Package 'callr'

April 11, 2018

Title Call R from R

Version 2.0.3

Author Gábor Csárdi, Winston Chang

Maintainer Gábor Csárdi <csardi.gabor@gmail.com>

Description It is sometimes useful to perform a computation in a separate R process, without affecting the current R process at all. This packages does exactly that.

License MIT + file LICENSE

LazyData true

URL https://github.com/r-lib/callr#readme

BugReports https://github.com/r-lib/callr/issues

RoxygenNote 6.0.1.9000

Imports assertthat, crayon, debugme, R6, utils

Suggests covr, parallel, testthat, withr

LinkingTo testthat

Encoding UTF-8

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-04-11 03:50:18 UTC

R topics documented:

`callr` *Call R from R*

Description

It is sometimes useful to perform a computation in a separate R process, without affecting the current R process at all. This packages does exactly that.

`poll` *Poll for process I/O or termination*

Description

Wait until one of the specified processes produce standard output or error, terminates, or a timeout occurs.

Usage

```
poll(processes, ms)
```

Arguments

`processes`	A list of `process` objects to wait on. If this is a named list, then the returned list will have the same names. This simplifies the identification of the processes. If an empty list, then the
`ms`	Integer scalar, a timeout for the polling, in milliseconds. Supply -1 for an infitite timeout, and 0 for not waiting at all.

Value

A list of character vectors of length two. There is one list element for each process, in the same order as in the input list. The character vectors' elements are named `output` and `error` and their possible values are: `nopipe`, `ready`, `timeout`, `closed`, `silent`. See details about these below.

Explanation of the return values

- `nopipe` means that the stdout or stderr from this process was not captured.
- `ready` means that stdout or stderr from this process are ready to read from. Note that end-of-file on these outputs also triggers `ready`.
- timeout‘: the processes are not ready to read from and a timeout happened.
- `closed`: the connection was already closed, before the polling started.
- `silent`: the connection is not ready to read from, but another connection was.

Known issues

`poll()` cannot wait on the termination of a process directly. It is only signalled through the closed stdout and stderr pipes. This means that if both stdout and stderr are ignored or closed for a process, then you will not be notified when it exits. If you want to wait for just a single process to end, it can be done with the `$wait()` method.

Examples

```
## Different commands to run for windows and unix
## Not run:
cmd1 <- switch(
  .Platform$OS.type,
  "unix" = c("sh", "-c", "sleep 1; ls"),
  c("cmd", "/c", "ping -n 2 127.0.0.1 && dir /b")
)
cmd2 <- switch(
  .Platform$OS.type,
  "unix" = c("sh", "-c", "sleep 2; ls 1>&2"),
  c("cmd", "/c", "ping -n 2 127.0.0.1 && dir /b 1>&2")
)

## Run them. p1 writes to stdout, p2 to stderr, after some sleep
p1 <- process$new(cmd1[1], cmd1[-1], stdout = "|")
p2 <- process$new(cmd2[1], cmd2[-1], stderr = "|")

## Nothing to read initially
poll(list(p1 = p1, p2 = p2), 0)

## Wait until p1 finishes. Now p1 has some output
p1$wait()
poll(list(p1 = p1, p2 = p2), -1)

## Close p1's connection, p2 will have output on stderr, eventually
close(p1$get_output_connection())
poll(list(p1 = p1, p2 = p2), -1)

## Close p2's connection as well, no nothing to poll
close(p2$get_error_connection())
poll(list(p1 = p1, p2 = p2), 0)

## End(Not run)
```

process *External process*

Description

Managing external processes from R is not trivial, and this class aims to help with this deficiency. It is essentially a small wrapper around the `system` base R function, to return the process id of the started process, and set its standard output and error streams. The process id is then used to manage the process.

Usage

```
p <- process$new(command = NULL, args,
                 stdout = NULL, stderr = NULL, cleanup = TRUE,
                 echo_cmd = FALSE, supervise = FALSE,
                 windows_verbatim_args = FALSE,
                 windows_hide_window = FALSE,
                 encoding = "", post_process = NULL)

p$is_alive()
p$signal(signal)
p$kill(grace = 0.1)
p$wait(timeout = -1)
p$get_pid()
p$get_exit_status()
p$restart()
p$get_start_time()

p$read_output(n = -1)
p$read_error(n = -1)
p$read_output_lines(n = -1)
p$read_error_lines(n = -1)
p$get_output_connection()
p$get_error_connection()
p$is_incomplete_output()
p$is_incomplete_error()
p$read_all_output()
p$read_all_error()
p$read_all_output_lines()
p$read_all_error_lines()

p$poll_io(timeout)

p$get_result()

print(p)
```

