Package 'rotor'

December 13, 2020

Type Package

Title Log Rotation and Conditional Backups

Version 0.3.5

Maintainer Stefan Fleck <stefan.b.fleck@gmail.com>

Description Conditionally rotate or back-up files based on their size or the date of the last backup; inspired by the 'Linux' utility 'logrotate'.

License MIT + file LICENSE

URL https://s-fleck.github.io/rotor/

BugReports https://github.com/s-fleck/rotor/issues

Imports dint, R6, tools

Suggests covr, crayon, data.table, digest, rmarkdown, testthat, uuid, ulid, zip

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1.9000

NeedsCompilation no

Author Stefan Fleck [aut, cre] (<https://orcid.org/0000-0003-3344-9851>)

Repository CRAN

Date/Publication 2020-12-13 15:20:02 UTC

R topics documented:

BackupQueue 2
BackupQueueDate 4
BackupQueueDateTime 5
BackupQueueIndex 7
backup_info 8
Cache 10
DirectoryQueue 14
rotate 15
rotate_rds 21

Index **25**

`BackupQueue` *An R6 Class for managing backups (abstract base class)*

Description

`BackupQueue`is an abstract class not intended for direct usage, please refer to BackupQueueIndex, BackupQueueDateTime, BackupQueueDate instead.

Details

This class is part of the R6 API of **rotor** which is intended for developers that want to extend this package. For normal usage, the simpler functional API is recommended (see `rotate()`).

Super class

`rotor::DirectoryQueue` -> `BackupQueue`

Public fields

`dir` `character` scalar. Directory in which to place the backups.

`n` `integer` scalar. The number of backups that exist for `BackupQueue$origin`

Active bindings

`dir` `character` scalar. Directory in which to place the backups.

`n` `integer` scalar. The number of backups that exist for `BackupQueue$origin`

`file` `character` scalar. The file to backup/rotate.

`compression` (Optional) compression to use `compression` argument of `rotate()`.

`max_backups` Maximum number/size/age of backups. See `max_backups` argument of `rotate()`

`has_backups` Returns `TRUE` if at least one backup of `BackupQueue$origin` exists All backups of self$origin

Methods

Public methods:

- `BackupQueue$new()`
- `BackupQueue$prune()`
- `BackupQueue$prune_identical()`
- `BackupQueue$print()`
- `BackupQueue$push_backup()`
- `BackupQueue$set_origin()`
- `BackupQueue$set_compression()`
- `BackupQueue$set_max_backups()`

- BackupQueue$set_file()
- BackupQueue$set_backup_dir()

Method `new()`**:**

Usage:

```
BackupQueue$new(
  origin,
  dir = dirname(origin),
  max_backups = Inf,
  compression = FALSE,
  backup_dir = NULL
)
```

Method `prune()`**:** Delete all backups except `max_backups`. See `prune_backups()`.

Usage:

```
BackupQueue$prune(max_backups = self$max_backups)
```

Method `prune_identical()`**:** Delete all identical backups. Uses `tools::md5sum()` to compare the files. Set the file to be backed up

Usage:

```
BackupQueue$prune_identical()
```

Method `print()`**:**

Usage:

```
BackupQueue$print()
```

Method `push_backup()`**:**

Usage:

```
BackupQueue$push_backup(...)
```

Method `set_origin()`**:**

Usage:

```
BackupQueue$set_origin(x)
```

Arguments:

`x` a `character` scalar. Path to a file Set the file to be backed up

Method `set_compression()`**:**

Usage:

```
BackupQueue$set_compression(x)
```

Arguments:

`x` a `character` scalar. Path to a file

Method `set_max_backups()`**:**

Usage:

```
BackupQueue$set_max_backups(x)
```

Method `set_file()`**:**

Usage:

```
BackupQueue$set_file(x)
```

Method `set_backup_dir()`**:**

Usage:

```
BackupQueue$set_backup_dir(x)
```

See Also

Other R6 Classes: `BackupQueueDateTime`, `BackupQueueDate`, `BackupQueueIndex`, `Cache`, `DirectoryQueue`

`BackupQueueDate` *An R6 class for managing datestamped backups*

Description

A BackupQueue for date-stamped backups, e.g. `foo.log`, foo.2020-07-24.log

Details

This class is part of the R6 API of **rotor** which is intended for developers that want to extend this package. For normal usage, the simpler functional API is recommended (see `rotate()`).

