = NULL)
```

Arguments

`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

rrq_worker_envir_set *Set worker environment*

Description

Register a function to create an environment when creating a worker. When a worker starts, they will run this function.

Usage

```
rrq_worker_envir_set(create, notify = TRUE, controller = NULL)
```

Arguments

create	A function that will create an environment. It will be called with one parameter (an environment), in a fresh R session. The function rrq_envir() can be used to create a suitable function for the most common case (loading packages and sourcing scripts).
notify	Boolean, indicating if we should send a REFRESH message to all workers to update their environment.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

rrq_worker_exists *Test if a worker exists*

Description

Test if a worker exists

Usage

```
rrq_worker_exists(name, controller = NULL)
```

Arguments

name	Name of the worker
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Value

A logical value

rrq_worker_info *Worker information*

Description

Returns a list of information about active workers (or exited workers if `worker_ids` includes them).

Usage

```
rrq_worker_info(worker_ids = NULL, controller = NULL)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers are used.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A list of `worker_info` objects

rrq_worker_len *Number of active workers*

Description

Returns the number of active workers

Usage

```
rrq_worker_len(controller = NULL)
```

Arguments

`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

An integer

rrq_worker_list *List active workers*

Description

Returns the ids of active workers. This does not include exited workers; use `rrq_worker_list_exited()` for that.

Usage

```
rrq_worker_list(controller = NULL)
```

Arguments

`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A character vector of worker names

rrq_worker_list_exited *List exited workers*

Description

Returns the ids of workers known to have exited

Usage

```
rrq_worker_list_exited(controller = NULL)
```

Arguments

`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A character vector of worker names

rrq_worker_load *Report on worker load*

Description

Report on worker "load" (the number of workers being used over time). Reruns an object of class `worker_load`, for which a `mean` method exists (this function is a work in progress and the interface may change).

Usage

```
rrq_worker_load(worker_ids = NULL, controller = NULL)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers are used.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

An object of class "worker_load", which has a pretty print method.

rrq_worker_log_tail *Returns the last (few) elements in the worker log, in a programmatically useful format (see Value).*

Description

Returns the last (few) elements in the worker log, in a programmatically useful format (see Value).

Usage

```
rrq_worker_log_tail(worker_ids = NULL, n = 1, controller = NULL)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers are used.
`n`	Number of elements to select, the default being the single last entry. Use `Inf` or `0` to indicate that you want all log entries
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A data.frame with columns:

- `worker_id`: the worker id
- `child`: the process id, an integer, where logs come from a child process from a task queued with `separate_process = TRUE`
- `time`: the time from Redis when the event happened; see redux::redis_time to convert this to an R time
- `command`: the command sent from or to the worker
- `message`: the message corresponding to that command

`rrq_worker_process_log` *Read worker process log*

Description

Return the contents of a worker's process log, if it is located on the same physical storage (including network storage) as the controller. This will generally behave for workers started with rrq_worker_spawn but may require significant care otherwise.

Usage

```
rrq_worker_process_log(worker_id, controller = NULL)
```

Arguments

`worker_id`	The worker id for which the log is required
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

`rrq_worker_script` *Write worker runner script*

Description

Write a small script that can be used to launch a rrq worker. The resulting script takes the same arguments as the rrq_worker constructor, but from the command line. See Details.

Usage

```
rrq_worker_script(path, versioned = FALSE)
```

Arguments

path	The path to write to. Should be a directory (or one will be created if it does not yet exist). The final script will be `file.path(path, "rrq_worker")`
versioned	Logical, indicating if we should write a versioned R script that will use the same path to `Rscript` as the running session. If `FALSE` we use `#!/usr/bin/env Rscript` which will pick up `Rscript` from the path. You may want to use a versioned script in tests or if you have multiple R versions installed simultaneously.

Details

If you need to launch rrq workers from a script, it's convenient not to have to embed R code like:

```
Rscript -e 'rrq::rrq_worker$new("myqueue")'
```

as this is error-prone and unpleasant to quote and read. You can use the function `rrq_worker_script` to write out a small helper script which lets you write:

```
./path/rrq_worker myqueue
```

instead.

The helper script supports the same arguments as the `[rrq::rrq_worker]` constructor:

- `queue_id` as the sole positional argument
- `name_config` as `--config`
- `worker_id` as `--worker-id`

To change the redis connection settings, set the `REDIS_URL` environment variable (see `redux::hiredis()` for details).