Arguments

- p: process object.
- command: Character scalar, the command to run. Note that this argument is not passed to a shell, so no tilde-expansion or variable substitution is performed on it. It should not be quoted with base::shQuote(). See base::normalizePath() for tilde-expansion.
- args: Character vector, arguments to the command. They will be used as is, without a shell. They don't need to be escaped.
- stdout: What to do with the standard output. Possible values: NULL: discard it; a string, redirect it to this file; "|": create a connection for it.
- stderr: What to do with the standard error. Possible values: NULL: discard it; a string, redirect it to this file; "|": create a connection for it.
- cleanup: Whether to kill the process (and its children) if the process object is garbage collected.
- echo_cmd: Whether to print the command to the screen before running it.
- supervise: Whether to register the process with a supervisor. If TRUE, the supervisor will ensure that the process is killed when the R process exits.
- windows_verbatim_args: Whether to omit quoting the arguments on Windows. It is ignored on other platforms.
- windows_hide_window: Whether to hide the application's window on Windows. It is ignored on other platforms.
- signal: An integer scalar, the id of the signal to send to the process. See tools::pskill() for the list of signals.
- grace: Currently not used.
- timeout: Timeout in milliseconds, for the wait or the I/O polling.
- n: Number of characters or lines to read.
- encoding: The encoding to assume for stdout and stderr. By default the encoding of the current locale is used. Note that callr always reencodes the output of both streams in UTF-8 currently. If you want to read them without any conversion, on all platforms, specify "UTF-8" as encoding.
- post_process: An optional function to run when the process has finished. Currently it only runs if $get_result() is called. It is only run once.

Details

$new() starts a new process in the background, and then returns immediately.

$is_alive() checks if the process is alive. Returns a logical scalar.

$signal() sends a signal to the process. On Windows only the SIGINT, SIGTERM and SIGKILL signals are interpreted, and the special 0 signal, The first three all kill the process. The 0 signal return TRUE if the process is alive, and FALSE otherwise. On Unix all signals are supported that the OS supports, and the 0 signal as well.

$kill() kills the process. It also kills all of its child processes, except if they have created a new process group (on Unix), or job object (on Windows). It returns TRUE if the process was killed, and FALSE if it was no killed (because it was already finished/dead when callr tried to kill it).

`$wait()` waits until the process finishes, or a timeout happens. Note that if the process never finishes, and the timeout is infinite (the default), then R will never regain control. It returns the process itself, invisibly. In some rare cases, `$wait()` might take a bit longer than specified to time out. This happens on Unix, when another package overwrites the processx SIGCHLD signal handler, after the processx process has started. One such package is parallel, if used with fork clusters, e.g. through `parallel::mcparallel()`.

`$get_pid()` returns the process id of the process.

`$get_exit_status` returns the exit code of the process if it has finished and `NULL` otherwise. On Unix, in some rare cases, the exit status might be `NA`. This happens if another package (or R itself) overwrites the processx SIGCHLD handler, after the processx process has started. In these cases processx cannot determine the real exit status of the process. One such package is parallel, if used with fork clusters, e.g. through the `parallel::mcparallel()` function.

`$restart()` restarts a process. It returns the process itself.

`$get_start_time()` returns the time when the process was started.

`$is_supervised()` returns whether the process is being tracked by supervisor process.

`$supervise()` if passed `TRUE`, tells the supervisor to start tracking the process. If `FALSE`, tells the supervisor to stop tracking the process. Note that even if the supervisor is disabled for a process, if it was started with `cleanup=TRUE`, the process will still be killed when the object is garbage collected.

`$read_output()` reads from the standard output connection of the process. If the standard output connection was not requested, then then it returns an error. It uses a non-blocking text connection. This will work only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_error()` is similar to `$read_output`, but it reads from the standard error stream.

`$read_output_lines()` reads lines from standard output connection of the process. If the standard output connection was not requested, then then it returns an error. It uses a non-blocking text connection. This will work only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_error_lines()` is similar to `$read_output_lines`, but it reads from the standard error stream.

`$has_output_connection()` returns `TRUE` if there is a connection object for standard output; in other words, if `stdout="|"`. It returns `FALSE` otherwise.