Super classes

`rotor::DirectoryQueue` -> `rotor::BackupQueue` -> `rotor::BackupQueueDateTime` -> `BackupQueueDate`

Methods

Public methods:

- `BackupQueueDate$new()`
- `BackupQueueDate$set_fmt()`

Method `new()`**:**

Usage:

```
BackupQueueDate$new(
  origin,
  dir = dirname(origin),
  max_backups = Inf,
  compression = FALSE,
  fmt = "%Y-%m-%d",
  cache_backups = FALSE,
  backup_dir = NULL
)
```

Method `set_fmt()`**:**

Usage:

```
BackupQueueDate$set_fmt(x)
```

See Also

Other R6 Classes: BackupQueueDateTime, BackupQueueIndex, BackupQueue, Cache, DirectoryQueue

---

BackupQueueDateTime *An R6 class for managing timestamped backups*

Description

A BackupQueue for timestamped backups, e.g. `foo.log`, foo.2020-07-24_10-54-30.log

Details

This class is part of the R6 API of **rotor** which is intended for developers that want to extend this package. For normal usage, the simpler functional API is recommended (see `rotate()`).

Super classes

`rotor::DirectoryQueue` -> `rotor::BackupQueue` -> BackupQueueDateTime

Active bindings

`fmt` See `format` argument of `rotate_date()` `logical` scalar. If `TRUE` (the default) the list of backups is cached, if `FALSE` it is read from disk every time this appender triggers. Caching brings a significant speedup for checking whether to rotate or not based on the age of the last backup, but is only safe if there are no other programs/functions interacting with the backups. This is only advantageous for high frequency file rotation (i.e. several times per second) `POSIXct` scalar. Timestamp of the last rotation (the last backup)

Methods

Public methods:

- `BackupQueueDateTime$new()`
- `BackupQueueDateTime$push()`
- `BackupQueueDateTime$prune()`
- `BackupQueueDateTime$should_rotate()`
- `BackupQueueDateTime$update_backups_cache()`
- `BackupQueueDateTime$set_max_backups()`
- `BackupQueueDateTime$set_fmt()`
- `BackupQueueDateTime$set_cache_backups()`

Method `new()`:

Usage:

```
BackupQueueDateTime$new(
  origin,
  dir = dirname(origin),
  max_backups = Inf,
  compression = FALSE,
  fmt = "%Y-%m-%d--%H-%M-%S",
  cache_backups = FALSE,
  backup_dir = NULL
)
```

Method `push()`**:** Create a new time-stamped backup (e.g. 'logfile.2020-07-22_12-26-29.log')

Usage:

```
BackupQueueDateTime$push(overwrite = FALSE, now = Sys.time())
```

Arguments:

`overwrite` `logical` scalar. Overwrite backups with the same filename (i.e timestamp)?

`now` `POSIXct` scalar. Can be used as an override mechanism for the current system time if necessary.

Method `prune()`**:**

Usage:

```
BackupQueueDateTime$prune(max_backups = self$max_backups)
```

Method `should_rotate()`**:** Should a file of `size` and `age` be rotated? See `size` and `age` arguments of `rotate_date()`. `now` overrides the current system time, 'last_rotation" overrides the date of the last rotation.

Usage:

```
BackupQueueDateTime$should_rotate(
  size,
  age,
  now = Sys.time(),
  last_rotation = self$last_rotation %||% file.info(self$origin)$ctime,
  verbose = FALSE
)
```

Returns: `TRUE` or `FALSE`

Method `update_backups_cache()`**:** Force update of the backups cache (only if $cache_backups == TRUE).

Usage:

```
BackupQueueDateTime$update_backups_cache()
```

Method `set_max_backups()`**:**

Usage:

```
BackupQueueDateTime$set_max_backups(x)
```

Method `set_fmt()`**:**

Usage:

```
BackupQueueDateTime$set_fmt(x)
```

Method `set_cache_backups()`**:**

Usage:

```
BackupQueueDateTime$set_cache_backups(x)
```

See Also

Other R6 Classes: `BackupQueueDate`, `BackupQueueIndex`, `BackupQueue`, `Cache`, `DirectoryQueue`

`BackupQueueIndex` *An R6 class for managing indexed backups*

Description

A BackupQueue for indexed backups, e.g. `foo.log`, `foo.1.log`, `foo.2.log`, ...

Details

This class is part of the R6 API of **rotor** which is intended for developers that want to extend this package. For normal usage, the simpler functional API is recommended (see `rotate()`).

