Package 'memoise'

July 16, 2025

Title 'Memoisation' of Functions

Version 2.0.1

Description Cache the results of a function so that when you call it again with the same arguments it returns the previously computed value.

License MIT + file LICENSE

URL https://memoise.r-lib.org, https://github.com/r-lib/memoise

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

Imports rlang (>= 0.4.10), cachem

Suggests digest, aws.s3, covr, googleAuthR, googleCloudStorageR, httr, testthat

Encoding UTF-8

RoxygenNote 7.1.2

NeedsCompilation no

Author Hadley Wickham [aut],
Jim Hester [aut],
Winston Chang [aut, cre],
Kirill Müller [aut],
Daniel Cook [aut],
Mark Edmondson [ctb]

Maintainer Winston Chang <winston@rstudio.com>

Repository CRAN

Date/Publication 2021-11-26 16:11:10 UTC

Contents

cache_filesystem 2
cache_gcs 3
cache_memory 3
cache_s3 4

drop_cache 5
forget 5
has_cache 6
is.memoised 7
memoise 7
timeout 10

Index **11**

`cache_filesystem` *Filesystem Cache*

Description

Use a cache on the local filesystem that will persist between R sessions.

Usage

```
cache_filesystem(path, algo = "xxhash64", compress = FALSE)
```

Arguments

`path`	Directory in which to store cached items.
`algo`	The hashing algorithm used for the cache, see `digest` for available algorithms.
`compress`	Argument passed to `saveRDS`. One of FALSE, "gzip", "bzip2" or "xz". Default: FALSE.

Examples

```
## Not run:
# Use with Dropbox

db <- cache_filesystem("~/Dropbox/.rcache")

mem_runif <- memoise(runif, cache = db)

# Use with Google Drive

gd <- cache_filesystem("~/Google Drive/.rcache")

mem_runif <- memoise(runif, cache = gd)


## End(Not run)
```

cache_gcs *Google Cloud Storage Cache Google Cloud Storage backed cache, for remote caching.*

Description

Google Cloud Storage Cache Google Cloud Storage backed cache, for remote caching.

Usage

```
cache_gcs(
  cache_name = googleCloudStorageR::gcs_get_global_bucket(),
  algo = "sha512",
  compress = FALSE
)
```

Arguments

cache_name	Bucket name for storing cache files.
algo	The hashing algorithm used for the cache, see digest for available algorithms.
compress	Argument passed to saveRDS. One of FALSE, "gzip", "bzip2" or "xz". Default: FALSE.

Examples

```
## Not run:
library(googleCloudStorageR)
# Set GCS credentials.
Sys.setenv("GCS_AUTH_FILE"="<google-service-json>",
           "GCS_DEFAULT_BUCKET"="unique-bucket-name")

gcs <- cache_gcs("unique-bucket-name")
mem_runif <- memoise(runif, cache = gcs)

## End(Not run)
```

cache_memory *In Memory Cache*

Description

A cache in memory, that lasts only in the current R session.

Usage

```
cache_memory(algo = "sha512")
```

Arguments

algo	The hashing algorithm used for the cache, see digest for available algorithms.

---

cache_s3 *Amazon Web Services S3 Cache Amazon Web Services S3 backed cache, for remote caching.*

Description

Amazon Web Services S3 Cache Amazon Web Services S3 backed cache, for remote caching.

Usage

```
cache_s3(cache_name, algo = "sha512", compress = FALSE)
```

Arguments

cache_name	Bucket name for storing cache files.
algo	The hashing algorithm used for the cache, see digest for available algorithms.
compress	Argument passed to saveRDS. One of FALSE, "gzip", "bzip2" or "xz". Default: FALSE.

Examples

```
## Not run:
# Set AWS credentials.
Sys.setenv("AWS_ACCESS_KEY_ID" = "<access key>",
           "AWS_SECRET_ACCESS_KEY" = "<access secret>")

# Set up a unique bucket name.
s3 <- cache_s3("unique-bucket-name")
mem_runif <- memoise(runif, cache = s3)

## End(Not run)
```

drop_cache *Drops the cache of a memoised function for particular arguments.*

Description

Drops the cache of a memoised function for particular arguments.