`$has_error_connection()` returns `TRUE` if there is a connection object for standard error; in other words, if `stderr="|"`. It returns `FALSE` otherwise.

`$get_output_connection()` returns a connection object, to the standard output stream of the process.

`$get_error_conneciton()` returns a connection object, to the standard error stream of the process.

`$is_incomplete_output()` return `FALSE` if the other end of the standard output connection was closed (most probably because the process exited). It return `TRUE` otherwise.

`$is_incomplete_error()` return `FALSE` if the other end of the standard error connection was closed (most probably because the process exited). It return `TRUE` otherwise.

`$read_all_output()` waits for all standard output from the process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character scalar. This will return content only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_all_error()` waits for all standard error from the process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character scalar. This will return content only if `stderr="|"` was used. Otherwise, it will throw an error.

`$read_all_output_lines()` waits for all standard output lines from a process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character vector. This will return content only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_all_error_lines()` waits for all standard error lines from a process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character vector. This will return content only if `stderr="|"` was used. Otherwise, it will throw an error.

`$get_output_file()` if the `stdout` argument was a filename, this returns the absolute path to the file. If `stdout` was `"|"` or `NULL`, this simply returns that value.

`$get_error_file()` if the `stderr` argument was a filename, this returns the absolute path to the file. If `stderr` was `"|"` or `NULL`, this simply returns that value.

`$poll_io()` polls the process's connections for I/O. See more in the *Polling* section, and see also the `poll()` function to poll on multiple processes.

`$get_result()` returns the result of the post processesing function. It can only be called once the process has finished. If the process has no post-processing function, then `NULL` is returned.

`print(p)` or `p$print()` shows some information about the process on the screen, whether it is running and it's process id, etc.

Polling

The `poll_io()` function polls the standard output and standard error connections of a process, with a timeout. If there is output in either of them, or they are closed (e.g. because the process exits) `poll_io()` returns immediately.

In addition to polling a single process, the `poll()` function can poll the output of several processes, and returns as soon as any of them has generated output (or exited).

Examples

```
# CRAN does not like long-running examples
## Not run:
p <- process$new("sleep", "2")
p$is_alive()
p
p$kill()
p$is_alive()

p$restart()
p$is_alive()
Sys.sleep(3)
p$is_alive()

## End(Not run)
```

r *Evaluate an expression in another R session*

Description

From `callr` version 2.0.0, `r()` is equivalent to `r_safe()`, and tries to set up a less error prone execution environment. In particular:

- It makes sure that at least one reasonable CRAN mirror is set up.
- Adds some command line arguments are added to avoid saving `.RData` files, etc.
- Ignores the system and user profiles.
- Various environment variables are set: `CYGWIN` to avoid warnings about DOS-style paths, `R_TESTS` to avoid issues when `callr` is invoked from unit tests, `R_BROWSER` and `R_PDFVIEWER` to avoid starting a browser or a PDF viewer. See `rcmd_safe_env()`.

Usage

```
r(func, args = list(), libpath = .libPaths(),
  repos = c(getOption("repos"), c(CRAN = "https://cloud.r-project.org")),
  stdout = NULL, stderr = NULL, error = getOption("callr.error", "error"),
  cmdargs = c("--no-site-file", "--no-environ", "--slave", "--no-save",
  "--no-restore"), show = FALSE, callback = NULL, block_callback = NULL,
  spinner = show && interactive(), system_profile = FALSE,
  user_profile = FALSE, env = rcmd_safe_env(), timeout = Inf)

r_safe(func, args = list(), libpath = .libPaths(),
  repos = c(getOption("repos"), c(CRAN = "https://cloud.r-project.org")),
  stdout = NULL, stderr = NULL, error = getOption("callr.error", "error"),
  cmdargs = c("--no-site-file", "--no-environ", "--slave", "--no-save",
  "--no-restore"), show = FALSE, callback = NULL, block_callback = NULL,
  spinner = show && interactive(), system_profile = FALSE,
  user_profile = FALSE, env = rcmd_safe_env(), timeout = Inf)
```

Arguments

`func` Function object to call in the new R process. The function should be self-contained and only refer to other functions and use variables explicitly from other packages using the `::` notation. The environment of the function is set to `.GlobalEnv` before passing it to the child process. Because of this, it is good practice to create an anonymous function and pass that to `callr`, instead of passing a function object from a (base or other) package. In particular

`r(.libPaths)`

does not work, because `.libPaths` is defined in a special environment, but

`r(function() .libPaths())`

works just fine.

