Package ‘jobqueue’

April 24, 2015

Type Package

Title Generic Asynchronous Job Queue

Version 1.0-4

Date 2015-04-22

Author Bastian Laubner [aut, cre]

Maintainer Bastian Laubner <`bastian.laubner@bramblecloud.com`>

URL `https://www.bramblecloud.com`

Depends R (>= 2.14.0)

Imports parallel

Description
A general-purpose job queue based on a 1-node socket cluster as provided by the parallel package.

License MIT + file LICENSE

R topics documented:

`jobqueue-package` *Job queue for computations in a separate thread*

Description

A generic implementation of an asynchronous job queue based on a 1-node-SOCKcluster from the parallel package.

Details

This package provides a queue-like interface that allows background computations while the calling R session remains responsive. The focus of the jobqueue package is on providing a simple and intuitive user interface.

Job results are available for pick-up in the order that the jobs were sent. Alternatively, jobs can be tagged and results can be retrieved based on their tag.

Job queues are created using `Q.make`. Jobs are typically pushed to the queue using `Q.push` and their results collected with `Q.collect` or `Q.pop`.

Note that queues have their own workspace which is separate from your current workspace. That means that you have to load any packages and set all variables that you need for your computations in the queue (see examples below).

Author(s)

Bastian Laubner <bastian.laubner@bramblecloud.com>

References

www.bramblecloud.com uses the jobqueue package for basic threading.

See Also

`Q.make`, `Q.push`, `Q.collect`

Examples

```
# create job queue
Q = Q.make()

# push computation job to queue
Q.push(Q, sum(1:1e4) )

# perform other computations here while the job is computed
# in the background

# retrieve result from job queue
Q.pop(Q)

# load package in the queue
Q.push(Q, library(stats4), mute=TRUE)

# set variables in the queue - you can also use Q.push or Q.sync
Q.assign(Q, "a", 5)

# close the queue and free its resources
Q.close(Q)
```

`Q.collect` *Functions for retrieving results from a job queue*

Description

`Q.collect` returns the queue's first available result in a standardized format. `Q.pop` is a simplified version of Q.collect that returns the pure result without any further information. `Q.collect.all` and `Q.pop.all` return all results that are currently available in the queue. `Q.isready` returns `TRUE` if there is a result available in the queue, `FALSE` otherwise.

Usage

```
Q.collect(Q, tag = NULL, wait = TRUE)
Q.pop(Q, tag = NULL, wait = TRUE)

Q.collect.all(Q, tag = NULL)
Q.pop.all(Q, tag = NULL)

Q.isready(Q, write = FALSE, timeout = 0)
```

Arguments

`Q`	A job queue as created by `Q.make`.
`tag`	Setting `tag` will cause `Q.collect` or `Q.pop` to return the first available result with the corresponding tag. Setting `tag` for `Q.collect.all` or `Q.pop.all` will return all available results with that tag.
`wait`	Maximum total time to wait until a result becomes available (in seconds). Logical values `TRUE`/`FALSE` are coerced to `1`/`0`, respectively. Setting `wait=Inf` will hang if no result ever becomes available.
`write`	Set `write=TRUE` to test if available for writing. Queue should always be available for writing. `write=FALSE` tests if there is a result available in the queue.
`timeout`	Time to wait (in seconds) for the socket to respond.

Details

`Q.collect` returns a list containing the result at entry `$value`. The return value always contains a boolean entry `$has.result` which is `FALSE` if there was no result to be read (either because queue has not been tasked with anything or no result is ready for pick-up yet), `TRUE` otherwise. If a result could be read from the socket, then the return value contains the entry `$success` which is `FALSE` if the computation produced an error. Furthermore, if the job was sent with values for tag and tranche, then the result contains these values at entries `$tag` and `$tranche`.

`Q.collect` discards muted results and returns the first available non-muted result. Setting `tag` will cause `Q.collect` to return the first available result with the corresponding tag. If no result with this tag becomes available within the waiting time, then the result's `has.result` field is `FALSE`. If you set a tag, make sure wait is not set to `0`/`FALSE` to give the queue time to sieve through the available results.