Super classes

`rotor::DirectoryQueue` -> `rotor::BackupQueue` -> `BackupQueueIndex`

Methods

Public methods:

- `BackupQueueIndex$push()`
- `BackupQueueIndex$prune()`
- `BackupQueueIndex$prune_identical()`
- `BackupQueueIndex$should_rotate()`
- `BackupQueueIndex$pad_index()`
- `BackupQueueIndex$increment_index()`

Method `push()`**:** Create a new index-stamped backup (e.g. '`logfile.1.log`')

Usage:

```
BackupQueueIndex$push()
```

Method `prune()`**:**

Usage:

```
BackupQueueIndex$prune(max_backups = self$max_backups)
```

Method `prune_identical()`**:**

Usage:

```
BackupQueueIndex$prune_identical()
```

Method `should_rotate()`**:** Should a file of `size` be rotated? See `size` argument of `rotate()`

Usage:

```
BackupQueueIndex$should_rotate(size, verbose = FALSE)
```

Returns: `TRUE` or `FALSE`

Method `pad_index()`**:** Pad the indices in the filenames of indexed backups to the number of digits of the largest index. Usually does not have to be called manually.

Usage:

```
BackupQueueIndex$pad_index()
```

Method `increment_index()`**:** Increment die Indices of all backups by `n` Usually does not have to be called manually.

Usage:

```
BackupQueueIndex$increment_index(n = 1)
```

Arguments:

`n` `integer` > 0

See Also

Other R6 Classes: `BackupQueueDateTime`, `BackupQueueDate`, `BackupQueue`, `Cache`, `DirectoryQueue`

`backup_info` *Discover existing backups*

Description

These function return information on the backups of a file (if any exist)

Usage

```
backup_info(file, dir = dirname(file))

list_backups(file, dir = dirname(file))

n_backups(file, dir = dirname(file))

newest_backup(file, dir = dirname(file))

oldest_backup(file, dir = dirname(file))
```

Arguments

`file`	`character` scalar: Path to a file.
`dir`	`character` scalar. The directory in which the backups of `file` are stored (defaults to `dirname(file)`)

Value

`backup_info()` returns a `data.frame` similar to the output of `file.info()`

`list_backups()` returns the paths to all backups of `file`

`n_backups()` returns the number of backups of `file` as an `integer` scalar

`newest_backup()` and `oldest_backup()` return the paths to the newest or oldest backup of `file` (or an empty `character` vector if none exist)

Intervals

In **rotor**, an interval is a character string in the form `"<number> <interval>"`. The following intervals are possible: `"day(s)"`, `"week(s)"`, `"month(s)"`, `"quarter(s)"`, `"year(s)"`. The plural `"s"` is optional (so `"2 weeks"` and `"2 week"` are equivalent). Please be aware that weeks are ISOweeks and start on Monday (not Sunday as in some countries).

Interval strings can be used as arguments when backing up or rotating files, or for pruning backup queues (i.e. limiting the number of backups of a single) file.

When rotating/backing up `"1 months"` means "make a new backup if the last backup is from the preceding month". E.g if the last backup of `myfile` is from `2019-02-01` then `backup_time(myfile, age = "1 month")` will only create a backup if the current date is at least `2019-03-01`.

When pruning/limiting backup queues, `"1 year"` means "keep at least most one year worth of backups". So if you call `backup_time(myfile, max_backups = "1 year")` on `2019-03-01`, it will create a backup and then remove all backups of `myfile` before `2019-01-01`.

See Also

`rotate()`

Examples

```
# setup example files
tf <- tempfile("test", fileext = ".rds")
saveRDS(cars, tf)
backup(tf)
backup(tf)

backup_info(tf)
list_backups(tf)
n_backups(tf)
newest_backup(tf)
oldest_backup(tf)

# cleanup
prune_backups(tf, 0)
n_backups(tf)
file.remove(tf)
```

Cache *An R6 class for managing a persistent file-based cache*

Description

Cache provides an R6 API for managing an on-disk key-value store for R objects. The objects are serialized to a single folder as .rds files and the key of the object equals the name of the file. Cache supports automatic removal of old files if the cache folder exceeds a predetermined number of files, total size, or if the individual files exceed a certain age.

Details

This class is part of the R6 API of **rotor** which is intended for developers that want to extend this package. For normal usage, the simpler functional API is recommended (see `rotate()`).

Super class

`rotor::DirectoryQueue` -> Cache

Public fields

`dir` a `character` scalar. path of the directory in which to store the cache files

`n` `integer` scalar: number of files in the cache

`max_files` see the `compress` argument of `base::saveRDS()`. **Note**: this differs from the $compress argument of `rotate()`.

`max_files` `integer` scalar: maximum number of files to keep in the cache

Active bindings

`dir` a `character` scalar. path of the directory in which to store the cache files

`n` `integer` scalar: number of files in the cache

`max_files` see the `compress` argument of `base::saveRDS()`. **Note**: this differs from the $compress argument of `rotate()`.