args	Arguments to pass to the function. Must be a list.
libpath	The library path.
repos	The *repos* option. If NULL, then no *repos* option is set. This options is only used if user_profile or system_profile is set FALSE, as it is set using the system or the user profile.
stdout	The name of the file the standard output of the child R process will be written to. If the child process runs with the --slave option (the default), then the commands are not echoed and will not be shown in the standard output. Also note that you need to call print() explicitly to show the output of the command(s).
stderr	The name of the file the standard error of the child R process will be written to. In particular message() sends output to the standard error. If nothing was sent to the standard error, then this file will be empty. This can be the same file as stderr, although there is no guarantee that the lines will be in the correct chronological order.
error	What to do if the remote process throws an error. See details below.
cmdargs	Command line arguments to pass to the R process. Note that c("-f", rscript) is appended to this, rscript is the name of the script file to run. This contains a call to the supplied function and some error handling code.
show	Logical, whether to show the standard output on the screen while the child process is running. Note that this is independent of the stdout and stderr arguments. The standard error is not shown currently.
callback	A function to call for each line of the standard output and standard error from the child process. It works together with the show option; i.e. if show = TRUE, and a callback is provided, then the output is shown of the screen, and the callback is also called.
block_callback	A function to call for each block of the standard output and standard error. This callback is not line oriented, i.e. multiple lines or half a line can be passed to the callback.
spinner	Whether to show a calming spinner on the screen while the child R session is running. By default it is shown if show = TRUE and the R session is interactive.
system_profile	Whether to use the system profile file.
user_profile	Whether to use the user's profile file.
env	Environment variables to set for the child process.
timeout	Timeout for the function call to finish. It can be a base::difftime object, or a real number, meaning seconds. If the process does not finish before the timeout period expires, then a system_command_timeout_error error is thrown. Inf means no timeout.

Details

The pre-2.0.0 r() function is called r_copycat() now.

Value

Value of the evaluated expression.

Error handling

callr handles errors properly. If the child process throws an error, then callr throws an error with the same error message in the parent process.

The error expert argument may be used to specify a different behavior on error. The following values are possible:

- error is the default behavior: throw an error in the parent, with the same error message. In fact the same error object is thrown again.
- stack also throws an error in the parent, but the error is of a special kind, class callr_error, and it contains both the original error object, and the call stack of the child, as written out by utils::dump.frames().
- debugger is similar to stack, but in addition to returning the complete call stack, it also start up a debugger in the child call stack, via utils::debugger().

The default error behavior can be also set using the callr.error option. This is useful to debug code that uses callr.

See Also

Other callr functions: r_copycat, r_vanilla

Examples

```
## Not run:
# Workspace is empty
r(function() ls())

# library path is the same by default
r(function() .libPaths())
.libPaths()

## End(Not run)
```

---

rcmd *Run an R CMD command*

Description

Run an R CMD command form within R. This will usually start another R process, from a shell script.

Usage

```
rcmd(cmd, cmdargs = character(), libpath = .libPaths(),
  repos = c(getOption("repos"), c(CRAN = "https://cloud.r-project.org")),
  stdout = NULL, stderr = NULL, echo = FALSE, show = FALSE,
  callback = NULL, block_callback = NULL, spinner = show && interactive(),
  system_profile = FALSE, user_profile = FALSE, env = rcmd_safe_env(),
  timeout = Inf, wd = ".", fail_on_status = FALSE)

rcmd_safe(cmd, cmdargs = character(), libpath = .libPaths(),
  repos = c(getOption("repos"), c(CRAN = "https://cloud.r-project.org")),
  stdout = NULL, stderr = NULL, echo = FALSE, show = FALSE,
  callback = NULL, block_callback = NULL, spinner = show && interactive(),
  system_profile = FALSE, user_profile = FALSE, env = rcmd_safe_env(),
  timeout = Inf, wd = ".", fail_on_status = FALSE)
```