`Q.pop` is a simple version of `Q.collect` that only returns the result (without `has.result`, `tag`, `tranche`, etc.). Note: a `NULL` return value may either mean that the return value was `NULL` or that there was no result available at this time. Use `Q.collect` if you need to tell the difference.

`Q.collect.all` and `Q.pop.all` return all results that are currently available in the queue. There is no `wait` parameter as this would be practically equivalent to `Sys.sleep(wait)` followed by `Q.collect.all(Q)`.

There is usually no need to use `Q.isready` since the field `has.result` in `Q.collect`'s return value indicates whether the queue was ready or not.

Value

`Q.collect` returns a list which is guaranteed to have a logical field `has.result`. If `has.result=TRUE`, then the result also has a logical field `success` indicating whether or not an error occurred, and a field `value` containing the actual job result.

`Q.pop` simply returns the `value` field of `Q.collect`.

`Q.collect.all` and `Q.pop.all` return lists of elements in the respective formats.

`Q.isready` returns logical.

Note

If tag is specified for any of the retrieval functions, then available results with different tags will still be retrieved from the socket but stored in `Q$data$rack` for later retrieval from the queue. This is handled automatically by the queue.

Author(s)

Bastian Laubner

See Also

`Q.make`, as well as `Q.push` and `jobqueue` for further examples.

Examples

```
Q = Q.make()
Q.push(Q, "Some job")

# safe way to check if the job returned a value
job = Q.collect(Q)
if (job$has.result) {
  # do something with the result
  print(job$value)
}

# safe way with error checking
if (job$has.result) {
  if (job$success) {
    # do something with the job result
    print(job$value)
  } else {
    # an error occurred during processing, error message in job$value
    stop(job$value)
  }
}

Q.close(Q)
```

`Q.make` *Functions for creating and closing a job queue*

Description

`Q.make` creates a 1-node socket cluster to be used with all the other jobqueue functions. `Q.close` destroys the jobqueue and frees its resources, deleting all data which it may still contain.

Usage

```
Q.make(...)
Q.close(Q, cleanup = TRUE)
```

Arguments

`...`	`Q.make` passes all supplied arguments through to `parallel::makeCluster(1, "PSOCK", ...)`. Usually, no such options need to be specified. Setting `methods=FALSE` reduces the memory usage of the queue, but can only be used if the methods package and S4 classes are not needed.
`Q`	A job queue as created by `Q.make`.
`cleanup`	If `TRUE`, the queue's temporary data and all variables assigned with `Q.assign` are deleted before closing the socket. Otherwise, only the socket is closed (potentially not freeing all resources).

Value

`Q.make` returns an S3 jobqueue object to be used with the other jobqueue functions.

`Q.close` returns NULL invisibly.

Author(s)

Bastian Laubner

See Also

`Q.push`, `Q.collect`

Examples

```
Q = Q.make()
Q.close(Q)
```

`Q.push` *Functions for sending jobs to a job queue*

Description

`Q.push` is the standard function for enqueing jobs. `Q.do` is a convenience function that enqueues a job and immediately tries to pick up its result. `Q.assign` is used for assigning local values to variables in the queue's workspace. `Q.sync` is a convenience function for copying local variables 1-to-1 to the queue's workspace. `Q.do.call` is a convenience function for executing functions in the queue and storing their result in the queue's workspace.

Usage

```
Q.push(Q, ..., local = list(), tag = NULL, tranche = NULL, mute = FALSE)
Q.do(Q, ..., local = list(), tag = NULL, tranche = NULL, mute = FALSE, wait = TRUE)
Q.assign(Q, var, value, envir = globalenv())
Q.sync(Q, ...)
Q.do.call(Q, fun, args, var = NULL, envir = globalenv(),
          tag = NULL, tranche = NULL, mute = FALSE)
```