Usage

```
drop_cache(f)
```

Arguments

f	Memoised function.

Value

A function, with the same arguments as `f`, that can be called to drop the cached results of `f`.

See Also

`has_cache`, `memoise`

Examples

```
mem_sum <- memoise(sum)
mem_sum(1, 2, 3)
mem_sum(2, 3, 4)
has_cache(mem_sum)(1, 2, 3) # TRUE
has_cache(mem_sum)(2, 3, 4) # TRUE
drop_cache(mem_sum)(1, 2, 3) # TRUE
has_cache(mem_sum)(1, 2, 3) # FALSE
has_cache(mem_sum)(2, 3, 4) # TRUE
```

forget *Forget past results. Resets the cache of a memoised function. Use* `drop_cache` *to reset the cache only for particular arguments.*

Description

Forget past results. Resets the cache of a memoised function. Use `drop_cache` to reset the cache only for particular arguments.

Usage

```
forget(f)
```

Arguments

f	memoised function

See Also

memoise, is.memoised, drop_cache

Examples

```
memX <- memoise(function() { Sys.sleep(1); runif(1) })
# The forget() function
system.time(print(memX()))
system.time(print(memX()))
forget(memX)
system.time(print(memX()))
```

---

has_cache *Test whether a memoised function has been cached for particular arguments.*

Description

Test whether a memoised function has been cached for particular arguments.

Usage

```
has_cache(f)
```

Arguments

f	Function to test.

Value

A function, with the same arguments as f, that can be called to test if f has cached results.

See Also

is.memoised, memoise, drop_cache

Examples

```
mem_sum <- memoise(sum)
has_cache(mem_sum)(1, 2, 3) # FALSE
mem_sum(1, 2, 3)
has_cache(mem_sum)(1, 2, 3) # TRUE
```

is.memoised — *Test whether a function is a memoised copy. Memoised copies of functions carry an attribute* `memoised = TRUE`*, which is what* `is.memoised()` *tests for.*

Description

Test whether a function is a memoised copy. Memoised copies of functions carry an attribute `memoised = TRUE`, which is what `is.memoised()` tests for.

Usage

```
is.memoised(f)
```

Arguments

`f`	Function to test.

See Also

`memoise`, `forget`

Examples

```
mem_lm <- memoise(lm)
is.memoised(lm) # FALSE
is.memoised(mem_lm) # TRUE
```

memoise — *Memoise a function*

Description

`mf <- memoise(f)` creates `mf`, a memoised copy of `f`. A memoised copy is basically a lazier version of the same function: it saves the answers of new invocations, and re-uses the answers of old ones. Under the right circumstances, this can provide a very nice speedup indeed.

Usage

```
memoise(
  f,
  ...,
  envir = environment(f),
  cache = cachem::cache_mem(max_size = 1024 * 1024^2),
  omit_args = c(),
  hash = function(x) rlang::hash(x)
)
```

Arguments

f	Function of which to create a memoised copy.
...	optional variables to use as additional restrictions on caching, specified as one-sided formulas (no LHS). See Examples for usage.
envir	Environment of the returned function.
cache	Cache object. The default is a [cachem::cache_mem()] with a max size of 1024 MB.
omit_args	Names of arguments to ignore when calculating hash.
hash	A function which takes an R object as input and returns a string which is used as a cache key.

Details

There are two main ways to use the `memoise` function. Say that you wish to memoise `glm`, which is in the `stats` package; then you could use
`mem_glm <- memoise(glm)`, or you could use
`glm <- memoise(stats::glm)`.
The first form has the advantage that you still have easy access to both the memoised and the original function. The latter is especially useful to bring the benefits of memoisation to an existing block of R code.