Arguments

cmd	Command to run. See `R --help` from the command line for the various commands. In the current version of R (3.2.4) these are: BATCH, COMPILE, SHLIB, INSTALL, REMOVE, build, check, LINK, Rprof, Rdconv, Rd2pdf, Rd2txt, Stangle, Sweave, Rdiff, config, javareconf, rtags.
cmdargs	Command line arguments.
libpath	The library path.
repos	The *repos* option. If `NULL`, then no *repos* option is set. This options is only used if `user_profile` or `system_profile` is set `FALSE`, as it is set using the system or the user profile.
stdout	Optionally a file name to send the standard output to.
stderr	Optionally a file name to send the standard error to.
echo	Whether to echo the complete command run by `rcmd`.
show	Logical, whether to show the standard output on the screen while the child process is running. Note that this is independent of the `stdout` and `stderr` arguments. The standard error is not shown currently.
callback	A function to call for each line of the standard output and standard error from the child process. It works together with the `show` option; i.e. if `show = TRUE`, and a callback is provided, then the output is shown of the screen, and the callback is also called.
block_callback	A function to call for each block of the standard output and standard error. This callback is not line oriented, i.e. multiple lines or half a line can be passed to the callback.
spinner	Whether to show a calming spinner on the screen while the child R session is running. By default it is shown if `show = TRUE` and the R session is interactive.
system_profile	Whether to use the system profile file.
user_profile	Whether to use the user's profile file.
env	Environment variables to set for the child process.

timeout	Timeout for the function call to finish. It can be a base::difftime object, or a real number, meaning seconds. If the process does not finish before the timeout period expires, then a system_command_timeout_error error is thrown. Inf means no timeout.
wd	Working directory to use for running the command. Defaults to the current working directory.
fail_on_status	Whether to throw an R error if the command returns with a non-zero status code. By default no error is thrown.

Details

Starting from callr 2.0.0, rcmd() has safer defaults, the same as the rcmd_safe() default values. Use rcmd_copycat() for the old defaults.

Value

A list with the command line $command), standard output ($stdout), standard error ($stderr), exit status ($status) of the external R CMD command, and whether a timeout was reached ($timeout).

See Also

Other R CMD commands: rcmd_bg, rcmd_copycat

Examples

```
## Not run:
rcmd("config", "CC")

## End(Not run)
```

---

rcmd_bg *Run an R CMD command in the background*

Description

The child process is started in the background, and the function return immediately.

Usage

```
rcmd_bg(cmd, cmdargs = character(), libpath = .libPaths(), stdout = "|",
  stderr = "|", repos = c(getOption("repos"), c(CRAN =
  "https://cloud.r-project.org")), system_profile = FALSE,
  user_profile = FALSE, env = rcmd_safe_env(), wd = ".",
  supervise = FALSE)
```

Arguments

cmd	Command to run. See `R --help` from the command line for the various commands. In the current version of R (3.2.4) these are: BATCH, COMPILE, SHLIB, INSTALL, REMOVE, build, check, LINK, Rprof, Rdconv, Rd2pdf, Rd2txt, Stangle, Sweave, Rdiff, config, javareconf, rtags.
cmdargs	Command line arguments.
libpath	The library path.
stdout	Optionally a file name to send the standard output to.
stderr	Optionally a file name to send the standard error to.
repos	The *repos* option. If `NULL`, then no *repos* option is set. This options is only used if `user_profile` or `system_profile` is set `FALSE`, as it is set using the system or the user profile.
system_profile	Whether to use the system profile file.
user_profile	Whether to use the user's profile file.
env	Environment variables to set for the child process.
wd	Working directory to use for running the command. Defaults to the current working directory.
supervise	Whether to register the process with a supervisor. If `TRUE`, the supervisor will ensure that the process is killed when the R process exits.

Value

It returns a process object.

See Also

Other R CMD commands: `rcmd_copycat`, `rcmd`

---

rcmd_copycat *Call and R CMD command, while mimicking the current R session*

Description

This function is similar to `rcmd()`, but it has slightly different defaults:

- The `repos` options is unchanged.
- No extra environment variables are defined.

Usage

```
rcmd_copycat(cmd, cmdargs = character(), libpath = .libPaths(),
  repos = getOption("repos"), env = character(), ...)
```

Arguments

cmd	Command to run. See `R --help` from the command line for the various commands. In the current version of R (3.2.4) these are: BATCH, COMPILE, SHLIB, INSTALL, REMOVE, build, check, LINK, Rprof, Rdconv, Rd2pdf, Rd2txt, Stangle, Sweave, Rdiff, config, javareconf, rtags.
cmdargs	Command line arguments.
libpath	The library path.
repos	The *repos* option. If `NULL`, then no *repos* option is set. This options is only used if `user_profile` or `system_profile` is set `FALSE`, as it is set using the system or the user profile.
env	Environment variables to set for the child process.
...	Additional arguments are passed to `rcmd()`.