Arguments

`Q`	A job queue as created by `Q.make`.
`...`	For `Q.push` and `Q.do`: an R expression which is executed in the queue's workspace as it is, after substituting the local values of the elements specified in `local`. Use {} to enclose multiple commands/lines. For `Q.sync`: unquoted variable names.
`local`	A named list of the form `list(var1=value1, var2=value2, ...)`. Variables `var1, var2, ...` in the expression supplied to `Q.push` will be replaced with `value1, value2, ...`, respectively (evaluated in the local workspace), before sending the expression for evaluation to the queue.
`tag`	A tag for the job that can be used for later retrieval. Values will be coerced to string using `as.character`.
`tranche`	Another job tag that makes it convenient to subdivide a job with one tag into several parts. Can be an arbitrary value.
`mute`	If set to `TRUE`, the queue does not return the result of the computation.
`wait`	Maximum total time to wait until a result becomes available (in seconds). See `Q.pop` for more.
`var`	Variable to remotely assign in the queue's workspace (given as quoted name).
`value`	Value to assign (evaluated locally).
`envir`	Environment to assign to in the queue's workspace (evaluated remotely).
`fun`	Function from the queue's workspace to execute (given as quoted name).
`args`	Function arguments given as a list (evaluated locally).

Details

`Q.push` is the standard function used to send jobs to the queue. As the queue is running in a separate thread (via Rscript), it also has a workspace which is different from the local session (and which is practically blank in the beginning). The expression given to `Q.push` is evaluated remotely in the queue's global environment, so `Q.push(Q, a+1)` adds 1 to the value of `a` from the queue's workspace. If `a` cannot be found there, an error is thrown.

Values from the local workspace can be injected into the expression using `local`. For example, `Q.push(Q, a+1, local=list(a=b))` uses the value of local variable `b` in place of `a`. Here, `b` can also be any other expression which is evaluated locally.

`Q.do` performs a `Q.push` command, immediately followed by a `Q.pop`. Its primary use is as a convenience function in interactive sessions for quickly evaluated jobs. If the return value is irrelevant, it is better to use `Q.push` with `mute=TRUE`. You should be sure there are no other jobs in the queue that could prevent the timely pickup (otherwise use a unique tag as in the example below to prevent interference, and consider increasing `wait`).

`Q.assign` is a convenience function for assigning a local value to a remote variable. The assignment is done to the global environment unless envir is specified (given as an unquoted environment variable which is evaluated remotely). The queue's reply for this assignment operation is muted. To check if the assignment was successful check the value returned for `Q.push(queue, var)` (here var is without quotes). The assignment operation is a normal task in the queue's pipeline and is executed only once all previous tasks were completed. `Q.assign(Q, "var", value)` is equivalent to `Q.push(Q, var <- value, local=list(value=value))`.

`Q.sync` is another simplification of `Q.assign`. The specified variables (unquoted) are assigned under the same name to the queue's global environment. Local variable values are obtained as if the variable was used at the prompt.

`Q.do.call` executes the remote function fun (given as quoted name) with local arguments args and assigns result to remote variable var (given as quoted name), if specified. The evaluation result is also stored for retrieval as with `Q.push` (unless `mute=TRUE`). Use `quote(variable)` in args-list for variables or expressions whose remote value should be used. NOTE: unlike `Q.do`, this function does not attempt to pick up the computed result immediately (the name `Q.do.call` imitates the base function `do.call`).

Value

`Q.push` and `Q.do.call` return NULL invisibly.

`Q.do` has the same return value as `Q.pop`.

`Q.assign` returns the (locally evaluated) value which was sent to the queue for assignment.

`Q.sync` invisibly returns the character vector of variable names sent for assignment.

Note

The string `Q.mute` is reserved by the queue to tag jobs whose result should be muted. The list `Q$data$mute` can be extended to include further tags that should be muted.

Author(s)

Bastian Laubner

See Also

`Q.make`, `Q.collect`

Examples

```
Q = Q.make()

Q.push(Q, sum(1:1e4))
Q.push(Q, {a = 1})  # braces needed for assignment with =, not needed for a <- 1
Q.assign(Q, "b", 2)
c = 3
Q.sync(Q,c)
Q.do.call(Q, "sum", list(1:3), "d")
Q.do(Q, a+b+c == d, tag="test")  # TRUE - note: have to use a unique tag here
Q.pop.all(Q)

# use tags so that jobs can be retrieved selectively
for (i in 1:10) Q.push(Q, paste("Result:", i), local=list(i=i), tag=i);
Q.pop(Q, tag=5)  # Result: 5
Q.pop(Q, tag=2)  # Result: 2
Q.pop(Q, tag=10)  # Result: 10
Q.pop.all(Q)

Q.close(Q)
```

Index