For example to create a worker `myworker` with configuration `myconfig` on queue `myqueue` you might use

```
./rrq_worker --config=myconfig --worker-id=myworker myqueue
```

Value

Invisibly, the path to the script

Examples

```
path <- rrq::rrq_worker_script(tempfile())
readLines(path)
```

`rrq_worker_spawn` *Spawn a worker*

Description

Spawn a worker in the background

Usage

```
rrq_worker_spawn(
  n = 1,
  logdir = NULL,
  timeout = 600,
  name_config = "localhost",
  worker_id_base = NULL,
  time_poll = 0.2,
  progress = NULL,
  controller = NULL
)
```

Arguments

`n`	Number of workers to spawn
`logdir`	Path of a log directory to write the worker process log to, interpreted relative to the current working directory
`timeout`	Time to wait for workers to appear. If 0 then we don't wait for workers to appear (you can run the `wait_alive` method of the returned object to run this test manually)
`name_config`	Name of the configuration to use. By default the `"localhost"` configuration is used
`worker_id_base`	Optional base to construct the worker ids from. If omitted a random base will be used. Actual ids will be created but appending integers to this base.
`time_poll`	Polling period (in seconds) while waiting for workers to come up.
`progress`	Show a progress bar while waiting for workers (when `timeout` is at least 0)
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Details

Spawning multiple workers. If `n` is greater than one, multiple workers will be spawned. This happens in parallel so it does not take n times longer than spawning a single worker.

Beware that signals like Ctrl-C passed to *this* R instance can still propagate to the child processes and can result in them dying unexpectedly. It is probably safer to start processes in a completely separate session.

Value

An `rrq_worker_manager` object with fields:

- `id`: the ids of the spawned workers
- `wait_alive`: a method to wait for workers to come alive
- `stop`: a method to stop workers
- `kill`: a method to kill workers abruptly by sending a signal
- `is_alive`: a method that checks if a worker is currently alive
- `logs`: a method that returns logs for a single worker

All the methods accept a vector of worker names, or integers, except `logs` which requires a single worker id (as a string or integer). For all methods except `logs`, the default of `NULL` means "all managed workers".

`rrq_worker_status` *Worker statuses*

Description

Returns a character vector of current worker statuses

Usage

```
rrq_worker_status(worker_ids = NULL, controller = NULL)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers are used.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A character vector of statuses, named by worker

`rrq_worker_stop` *Stop workers*

Description

Stop workers.

Usage

```
rrq_worker_stop(
  worker_ids = NULL,
  type = "message",
  timeout = 0,
  time_poll = 0.1,
  progress = NULL,
  controller = NULL
)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers will be stopped.
`type`	The strategy used to stop the workers. Can be `message`, `kill` or `kill_local` (see Details).
`timeout`	Optional timeout; if greater than zero then we poll for a response from the worker for this many seconds until they acknowledge the message and stop (only has an effect if `type` is `message`). If a timeout of greater than zero is given, then for a `message`-based stop we wait up to this many seconds for the worker to exit. That means that we might wait up to `2 * timeout` seconds for this function to return.
`time_poll`	If `type` is `message` and `timeout` is greater than zero, this is the polling interval used between redis calls. Increasing this reduces network load but decreases the ability to interrupt the process.
`progress`	Optional logical indicating if a progress bar should be displayed. If `NULL` we fall back on the value of the global option `rrq.progress`, and if that is unset display a progress bar if in an interactive session.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Details

The `type` parameter indicates the strategy used to stop workers, and interacts with other parameters. The strategies used by the different values are:

- `message`, in which case a `STOP` message will be sent to the worker, which they will receive after finishing any currently running task (if `RUNNING`; `IDLE` workers will stop immediately).

- `kill`, in which case a kill signal will be sent via the heartbeat (if the worker is using one). This will kill the worker even if is currently working on a task, eventually leaving that task with a status of `DIED`.
- `kill_local`, in which case a kill signal is sent using operating system signals, which requires that the worker is on the same machine as the controller.

Value

The names of the stopped workers, invisibly.

`rrq_worker_task_id` *Current task id for workers*

Description

Returns the task id that each worker is working on, if any.

Usage

```
rrq_worker_task_id(worker_ids = NULL, controller = NULL)
```

Arguments

`worker_ids`	Optional vector of worker ids. If `NULL` then all active workers are used.
`controller`	The controller to use. If not given (or `NULL`) we'll use the controller registered with `rrq_default_controller_set()`.

Value

A character vector, `NA` where nothing is being worked on, otherwise corresponding to a task id.

`rrq_worker_wait` *Wait for workers*

Description

Wait for workers to appear.

Usage

```
rrq_worker_wait(
  worker_ids,
  timeout = Inf,
  time_poll = 0.2,
  progress = NULL,
  controller = NULL
)
```

Arguments

worker_ids	A vector of worker ids to wait for
timeout	Timeout in seconds; default is to wait forever
time_poll	Poll interval, in seconds. Must be an integer
progress	Optional logical indicating if a progress bar should be displayed. If NULL we fall back on the value of the global option rrq.progress, and if that is unset display a progress bar if in an interactive session.
controller	The controller to use. If not given (or NULL) we'll use the controller registered with rrq_default_controller_set().

Index