See Also

Other R CMD commands: `rcmd_bg`, `rcmd`

---

rcmd_process *External R CMD Process*

Description

An `R CMD *` command that runs in the background. This is an R6 class that extends the process class.

Usage

```
rp <- rcmd_process$new(options)
```

Arguments

- `options` A list of options created via `rcmd_process_options()`.

Details

`rcmd_process$new` creates a new instance. Its `options` argument is best created by the `r_process_options()` function.

Examples

```
## Not run:
options <- rcmd_process_options(cmd = "config", cmdargs = "CC")
rp <- rcmd_process$new(options)
rp$wait()
rp$read_output_lines()

## End(Not run)
```

rcmd_process_options *Create options for an rcmd_process object*

Description

Create options for an rcmd_process object

Usage

```
rcmd_process_options(...)
```

Arguments

...	Options to override, named arguments.

Value

A list of options.

`rcmd_process_options()` creates a set of options to initialize a new object from the `rcmd_process` class. Its arguments must be named, the names are used as option names. The options correspond to (some of) the arguments of the `rcmd()` function. At least the `cmd` option must be specified, to select the `R CMD` subcommand to run. Typically `cmdargs` is specified as well, to supply more arguments to `R CMD`.

Examples

```
## List all options and their default values:
rcmd_process_options()
```

rcmd_safe_env *rcmd_safe_env returns a set of environment variables that are more appropriate for rcmd_safe(). It is exported to allow manipulating these variables (e.g. add an extra one), before passing them to the rcmd() functions.*

Description

It currently has the following variables:

- `CYGWIN="nodosfilewarning"`: On Windows, do not warn about MS-DOS style file names.
- `R_TESTS=""` This variable is set by `R CMD check`, and makes the child R process load a startup file at startup, from the current working directory, that is assumed to be the `/test` dirctory of the package being checked. If the current working directory is changed to something else (as it typically is by `testthat`, then R cannot start. Setting it to the empty string ensures that `callr` can be used from unit tests.

- R_BROWSER="false": typically we don't want to start up a browser from the child R process.
- R_PDFVIEWER="false": similarly for the PDF viewer.
- R_ENVIRON_USER=tempfile(): this prevents R from loading the user .Renviron.

Usage

```
rcmd_safe_env()
```

Details

Note that callr also sets the R_LIBS, R_LIBS_USER, R_LIBS_SITE, R_PROFILE and R_PROFILE_USER environment variables appropriately, unless these are set by the user in the env argument of the r, etc. calls.

Value

A named character vector of environment variables.

---

run *Run external command, and wait until finishes*

Description

run provides an interface similar to base::system() and base::system2(), but based on the process class. This allows some extra features, see below.

Usage

```
run(command = NULL, args = character(), error_on_status = TRUE,
  echo_cmd = FALSE, echo = FALSE, spinner = FALSE, timeout = Inf,
  stdout_line_callback = NULL, stdout_callback = NULL,
  stderr_line_callback = NULL, stderr_callback = NULL,
  windows_verbatim_args = FALSE, windows_hide_window = FALSE,
  encoding = "")
```

Arguments

command	Character scalar, the command to run. It will be escaped via base::shQuote.
args	Character vector, arguments to the command. They will be escaped via base::shQuote.
error_on_status	Whether to throw an error if the command returns with a non-zero status, or it is interrupted. The error clases are system_command_status_error and system_command_timeout_error, respectively, and both errors have class system_command_error as well.
echo_cmd	Whether to print the command to run to the screen.

`echo` Whether to print the standard output and error to the screen. Note that the order of the standard output and error lines are not necessarily correct, as standard output is typically buffered.

`spinner` Whether to show a reassusing spinner while the process is running.

`timeout` Timeout for the process, in seconds, or as a `difftime` object. If it is not finished before this, it will be killed.

`stdout_line_callback`
`NULL`, or a function to call for every line of the standard output. See `stdout_callback` and also more below.

`stdout_callback`
`NULL`, or a function to call for every chunk of the standard output. A chunk can be as small as a single character. At most one of `stdout_line_callback` and `stdout_callback` can be non-`NULL`.

`stderr_line_callback`
`NULL`, or a function to call for every line of the standard error. See `stderr_callback` and also more below.

`stderr_callback`
`NULL`, or a function to call for every chunk of the standard error. A chunk can be as small as a single character. At most one of `stderr_line_callback` and `stderr_callback` can be non-`NULL`.