`max_files` `integer` scalar: maximum number of files to keep in the cache

`max_size` scalar `integer`, `character` or `Inf`. Delete cached files (starting with the oldest) until the total size of the cache is below `max_size`. `Integers` are interpreted as bytes. You can pass `character` vectors that contain a file size suffix like 1k (kilobytes), 3M (megabytes), 4G (gigabytes), 5T (terabytes). Instead of these short forms you can also be explicit and use the IEC suffixes `KiB`, `MiB`, `GiB`, `TiB`. In Both cases `1` kilobyte is `1024` bytes, `1` `megabyte` is `1024` kilobytes, etc... .

`max_age`
- a `Date` scalar: Remove all backups before this date
- a `character` scalar representing a Date in ISO format (e.g. `"2019-12-31"`)
- a `character` scalar representing an Interval in the form `"<number> <interval>"` (see `rotate()`)

`hashfun` `NULL` or a `function` to generate a unique hash from the object to be cached (see example). The hash *must* be a text string that is a valid filename on the target system. If $hashfun is `NULL`, a storage key must be supplied manually in `cache$push()`. If a new object is added with the same key as an existing object, the existing object will be overwritten without warning. All cached files

Methods

Public methods:

- `Cache$new()`
- `Cache$push()`
- `Cache$read()`
- `Cache$remove()`
- `Cache$pop()`
- `Cache$prune()`
- `Cache$purge()`
- `Cache$destroy()`
- `Cache$print()`
- `Cache$set_max_files()`
- `Cache$set_max_age()`
- `Cache$set_max_size()`
- `Cache$set_compression()`
- `Cache$set_hashfun()`

Method `new()`:

Usage:

```
Cache$new(
  dir = dirname(file),
  max_files = Inf,
  max_size = Inf,
  max_age = Inf,
  compression = TRUE,
  hashfun = digest::digest,
  create_dir = TRUE
)
```

Arguments:

`create_dir` `logical` scalar. If `TRUE` `dir` is created if it does not exist.

Examples:

```
td <- file.path(tempdir(), "cache-test")

# When using a real hash function as hashfun, identical objects will only
# be added to the cache once
cache_hash <- Cache$new(td, hashfun = digest::digest)
cache_hash$push(iris)
cache_hash$push(iris)
```

```
cache_hash$files
cache_hash$purge()

# To override this behaviour use a generator for unique ids, such as uuid
if (requireNamespace("uuid")){
  cache_uid <- Cache$new(td, hashfun = function(x) uuid::UUIDgenerate())
  cache_uid$push(iris)
  cache_uid$push(iris)
  cache_uid$files
  cache_uid$purge()
}

unlink(td, recursive = TRUE)
```

Method `push()`**:** push a new object to the cache

Usage:

```
Cache$push(x, key = self$hashfun(x))
```

Arguments:

`x` any R object

`key` a `character` scalar. Key under which to store the cached object. Must be a valid filename. Defaults to being generated by $hashfun() but may also be supplied manually.

Returns: a `character` scalar: the key of the newly added object

Method `read()`**:** read a cached file

Usage:

```
Cache$read(key)
```

Arguments:

`key` `character` scalar. key of the cached file to read.

Method `remove()`**:** remove a single file from the cache

Usage:

```
Cache$remove(key)
```

Arguments:

`key` `character` scalar. key of the cached file to remove

Method `pop()`**:** Read and remove a single file from the cache

Usage:

```
Cache$pop(key)
```

Arguments:

`key` `character` scalar. key of the cached file to read/remove

Method `prune()`**:** Prune the cache

Delete cached objects that match certain criteria. `max_files` and `max_size` deletes the oldest cached objects first; however, this is dependent on accuracy of the file modification timestamps

on your system. For example, ext3 only supports second-accuracy, and some windows version only support timestamps at a resolution of two seconds.
If two files have the same timestamp, they are deleted in the lexical sort order of their key. This means that by using a function that generates lexically sortable keys as `hashfun` (such as `ulid::generate()`) you can enforce the correct deletion order. There is no such workaround if you use a real hash function.

Usage:

```
Cache$prune(
  max_files = self$max_files,
  max_size = self$max_size,
  max_age = self$max_age,
  now = Sys.time()
)
```

Arguments:

`max_files`, `max_size`, `max_age` see section Active Bindings.

