Package: queue (via r-universe)

September 12, 2024

Title Simple Multi-Threaded Task Queuing

Version 0.0.2

Description Implements a simple multi-threaded task queue using R6
classes.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Imports callr, cli, R6, tibble

URL `https://github.com/djnavarro/queue`, `http://djnavarro.net/queue/`

BugReports `https://github.com/djnavarro/queue/issues`

Suggests covr, knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Repository https://djnavarro.r-universe.dev

RemoteUrl https://github.com/djnavarro/queue

RemoteRef HEAD

RemoteSha 4c70aad373fd518250c6bd6c29cebccb6d16dc65

Contents

Queue *R6 Class Representing a Multi-Worker Task Queue*

Description

A Queue executes tasks concurrently using multiple workers.

Details

The Queue class is primary interface provided by the queue package. It allows users to execute an arbitrary collection of tasks in parallel across multiple R sessions, managed automatically in the background. Once a new queue is initialised, tasks can be added to the queue using the `add()` method. Once all tasks are added, they are executed in parallel by calling the `run()` method. When completed, `run()` returns a tibble that contains the results for all tasks, and some additional metadata besides.

Internally, a Queue uses a `TaskList` object as its data store and a `WorkerPool` object to execute the tasks in parallel. These objects can be accessed by calling the `get_tasks()` method and the `get_workers()` methods. Usually you would not need to do this, but occasionally it can be useful because those objects have some handy methods that allow finer-grained control (see the documentation for `TaskList` and `WorkerPool` respectively).

Methods

Public methods:

- `Queue$new()`
- `Queue$add()`
- `Queue$run()`
- `Queue$get_workers()`
- `Queue$get_tasks()`
- `Queue$clone()`

Method `new()`**:** Create a task queue

Usage:

```
Queue$new(workers = 4L)
```

Arguments:

`workers` Either the number of workers to employ in the task queue, or a `WorkerPool` object to use when deploying the tasks.

Returns: A new Queue object

Method `add()`**:** Adds a task to the queue

Usage:

```
Queue$add(fun, args = list(), id = NULL)
```

Arguments:

`fun` The function to be called when the task is scheduled

`args` A list of arguments to be passed to the task function (optional)

`id` A string specifying a unique identifier for the task (optional: tasks will be named "task_1", "task_2", etc if this is unspecified)

Returns: Invisibly returns the `Task` object

Method `run()`: Execute tasks in parallel using the worker pool, assigning tasks to workers in the same order in which they were added to the queue

Usage:

```
Queue$run(
  timelimit = 60,
  message = "minimal",
  interval = 0.05,
  shutdown = TRUE
)
```

Arguments:

`timelimit` How long (in seconds) should the worker pool wait for a task to complete before terminating the child process and moving onto the next task? (default is 60 seconds, but this is fairly arbitrary)

`message` What messages should be reported by the queue while it is running? Options are "none" (no messages), "minimal" (a spinner is shown alongside counts of waiting, running, and completed tasks), and "verbose" (in addition to the spinner, each task is summarized as it completes). Default is "minimal".

`interval` How often should the task queue poll the workers to see if they have finished their assigned tasks? Specified in seconds.

`shutdown` Should the workers in the pool be shut down (i.e., all R sessions closed) once the tasks are completed. Defaults to `TRUE`.

Returns: Returns a tibble containing the results of all tasks and various other useful metadata. Contains one row per task in the `Queue`, and the following columns:

- `task_id` A character string specifying the task identifiers
- `worker_id` An integer specifying the worker process ids (pid)
- `state` A character string indicating the status of each task ("created", "waiting", "assigned", "running", or "done")
- `result` A list containing the function outputs, or NULL
- `runtime` Completion time for the task (NA if the task is not done)
- `fun` A list containing the functions
- `args` A list containing the arguments passed to each function
- `created` The time at which each task was created
- `queued` The time at which each task was added to a `Queue`
- `assigned` The time at which each task was assigned to a `Worker`
- `started` The time at which a `Worker` called each function
- `finished` The time at which a `Worker` output was returned for the task
- `code` The status code returned by the callr R session (integer)
- `message` The message returned by the callr R session (character)