`windows_verbatim_args`
Whether to omit the escaping of the command and the arguments on windows. Ignored on other platforms.

`windows_hide_window`
Whether to hide the window of the application on windows. Ignored on other platforms.

`encoding` The encoding to assume for `stdout` and `stderr`. By default the encoding of the current locale is used. Note that `callr` always reencodes the output of both streams in UTF-8 currently.

Details

`run` supports

- Specifying a timeout for the command. If the specified time has passed, and the process is still running, it will be killed (with all its child processes).
- Calling a callback function for each line or each chunk of the standard output and/or error. A chunk may contain multiple lines, and can be as short as a single character.

Value

A list with components:

- status The exit status of the process. If this is `NA`, then the process was killed and had no exit status.
- stdout The standard output of the command, in a character scalar.
- stderr The standard error of the command, in a character scalar.
- timeout Whether the process was killed because of a timeout.

Callbacks

Some notes about the callback functions. The first argument of a callback function is a character scalar (length 1 character), a single output or error line. The second argument is always the process object. You can manipulate this object, for example you can call `$kill()` on it to terminate it, as a response to a message on the standard output or error.

Examples

```
## Different examples for Unix and Windows
## Not run:
if (.Platform$OS.type == "unix") {
  run("ls")
  system.time(run("sleep", "10", timeout = 1,
    error_on_status = FALSE))
  system.time(
    run(
      "sh", c("-c", "for i in 1 2 3 4 5; do echo $i; sleep 1; done"),
      timeout = 2, error_on_status = FALSE
    )
  )
} else {
  run("ping", c("-n", "1", "127.0.0.1"))
  run("ping", c("-n", "6", "127.0.0.1"), timeout = 1,
    error_on_status = FALSE)
}

## End(Not run)
```

---

r_bg *Evaluate an expression in another R session, in the background*

Description

Starts evaluating an R function call in a background R process, and returns immediately.

Usage

```
r_bg(func, args = list(), libpath = .libPaths(),
  repos = c(getOption("repos"), c(CRAN = "https://cloud.r-project.org")),
  stdout = "|", stderr = "|", error = getOption("callr.error", "error"),
  cmdargs = c("--no-site-file", "--no-environ", "--slave", "--no-save",
  "--no-restore"), system_profile = FALSE, user_profile = FALSE,
  env = rcmd_safe_env(), supervise = FALSE)
```

Arguments

func	Function object to call in the new R process. The function should be self-contained and only refer to other functions and use variables explicitly from other packages using the :: notation. The environment of the function is set to .GlobalEnv before passing it to the child process. Because of this, it is good practice to create an anonymous function and pass that to callr, instead of passing a function object from a (base or other) package. In particular
	
	`r(.libPaths)`
	
	does not work, because .libPaths is defined in a special environment, but
	
	`r(function() .libPaths())`
	
	works just fine.

args	Arguments to pass to the function. Must be a list.
libpath	The library path.
repos	The *repos* option. If NULL, then no *repos* option is set. This options is only used if user_profile or system_profile is set FALSE, as it is set using the system or the user profile.
stdout	The name of the file the standard output of the child R process will be written to. If the child process runs with the --slave option (the default), then the commands are not echoed and will not be shown in the standard output. Also note that you need to call print() explicitly to show the output of the command(s).
stderr	The name of the file the standard error of the child R process will be written to. In particular message() sends output to the standard error. If nothing was sent to the standard error, then this file will be empty. This can be the same file as stderr, although there is no guarantee that the lines will be in the correct chronological order.
error	What to do if the remote process throws an error. See details below.
cmdargs	Command line arguments to pass to the R process. Note that c("-f", rscript) is appended to this, rscript is the name of the script file to run. This contains a call to the supplied function and some error handling code.
system_profile	Whether to use the system profile file.
user_profile	Whether to use the user's profile file.
env	Environment variables to set for the child process.
supervise	Whether to register the process with a supervisor. If TRUE, the supervisor will ensure that the process is killed when the R process exits.

Value

An r_process object, which inherits from process, so all process methods can be called on it, and in addition it also has a get_result() method to collect the result.

Examples

```
## Not run:
rx <- r_bg(function() 1 + 2)

# wait until it is done
rx$wait()
rx$is_alive()
rx$get_result()

## End(Not run)
```

`r_copycat` *Run an R process that mimics the current R process*

Description

Differences to `r()`:

- No extra repoditories are set up.
- The `--no-site-file`, `--no-environ`, `--no-save`, `--no-restore` command line arguments are not used. (But `--slave` still is.)
- The system profile and the user profile are loaded.
- No extra environment variables are set up.