`now` a `POSIXct` datetime scalar. The current time (for max_age)

Method `purge()`**:** purge the cache (remove all cached files)

Usage:

```
Cache$purge()
```

Method `destroy()`**:** purge the cache (remove all cached files)

Usage:

```
Cache$destroy()
```

Method `print()`**:**

Usage:

```
Cache$print()
```

Method `set_max_files()`**:**

Usage:

```
Cache$set_max_files(x)
```

Method `set_max_age()`**:**

Usage:

```
Cache$set_max_age(x)
```

Method `set_max_size()`**:**

Usage:

```
Cache$set_max_size(x)
```

Method `set_compression()`**:**

Usage:

```
Cache$set_compression(x)
```

Method `set_hashfun()`**:**

Usage:

```
Cache$set_hashfun(x)
```

See Also

Other R6 Classes: BackupQueueDateTime, BackupQueueDate, BackupQueueIndex, BackupQueue, DirectoryQueue

Examples

```
## ------------------------------------------------
## Method `Cache$new`
## ------------------------------------------------

td <- file.path(tempdir(), "cache-test")

# When using a real hash function as hashfun, identical objects will only
# be added to the cache once
cache_hash <- Cache$new(td, hashfun = digest::digest)
cache_hash$push(iris)
cache_hash$push(iris)
cache_hash$files
cache_hash$purge()

# To override this behaviour use a generator for unique ids, such as uuid
if (requireNamespace("uuid")){
  cache_uid <- Cache$new(td, hashfun = function(x) uuid::UUIDgenerate())
  cache_uid$push(iris)
  cache_uid$push(iris)
  cache_uid$files
  cache_uid$purge()
}

unlink(td, recursive = TRUE)
```

`DirectoryQueue` *An R6 class for managing persistent file-based queues (abstract base class)*

Description

Abstract class from which all other classes in **rotor** inherit their basic fields and methods.

Details

This class is part of the R6 API of **rotor** which is intended for developers that want to extend this package. For normal usage, the simpler functional API is recommended (see `rotate()`).

Active bindings

`dir` a `character` scalar. path of the directory in which to store the cache files

Methods

Public methods:

- DirectoryQueue$new()
- DirectoryQueue$push()
- DirectoryQueue$prune()
- DirectoryQueue$set_dir()

Method new():

Usage:

```
DirectoryQueue$new(...)
```

Method push():

Usage:

```
DirectoryQueue$push(x, ...)
```

Method prune():

Usage:

```
DirectoryQueue$prune(x, ...)
```

Method set_dir():

Usage:

```
DirectoryQueue$set_dir(x, create = TRUE)
```

See Also

Other R6 Classes: BackupQueueDateTime, BackupQueueDate, BackupQueueIndex, BackupQueue, Cache

---

rotate *Rotate or backup files*

Description

Functions starting with backup create backups of a file, while functions starting with rotate do the same but also replace the original file with an empty one (this is useful for log rotation)

prune_backups() physically deletes all backups of a file based on max_backups

prune_backups() physically deletes all backups of a file based on max_backups

Usage

```
rotate(
  file,
  size = 1,
  max_backups = Inf,
  compression = FALSE,
  dir = dirname(file),
  create_file = TRUE,
  dry_run = FALSE,
  verbose = dry_run
)

backup(
  file,
  size = 0,
  max_backups = Inf,
  compression = FALSE,
  dir = dirname(file),
  dry_run = FALSE,
  verbose = dry_run
)

prune_backups(
  file,
  max_backups,
  dir = dirname(file),
  dry_run = FALSE,
  verbose = dry_run
)

prune_identical_backups(
  file,
  dir = dirname(file),
  dry_run = FALSE,
  verbose = dry_run
)

rotate_date(
  file,
  age = 1,
  size = 1,
  max_backups = Inf,
  compression = FALSE,
  format = "%Y-%m-%d",
  dir = dirname(file),
  overwrite = FALSE,
  create_file = TRUE,
  now = Sys.Date(),
```

```
  dry_run = FALSE,
  verbose = dry_run
)

backup_date(
  file,
  age = 1,
  size = 1,
  max_backups = Inf,
  compression = FALSE,
  format = "%Y-%m-%d",
  dir = dirname(file),
  overwrite = FALSE,
  now = Sys.Date(),
  dry_run = FALSE,
  verbose = dry_run
)

rotate_time(
  file,
  age = -1,
  size = 1,
  max_backups = Inf,
  compression = FALSE,
  format = "%Y-%m-%d--%H-%M-%S",
  dir = dirname(file),
  overwrite = FALSE,
  create_file = TRUE,
  now = Sys.time(),
  dry_run = FALSE,
  verbose = dry_run
)

backup_time(
  file,
  age = -1,
  size = 1,
  max_backups = Inf,
  compression = FALSE,
  format = "%Y-%m-%d--%H-%M-%S",
  dir = dirname(file),
  overwrite = FALSE,
  now = Sys.time(),
  dry_run = FALSE,
  verbose = dry_run
)
```