Two example situations where `memoise` could be of use:

- You're evaluating a function repeatedly over the rows (or larger chunks) of a dataset, and expect to regularly get the same input.
- You're debugging or developing something, which involves a lot of re-running the code. If there are a few expensive calls in there, memoising them can make life a lot more pleasant. If the code is in a script file that you're `source()`ing, take care that you don't just put
  `glm <- memoise(stats::glm)`
  at the top of your file: that would reinitialise the memoised function every time the file was sourced. Wrap it in
  `if (!is.memoised(glm))` , or do the memoisation call once at the R prompt, or put it somewhere else where it won't get repeated.

It is recommended that functions in a package are not memoised at build-time, but when the package is loaded. The simplest way to do this is within `.onLoad()` with, for example

```
# file.R
fun <- function() {
 some_expensive_process()
}

# zzz.R
.onLoad <- function(libname, pkgname) {
 fun <<- memoise::memoise(fun)
}
```

See Also

forget, is.memoised, timeout, https://en.wikipedia.org/wiki/Memoization, drop_cache

Examples

```
# a() is evaluated anew each time. memA() is only re-evaluated
# when you call it with a new set of parameters.
a <- function(n) { runif(n) }
memA <- memoise(a)
replicate(5, a(2))
replicate(5, memA(2))

# Caching is done based on parameters' value, so same-name-but-
# changed-value correctly produces two different outcomes...
N <- 4; memA(N)
N <- 5; memA(N)
# ... and same-value-but-different-name correctly produces
#     the same cached outcome.
N <- 4; memA(N)
N2 <- 4; memA(N2)

# memoise() knows about default parameters.
b <- function(n, dummy="a") { runif(n) }
memB <- memoise(b)
memB(2)
memB(2, dummy="a")
# This works, because the interface of the memoised function is the same as
# that of the original function.
formals(b)
formals(memB)
# However, it doesn't know about parameter relevance.
# Different call means different caching, no matter
# that the outcome is the same.
memB(2, dummy="b")

# You can create multiple memoisations of the same function,
# and they'll be independent.
memA(2)
memA2 <- memoise(a)
memA(2)  # Still the same outcome
memA2(2) # Different cache, different outcome

# Multiple memoized functions can share a cache.
cm <- cachem::cache_mem(max_size = 50 * 1024^2)
memA <- memoise(a, cache = cm)
memB <- memoise(b, cache = cm)

# Don't do the same memoisation assignment twice: a brand-new
# memoised function also means a brand-new cache, and *that*
# you could as easily and more legibly achieve using forget().
# (If you're not sure whether you already memoised something,
#  use is.memoised() to check.)
```

```
memA(2)
memA <- memoise(a)
memA(2)

# Make a memoized result automatically time out after 10 seconds.
memA3 <- memoise(a, cache = cachem::cache_mem(max_age = 10))
memA3(2)
```

`timeout` *Return a new number after a given number of seconds*

Description

This function will return a number corresponding to the system time and remain stable until a given number of seconds have elapsed, after which it will update to the current time. This makes it useful as a way to timeout and invalidate a memoised cache after a certain period of time.

Usage

```
timeout(seconds, current = as.numeric(Sys.time()))
```

Arguments

`seconds`	Number of seconds after which to timeout.
`current`	The current time as a numeric.

Value

A numeric that will remain constant until the seconds have elapsed.

See Also

`memoise`

Examples

```
a <- function(n) { runif(n) }
memA <- memoise(a, ~timeout(10))
memA(2)
```

Index

cache_filesystem, 2
cache_gcs, 3
cache_memory, 3
cache_s3, 4

digest, *2–4*
drop_cache, *5*, 5, *6*, *9*

forget, 5, *7*, *9*

has_cache, *5*, 6

is.memoised, *6*, 7, *9*
is.memoized *(*is.memoised*)*, 7

memoise, *5–7*, 7, *10*
memoize *(*memoise*)*, 7

timeout, *9*, 10