Usage

```
r_copycat(func, args = list(), libpath = .libPaths(),
  repos = getOption("repos"), cmdargs = "--slave", system_profile = TRUE,
  user_profile = TRUE, env = character(), ...)
```

Arguments

`func`	Function object to call in the new R process. The function should be self-contained and only refer to other functions and use variables explicitly from other packages using the `::` notation. The environment of the function is set to `.GlobalEnv` before passing it to the child process. Because of this, it is good practice to create an anonymous function and pass that to `callr`, instead of passing a function object from a (base or other) package. In particular
	`r(.libPaths)`
	does not work, because `.libPaths` is defined in a special environment, but
	`r(function() .libPaths())`
	works just fine.

`args`	Arguments to pass to the function. Must be a list.
`libpath`	The library path.

repos	The *repos* option. If NULL, then no *repos* option is set. This options is only used if user_profile or system_profile is set FALSE, as it is set using the system or the user profile.
cmdargs	Command line arguments to pass to the R process. Note that c("-f", rscript) is appended to this, rscript is the name of the script file to run. This contains a call to the supplied function and some error handling code.
system_profile	Whether to use the system profile file.
user_profile	Whether to use the user's profile file.
env	Environment variables to set for the child process.
...	Additional arguments are passed to r().

See Also

Other callr functions: r_vanilla, r

---

r_process *External R Process*

Description

An R process that runs in the background. This is an R6 class that extends the process class.

Usage

```
rp <- r_process$new(options)
rp$get_result()
```

See process for the inherited methods.

Arguments

- options A list of options created via r_process_options().

Details

r_process$new creates a new instance. Its options argument is best created by the r_process_options() function.

rp$get_result() returns the result, an R object, from a finished background R process. If the process has not finished yet, it throws an error. (You can use rp$wait() to wait for the process to finish, optionally with a timeout.)

Examples

```
## Not run:
## List all options and their default values:
r_process_options()

## Start an R process in the background, wait for it, get result
opts <- r_process_options(func = function() 1 + 1)
rp <- r_process$new(opts)
rp$wait()
rp$get_result()

## End(Not run)
```

`r_process_options` *Create options for an r_process object*

Description

Create options for an r_process object

Usage

```
r_process_options(...)
```

Arguments

`...`	Options to override, named arguments.

Value

A list of options.

`r_process_options()` creates a set of options to initialize a new object from the `r_process` class. Its arguments must be named, the names are used as option names. The options correspond to (some of) the arguments of the `r()` function. At least the `func` option must be specified, this is the R function to run in the background.

Examples

```
## List all options and their default values:
r_process_options()
```

r_vanilla *Run an R child process, with no configuration*

Description

It tries to mimic a fresh R installation. In particular:

- No library path setting.
- No CRAN(-like) repository is set.
- The system and user profiles are not run.

Usage

```
r_vanilla(func, args = list(), libpath = character(), repos = c(CRAN =
  "@CRAN@"), cmdargs = "--slave", system_profile = FALSE,
  user_profile = FALSE, env = character(), ...)
```

Arguments

func	Function object to call in the new R process. The function should be self-contained and only refer to other functions and use variables explicitly from other packages using the :: notation. The environment of the function is set to .GlobalEnv before passing it to the child process. Because of this, it is good practice to create an anonymous function and pass that to callr, instead of passing a function object from a (base or other) package. In particular r(.libPaths) does not work, because .libPaths is defined in a special environment, but r(function() .libPaths()) works just fine.
args	Arguments to pass to the function. Must be a list.
libpath	The library path.
repos	The *repos* option. If NULL, then no *repos* option is set. This options is only used if user_profile or system_profile is set FALSE, as it is set using the system or the user profile.
cmdargs	Command line arguments to pass to the R process. Note that c("-f", rscript) is appended to this, rscript is the name of the script file to run. This contains a call to the supplied function and some error handling code.
system_profile	Whether to use the system profile file.
user_profile	Whether to use the user's profile file.
env	Environment variables to set for the child process.
...	Additional arguments are passed to r().

See Also

Other callr functions: r_copycat, r

Examples

```
## Not run:
# Compare to r()
r(function() .libPaths())
r_vanilla(function() .libPaths())

r(function() getOption("repos"))
r_vanilla(function() getOption("repos"))

## End(Not run)
```

Index