Arguments

file	character scalar: file to backup/rotate
size	scalar integer, character or Inf. Backup/rotate only if file is larger than this size. Integers are interpreted as bytes. You can pass character vectors that contain a file size suffix like 1k (kilobytes), 3M (megabytes), 4G (gigabytes), 5T (terabytes). Instead of these short forms you can also be explicit and use the IEC suffixes KiB, MiB, GiB, TiB. In Both cases 1 kilobyte is 1024 bytes, 1 megabyte is 1024 kilobytes, etc... . (if age *and* size are provided, both criteria must be TRUE to trigger rotation)
max_backups	maximum number of backups to keep • an integer scalar: Maximum number of backups to keep In addition for timestamped backups the following value are supported: • a Date scalar: Remove all backups before this date • a character scalar representing a Date in ISO format (e.g. "2019-12-31") • a character scalar representing an Interval in the form "<number> <interval>" (see below for more info)
compression	Whether or not backups should be compressed • FALSE for uncompressed backups, • TRUE for zip compression; uses zip() • a scalar integer between 1 and 9 to specify a compression level (requires the zip package, see its documentation for details) • the character scalars "utils::zip()" or "zip::zipr" to force a specific zip command
dir	character scalar. The directory in which the backups of file are stored (defaults to dirname(file))
create_file	logical scalar. If TRUE create an empty file in place of file after rotating.
dry_run	logical scalar. If TRUE no changes are applied to the file system (no files are created or deleted)
verbose	logical scalar. If TRUE additional informative messages are printed
age	minimum age after which to backup/rotate a file; can be • a character scalar representing an Interval in the form "<number> <interval>" (e.g. "2 months", see *Intervals* section below). • a Date or a character scalar representing a Date for a fixed point in time after which to backup/rotate. See format for which Date/Datetime formats are supported by rotor. (if age *and* size are provided, both criteria must be TRUE to trigger rotation)
format	a scalar character that can be a subset of of valid strftime() formatting strings. The default setting is "%Y-%m-%d--%H-%M-%S". • You can use an arbitrary number of dashes anywhere in the format, so "%Y-%m-%d--%H-%M-%S" and "%Y%m%d%H%M%S" are both legal. • T and _ can also be used as separators. For example, the following datetime formats are also possible: %Y-%m-%d_%H-%M-%S (Python logging default), %Y%m%dT%H%M%S (ISO 8601)

- All datetime components except %Y are optional. If you leave out part of the timestamp, the first point in time in the period is assumed. For example (assuming the current year is 2019) %Y is identical to `2019-01-01--00-00-00`.
- The timestamps must be lexically sortable, so `"%Y-%m-%d"` is legal, `"%m-%d-%Y"` and %Y-%d are not.

`overwrite` `logical` scalar. If `TRUE` overwrite backups if a backup of the same name (usually due to timestamp collision) exists.

`now` The current `Date` or time (`POSIXct`) as a scalar. You can pass a custom value here to to override the real system time. As a convenience you can also pass in `character` strings that follow the guidelines outlined above for `format`, but please note that these differ from the formats understood by `as.POSIXct()` or `as.Date()`.

Value

`file` as a `character` scalar (invisibly)

Side Effects

`backup()`, `backup_date()`, and `backup_time()` may create files (if the specified conditions are met). They may also delete backups, based on `max_backup`.

`rotate()`, `rotate_date()` and `rotate_time()` do the same, but in addition delete the input `file`, or replace it with an empty file if `create_file == TRUE` (the default).

`prune_backups()` may delete files, depending on `max_backups`.

`prune_backups()` may delete files, depending on `max_backups`.

Intervals

In **rotor**, an interval is a character string in the form `"<number> <interval>"`. The following intervals are possible: `"day(s)"`, `"week(s)"`, `"month(s)"`, `"quarter(s)"`, `"year(s)"`. The plural `"s"` is optional (so `"2 weeks"` and `"2 week"` are equivalent). Please be aware that weeks are ISOweeks and start on Monday (not Sunday as in some countries).

Interval strings can be used as arguments when backing up or rotating files, or for pruning backup queues (i.e. limiting the number of backups of a single) file.

When rotating/backing up `"1 months"` means "make a new backup if the last backup is from the preceding month". E.g if the last backup of `myfile` is from `2019-02-01` then `backup_time(myfile,age = "1 month")` will only create a backup if the current date is at least `2019-03-01`.