- `stdout` List column containing the contents of stdout during function execution
- `stderr` List column containing the contents of stderr during function execution
- `error` List column containing `NULL` values

Note: at present there is one field from the callr rsession::read() method that isn't captured here, and that's the error field. I'll add that after I've finished wrapping my head around what that actually does. The `error` column, at present, is included only as a placeholder

Method `get_workers()`**:** Retrieve the workers

Usage:

`Queue$get_workers()`

Returns: A `WorkerPool` object

Method `get_tasks()`**:** Retrieve the tasks

Usage:

`Queue$get_tasks()`

Returns: A `TaskList` object

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`Queue$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
queue <- Queue$new(workers = 4L)
wait <- function(x) Sys.sleep(runif(1))
for(i in 1:6) queue$add(wait)
queue$run()
```

`Task` *R6 Class Representing a Task*

Description

A `Task` stores a function, arguments, output, and metadata.

Details

A Task object is used as a storage class. It is a container used to hold an R function and any arguments to be passed to the function. It can also hold any output returned by the function, anything printed to stdout or stderr when the function is called, and various other metadata such as the process id of the worker that executed the function, timestamps, and so on.

The methods for Task objects fall into two groups, roughly speaking. The get_*() methods are used to return information about the Task, and the register_*() methods are used to register information related to events relevant to the Task status.

The retrieve() method is special, and returns a tibble containing all information stored about the task. Objects further up the hierarchy use this method to return nicely organised output that summarise the results from many tasks.

Methods

Public methods:

- Task$new()
- Task$retrieve()
- Task$get_task_fun()
- Task$get_task_args()
- Task$get_task_state()
- Task$get_task_id()
- Task$get_task_runtime()
- Task$register_task_created()
- Task$register_task_waiting()
- Task$register_task_assigned()
- Task$register_task_running()
- Task$register_task_done()
- Task$clone()

Method new(): Create a new task. Conceptually, a Task is viewed as a function that will be executed by the Worker to which it is assigned, and it is generally expected that any resources the function requires are passed through the arguments since the execution context will be a different R session to the one in which the function is defined.

Usage:

```
Task$new(fun, args = list(), id = NULL)
```

Arguments:

fun The function to be called when the task executes.

args A list of arguments to be passed to the function (optional).

id A string specifying a unique task identifier (optional).

Returns: A new Task object.

Method retrieve(): Retrieve a tidy summary of the task state.

Usage:

```
Task$retrieve()
```

Returns: A tibble containing a single row, and the following columns:

- `task_id` A character string specifying the task identifier
- `worker_id` An integer specifying the worker process id (pid)
- `state` A character string indicating the task status ("created", "waiting", "assigned", "running", or "done")
- `result` A list containing the function output, or NULL
- `runtime` Completion time for the task (NA if the task is not done)
- `fun` A list containing the function
- `args` A list containing the arguments
- `created` The time at which the task was created
- `queued` The time at which the task was added to a `Queue`
- `assigned` The time at which the task was assigned to a `Worker`
- `started` The time at which the `Worker` called the function
- `finished` The time at which the `Worker` output was returned
- `code` The status code returned by the callr R session (integer)
- `message` The message returned by the callr R session (character)
- `stdout` List containing the contents of stdout during function execution
- `stderr` List containing the contents of stderr during function execution
- `error` List containing `NULL`

Note: at present there is one field from the callr rsession::read() method that isn't captured here, and that's the error field. I'll add that after I've finished wrapping my head around what that actually does. The `error` column, at present, is included only as a placeholder

Method `get_task_fun()`**:** Retrieve the task function.

Usage:

```
Task$get_task_fun()
```

Returns: A function.

Method `get_task_args()`**:** Retrieve the task arguments

Usage:

```
Task$get_task_args()
```

Returns: A list.

Method `get_task_state()`**:** Retrieve the task state.

Usage:

```
Task$get_task_state()
```

Returns: A string specifying the current state of the task. Possible values are "created" (task exists), "waiting" (task exists and is waiting in a queue), "assigned" (task has been assigned to a worker but has not yet started), "running" (task is running on a worker), or "done" (task is completed and results have been assigned back to the task object)

Method `get_task_id()`**:** Retrieve the task id.

Usage:

```
Task$get_task_id()
```

Returns: A string containing the task identifier.

Method `get_task_runtime()`**:** Retrieve the task runtime.

Usage:

```
Task$get_task_runtime()
```

Returns: If the task has completed, a difftime value. If the task has yet to complete, a `NA` value is returned

Method `register_task_created()`**:** Register the task creation by updating internal storage. When this method is called, the state of the `Task` is set to "created" and a timestamp is recorded, registering the creation time for the task. This method is intended to be called by `Worker` objects. Users should not need to call it.

Usage:

```
Task$register_task_created()
```

Returns: Returns `NULL` invisibly.

Method `register_task_waiting()`**:** Register the addition of the task to a queue by updating internal storage. When this method is called, the state of the `Task` is set to "waiting" and a timestamp is recorded, registering the time at which the task was added to a queue. This method is intended to be called by `Worker` objects. Users should not need to call it.

Usage:

```
Task$register_task_waiting()
```

Returns: Returns `NULL` invisibly.

Method `register_task_assigned()`**:** Register the assignment of a task to a worker by updating internal storage. When this method is called, the state of the `Task` is set to "assigned" and a timestamp is recorded, registering the time at which the task was assigned to a `Worker`. In addition, the `worker_id` of the worker object (which is also it's pid) is registered with the task. This method is intended to be called by `Worker` objects. Users should not need to call it.

Usage:

```
Task$register_task_assigned(worker_id)
```

Arguments:

`worker_id` Identifier for the worker to which the task is assigned.

Returns: Returns `NULL` invisibly.

Method `register_task_running()`**:** Register the commencement of a task to a worker by updating internal storage. When this method is called, the state of the `Task` is set to "running" and a timestamp is recorded, registering the time at which the `Worker` called the task function. In addition, the `worker_id` is recorded, albeit somewhat unnecessarily since this information is likely already stored when `register_task_assigned()` is called. This method is intended to be called by `Worker` objects. Users should not need to call it.

Usage:

`Task$register_task_running(worker_id)`

Arguments:

`worker_id` Identifier for the worker on which the task is starting.

Returns: Returns `NULL` invisibly.

Method `register_task_done()`**:** Register the finishing of a task to a worker by updating internal storage. When this method is called, the state of the `Task` is set to "done" and a timestamp is recorded, registering the time at which the `Worker` returned results to the `Task`. The `results` object is read from the R session, and is stored locally by the `Task` at this time. This method is intended to be called by `Worker` objects. Users should not need to call it.

Usage:

`Task$register_task_done(results)`

Arguments:

`results` Results read from the R session.

Returns: Returns `NULL` invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`Task$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`TaskList` *R6 Class Representing a Task List*

Description

A `TaskList` stores and retrieves one or more tasks.

Details

The `TaskList` class is used as a storage class. It provides a container that holds a collection of `Task` objects, along with a collection of methods for adding, removing, and getting `Tasks`. It can also report on the status of the `Tasks` contained within the list and retrieve results from those `Tasks`. What it cannot do is manage interactions with `Workers` or arrange for the `Tasks` to be executed. That's the job of the `Queue`.

Methods

Public methods:

- `TaskList$new()`
- `TaskList$length()`
- `TaskList$add_task()`

- `TaskList$remove_task()`
- `TaskList$get_task()`
- `TaskList$get_state()`
- `TaskList$get_tasks_in_state()`
- `TaskList$retrieve()`
- `TaskList$clone()`

Method `new()`**:** Create a new task list

Usage:

```
TaskList$new()
```

Method `length()`**:** Return the number of tasks in the list

Usage:

```
TaskList$length()
```

Returns: Integer

Method `add_task()`**:** Add a task to the `TaskList`

Usage:

```
TaskList$add_task(task)
```

Arguments:

`task` The `Task` object to be added

Method `remove_task()`**:** This method removes one or more tasks from the `TaskList`.

Usage:

```
TaskList$remove_task(x)
```

Arguments:

`x` Indices of the tasks to be removed

Method `get_task()`**:** Return a single `Task` contained in the `TaskList`. The `Task` is not removed from the `TaskList`, and has reference semantics: if the listed task is completed by a `Worker`, then the status of any `Task` returned by this method will update automatically

Usage:

```
TaskList$get_task(x)
```

Arguments:

`x` The index the task to return

Returns: A `Task` object

Method `get_state()`**:** Return the status of all tasks in the `TaskList`.

Usage:

```
TaskList$get_state()
```

Returns: A character vector specifying the completion status for all listed tasks

Method `get_tasks_in_state()`**:** Return a list of tasks in a given state

Usage:

```
TaskList$get_tasks_in_state(x)
```

Arguments:

`x` The name of the state (e.g., "waiting")

Returns: A `TaskList` object

Method `retrieve()`**:** Retrieves the current state of all tasks.

Usage:

```
TaskList$retrieve()
```

Returns: Returns a tibble containing the results of all tasks and various other useful metadata. Contains one row per task in the `TaskList`, and the following columns:

- `task_id` A character string specifying the task identifiers
- `worker_id` An integer specifying the worker process ids (pid)
- `state` A character string indicating the status of each task ("created", "waiting", "assigned", "running", or "done")
- `result` A list containing the function outputs, or NULL
- `runtime` Completion time for the task (NA if the task is not done)
- `fun` A list containing the functions
- `args` A list containing the arguments passed to each function
- `created` The time at which each task was created
- `queued` The time at which each task was added to a `Queue`
- `assigned` The time at which each task was assigned to a `Worker`
- `started` The time at which a `Worker` called each function
- `finished` The time at which a `Worker` output was returned for the task
- `code` The status code returned by the callr R session (integer)
- `message` The message returned by the callr R session (character)
- `stdout` List column containing the contents of stdout during function execution
- `stderr` List column containing the contents of stderr during function execution
- `error` List column containing `NULL` values

If all tasks have completed this output is the same as the output as the `run()` method for a `Queue` object.
Note: at present there is one field from the callr rsession::read() method that isn't captured here, and that's the error field. I'll add that after I've finished wrapping my head around what that actually does. The `error` column, at present, is included only as a placeholder

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TaskList$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Worker *R6 Class Representing a Worker*

Description

A `Worker` manages an external R session and completes tasks.

Details

The `Worker` class interacts with an external R session, and possesses methods that allow it to work with `Task` objects. At its core, the class is a thin wrapper around a `callr::r_session` object, and in fact the session object itself can be obtained by calling the `get_worker_session()` method. In most cases this shouldn't be necessary however, because `Worker` objects are typically created as part of a `WorkerPool` that is managed by a `Queue`, and those higher level structures use the methods exposed by the `Worker` object.

Methods

Public methods:

- `Worker$new()`
- `Worker$get_worker_id()`
- `Worker$get_worker_state()`
- `Worker$get_worker_runtime()`
- `Worker$get_worker_task()`
- `Worker$get_worker_session()`
- `Worker$try_assign()`
- `Worker$try_start()`
- `Worker$try_finish()`
- `Worker$shutdown_worker()`
- `Worker$clone()`

Method `new()`**:** Create a new worker object.

Usage:

```
Worker$new()
```

Returns: A new `Worker` object.

Method `get_worker_id()`**:** Retrieve the worker identifier.

Usage:

```
Worker$get_worker_id()
```

Returns: The worker identifier, which also the process id for the R session

Method `get_worker_state()`**:** Retrieve the worker state.

Usage:

```
Worker$get_worker_state()
```

Returns: A string specifying the current state of the R session. Possible values are:

- `"starting"`: the R session is starting up.
- `"idle"`: the R session is ready to compute.
- `"busy"`: the R session is computing.
- `"finished"`: the R session has terminated.

Importantly, note that a task function that is still running and a task function that is essentially finished and waiting to return will both return "busy". To distinguish between these two cases you need to use the `poll_process()` method of a `callr::rsession`, as returned by `get_worker_session()`.

Method `get_worker_runtime()`**:** Return the total length of time the worker session has been running, and the length of the time that the current task has been running. If the session is finished both values are `NA`. If the session is idle (no task running) the total session time will return a value but the current task time will be `NA`.

Usage:

```
Worker$get_worker_runtime()
```

Returns: A vector of two difftimes.

Method `get_worker_task()`**:** Retrieve the task assigned to the worker.

Usage:

```
Worker$get_worker_task()
```

Returns: The `Task` object currently assigned to this `Worker`, or `NULL`.

Method `get_worker_session()`**:** Retrieve the R session associated with a `Worker`

Usage:

```
Worker$get_worker_session()
```

Returns: An R session object, see `callr::r_session`

Method `try_assign()`**:** Attempt to assign a task to this worker. This method checks that the task and the worker are both in an appropriate state. If they are, both objects register their connection to the other. This method is intended to be called by a `WorkerPool` or a `Queue`.

Usage:

```
Worker$try_assign(task)
```

Arguments:

`task` A `Task` object corresponding to the to-be-assigned task.

Returns: Invisibly returns `TRUE` or `FALSE`, depending on whether the attempt was successful.

Method `try_start()`**:** Attempt to start the task. This method checks to see if the that worker has an assigned task, and if so starts it running within the R session. It also registers the change of status within the `Task` object itself. This method is intended to be called by a `WorkerPool` or a `Queue`.

Usage:

```
Worker$try_start()
```

Returns: Invisibly returns `TRUE` or `FALSE`, depending on whether the attempt was successful.

Method `try_finish()`**:** Attempt to finish a running task politely. This method checks to see if the worker has a running task, and if so polls the R session to determine if the R process claims to be ready to return. If there is a ready-to-return task the results are read from the R process and returned to the `Task` object. The task status is updated, and then unassigned from the `Worker`. This method is intended to be called by a `WorkerPool` or a `Queue`.

Usage:

```
Worker$try_finish(timeout = 0)
```

Arguments:

`timeout` Length of time to wait when process is polled (default = 0)

Returns: Invisibly returns `TRUE` or `FALSE`, depending on whether the attempt was successful.

Method `shutdown_worker()`**:** Attempt to shut down the R session gracefully, after making an attempt to salvage any task that the worker believes it has been assigned. The salvage operation depends on the state of the task. If the `Task` has been assigned but not started, the `Worker` will return it to a "waiting" state in the hope that the `Queue` will assign it to another worker later, and unassign it. If the `Task` is running, the `Worker` will attempt to read from the R session and then register the `Task` as "done" regardless of the outcome. (The reason for this is to ensure that tasks that crash or freeze the R session don't get returned to the `Queue`).

Usage:

```
Worker$shutdown_worker(grace = 1000)
```

Arguments:

`grace` Grace period in milliseconds. If the process is still running after this period, it will be killed.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Worker$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`WorkerPool` *R6 Class Representing a Worker Pool*

Description

A `WorkerPool` manages multiple workers.

Details

The implementation for a `WorkerPool` is essentially a container that holds one or more `Worker` objects, and posesses methods that allow it to instruct them to assign, start, and complete `Tasks`. It can also check to see if any of the R sessions associated with the `Workers` have crashed or stalled, and replace them as needed.

Methods

Public methods:

- `WorkerPool$new()`
- `WorkerPool$get_pool_worker()`
- `WorkerPool$get_pool_state()`
- `WorkerPool$try_assign()`
- `WorkerPool$try_start()`
- `WorkerPool$try_finish()`
- `WorkerPool$refill_pool()`
- `WorkerPool$shutdown_pool()`
- `WorkerPool$shutdown_overdue_workers()`
- `WorkerPool$clone()`

Method `new()`**:** Create a new worker pool

Usage:

```
WorkerPool$new(workers = 4L)
```

Arguments:

`workers` The number of workers in the pool.

Returns: A new `WorkerPool` object.

Method `get_pool_worker()`**:** Return a specific `Worker`

Usage:

```
WorkerPool$get_pool_worker(x)
```

Arguments:

`x` An integer specifying the index of the worker in the pool.

Returns: The corresponding `Worker` object.

Method `get_pool_state()`**:** Return a summary of the worker pool

Usage:

```
WorkerPool$get_pool_state()
```

Returns: A named character vector specifying the current state of each worker ("starting", "idle", "busy", or "finished"). Names denote worker ids, and the interpretations of each return value is as follows:

- `"starting"`: the R session is starting up.
- `"idle"`: the R session is ready to compute.
- `"busy"`: the R session is computing.
- `"finished"`: the R session has terminated.

Method `try_assign()`**:** Attempt to assign tasks to workers. This method is intended to be called by `Queue` objects. When called, this method will iterate over tasks in the list and workers in the pool, assigning tasks to workers as long as there are both idle workers and waiting tasks. It returns once it runs out of one resource or the other. Note that this method assigns tasks to workers: it does not instruct the workers to to start working on the tasks. That is the job of `try_start()`.

Usage:

```
WorkerPool$try_assign(tasks)
```

Arguments:

`tasks` A `TaskList` object

Returns: Invisibly returns `NULL`

Method `try_start()`**:** Iterates over `Workers` in the pool and asks them to start any jobs that the have been assigned but have not yet started. This method is intended to be called by `Queue` objects.

Usage:

```
WorkerPool$try_start()
```

Returns: Invisibly returns `NULL`

Method `try_finish()`**:** Iterate over `Workers` in the pool and checks to see if any of the busy sessions are ready to return results. For those that are, it finishes the tasks and ensures those results are returned to the `Task` object. This method is intended to be called by `Queue` objects.

Usage:

```
WorkerPool$try_finish()
```

Returns: Invisibly returns `NULL`

Method `refill_pool()`**:** Check the `WorkerPool` looking for `Workers` that have crashed or been shutdown, and replace them with fresh workers.

Usage:

```
WorkerPool$refill_pool()
```

Returns: This function is called primarily for its side effect. It returns a named character documenting the outcome, indicating the current state of each worker: should not be "finished" for any worker. Names denote worker ids.

Method `shutdown_pool()`**:** Terminate all workers in the pool.

Usage:

```
WorkerPool$shutdown_pool(grace = 1000)
```

Arguments:

`grace` Grace period in milliseconds. If a worker process is still running after this period, it will be killed.

Returns: This function is called primarily for its side effect. It returns a named character documenting the outcome, indicating the current state of each worker: should be "finished" for all workers. Names denote worker ids.

Method `shutdown_overdue_workers()`**:** Terminate workers that have worked on their current task for longer than a pre-specified time limit.

Usage:

```
WorkerPool$shutdown_overdue_workers(timelimit, grace = 1000)
```

Arguments:

`timelimit` Pre-specified time limit for the task, in seconds.

`grace` Grace period for the shutdown, in milliseconds. If a worker process is still running after this period, it will be killed.

Returns: This function is called primarily for its side effect. It returns a named character documenting the outcome, indicating the current state of each worker: should be "finished" for all workers. Names denote worker ids.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
WorkerPool$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Index