When pruning/limiting backup queues, `"1 year"` means "keep at least most one year worth of backups". So if you call `backup_time(myfile,max_backups = "1 year")` on `2019-03-01`, it will create a backup and then remove all backups of `myfile` before `2019-01-01`.

See Also

`list_backups()`

Examples

```
# setup example file
tf <- tempfile("test", fileext = ".rds")
saveRDS(cars, tf)

# create two backups of `tf``
backup(tf)
backup(tf)
list_backups(tf)  # find all backups of a file

# If `size` is set, a backup is only created if the target file is at least
# that big. This is more useful for log rotation than for backups.
backup(tf, size = "100 mb")  # no backup becuase `tf` is to small
list_backups(tf)

# If `dry_run` is TRUE, backup() only shows what would happen without
# actually creating or deleting files
backup(tf, size = "0.1kb", dry_run = TRUE)

# rotate() is the same as backup(), but replaces `tf`` with an empty file
rotate(tf)
list_backups(tf)
file.size(tf)
file.size(list_backups(tf))

# prune_backups() can remove old backups
prune_backups(tf, 1)  # keep only one backup
list_backups(tf)

# rotate/backup_date() adds a date instead of an index
# you should not mix index backups and timestamp backups
# so we clean up first
prune_backups(tf, 0)
saveRDS(cars, tf)

# backup_date() adds the date instead of an index to the filename
backup_date(tf)

# `age` sets the minimum age of the last backup before creating a new one.
# the example below creates no new backup since it's less than a week
# since the last.
backup_date(tf, age = "1 week")

# `now` overrides the current date.
backup_date(tf, age = "1 year", now = "2999-12-31")
list_backups(tf)

# backup_time() creates backups with a full timestamp
backup_time(tf)

# It's okay to mix backup_date() and backup_time()
list_backups(tf)
```

```
# cleanup
prune_backups(tf, 0)
file.remove(tf)
```

`rotate_rds` *Serialize R objects to disk (with backup)*

Description

The rotate_rds*() functions are wrappers around base::saveRDS() that create a backup of the destination file (if it exists) instead of just overwriting it.

Usage

```
rotate_rds(
  object,
  file = "",
  ascii = FALSE,
  version = NULL,
  compress = TRUE,
  refhook = NULL,
  ...,
  on_change_only = FALSE
)

rotate_rds_date(
  object,
  file = "",
  ascii = FALSE,
  version = NULL,
  compress = TRUE,
  refhook = NULL,
  ...,
  age = -1L,
  on_change_only = FALSE
)

rotate_rds_time(
  object,
  file = "",
  ascii = FALSE,
  version = NULL,
  compress = TRUE,
  refhook = NULL,
  ...,
  age = -1L,
```

```
  on_change_only = FALSE
)
```

Arguments

object	R object to serialize.
file	a connection or the name of the file where the R object is saved to or read from.
ascii	a logical. If TRUE or NA, an ASCII representation is written; otherwise (default), a binary one is used. See the comments in the help for save.
version	the workspace format version to use. NULL specifies the current default version (3). The only other supported value is 2, the default from R 1.4.0 to R 3.5.0.
compress	a logical specifying whether saving to a named file is to use "gzip" compression, or one of "gzip", "bzip2" or "xz" to indicate the type of compression to be used. Ignored if file is a connection.
refhook	a hook function for handling reference objects.
...	Arguments passed on to rotate, rotate_date, rotate_time

max_backups maximum number of backups to keep

- an integer scalar: Maximum number of backups to keep

In addition for timestamped backups the following value are supported:

- a Date scalar: Remove all backups before this date
- a character scalar representing a Date in ISO format (e.g. "2019-12-31")
- a character scalar representing an Interval in the form "<number> <interval>" (see below for more info)

size scalar integer, character or Inf. Backup/rotate only if file is larger than this size. Integers are interpreted as bytes. You can pass character vectors that contain a file size suffix like 1k (kilobytes), 3M (megabytes), 4G (gigabytes), 5T (terabytes). Instead of these short forms you can also be explicit and use the IEC suffixes KiB, MiB, GiB, TiB. In Both cases 1 kilobyte is 1024 bytes, 1 megabyte is 1024 kilobytes, etc... .
(if age *and* size are provided, both criteria must be TRUE to trigger rotation)

dir character scalar. The directory in which the backups of file are stored (defaults to dirname(file))

compression Whether or not backups should be compressed

- FALSE for uncompressed backups,
- TRUE for zip compression; uses zip()
- a scalar integer between 1 and 9 to specify a compression level (requires the zip package, see its documentation for details)
- the character scalars "utils::zip()" or "zip::zipr" to force a specific zip command

dry_run logical scalar. If TRUE no changes are applied to the file system (no files are created or deleted)

verbose logical scalar. If TRUE additional informative messages are printed

create_file logical scalar. If TRUE create an empty file in place of file after rotating.

format a scalar character that can be a subset of of valid strftime() formatting strings. The default setting is "%Y-%m-%d--%H-%M-%S".

- You can use an arbitrary number of dashes anywhere in the format, so "%Y-%m-%d--%H-%M-%S" and "%Y%m%d%H%M%S" are both legal.
- T and _ can also be used as separators. For example, the following datetime formats are also possible: %Y-%m-%d_%H-%M-%S (Python logging default), %Y%m%dT%H%M%S (ISO 8601)
- All datetime components except %Y are optional. If you leave out part of the timestamp, the first point in time in the period is assumed. For example (assuming the current year is 2019) %Y is identical to 2019-01-01--00-00-00.
- The timestamps must be lexically sortable, so "%Y-%m-%d" is legal, "%m-%d-%Y" and %Y-%d are not.

now The current Date or time (POSIXct) as a scalar. You can pass a custom value here to to override the real system time. As a convenience you can also pass in character strings that follow the guidelines outlined above for format, but please note that these differ from the formats understood by as.POSIXct() or as.Date().

overwrite logical scalar. If TRUE overwrite backups if a backup of the same name (usually due to timestamp collision) exists.

on_change_only logical scalar. Rotate only if object is different from the object saved in file.

age minimum age after which to backup/rotate a file; can be

- a character scalar representing an Interval in the form "<number> <interval>" (e.g. "2 months", see *Intervals* section below).
- a Date or a character scalar representing a Date for a fixed point in time after which to backup/rotate. See format for which Date/Datetime formats are supported by rotor.

(if age *and* size are provided, both criteria must be TRUE to trigger rotation)

Value

the path to file (invisibly)

Note

The default value for age is different for rotate_rds_date() (-1) than for rotate_date() (1) to make it a bit safer. This means if you execute rotate_date() twice on the same file on a given day it will silently not rotate the file, while rotate_rds_date() will throw an error.

Examples

```
dest <- tempfile()
rotate_rds(iris, dest)
rotate_rds(iris, dest)
rotate_rds(iris, dest)

list_backups(dest)
```

```
# cleanup
unlink(list_backups(dest))
unlink(dest)
```

Index

∗ **R6 Classes**
  BackupQueue, 2
  BackupQueueDate, 4
  BackupQueueDateTime, 5
  BackupQueueIndex, 7
  Cache, 10
  DirectoryQueue, 14
.rds, *10*

as.Date(), *19*, *23*
as.POSIXct(), *19*, *23*

backup *(*rotate*)*, 15
backup_date *(*rotate*)*, 15
backup_info, 8
backup_time *(*rotate*)*, 15
BackupQueue, 2, *5*, *7, 8*, *14, 15*
BackupQueueDate, *2*, *4*, 4, *7, 8*, *14, 15*
BackupQueueDateTime, *2*, *4, 5*, 5, *8*, *14, 15*
BackupQueueIndex, *2*, *4, 5*, *7*, 7, *14, 15*
base::saveRDS(), *10*, *21*

Cache, *4, 5*, *7, 8*, 10, *15*
connection, *22*

DirectoryQueue, *4, 5*, *7, 8*, *14*, 14

file.info(), *9*

list_backups *(*backup_info*)*, 8
list_backups(), *19*

n_backups *(*backup_info*)*, 8
newest_backup *(*backup_info*)*, 8

oldest_backup *(*backup_info*)*, 8

prune_backups *(*rotate*)*, 15
prune_backups(), *3*
prune_identical_backups *(*rotate*)*, 15

R6, *2*, *4, 5*, *7*, *10*, *14*
rotate, 15, *22*
rotate(), *2*, *4, 5*, *7–10*, *14*
rotate_date, *22*
rotate_date *(*rotate*)*, 15
rotate_date(), *5, 6*, *23*
rotate_rds, 21
rotate_rds_date *(*rotate_rds*)*, 21
rotate_rds_time *(*rotate_rds*)*, 21
rotate_time, *22*
rotate_time *(*rotate*)*, 15
rotor::BackupQueue, *4, 5*, *7*
rotor::BackupQueueDateTime, *4*
rotor::DirectoryQueue, *2*, *4, 5*, *7*, *10*

save, *22*

tools::md5sum(), *3*

ulid::generate(), *13*

zip(), *18*, *22*