Package 'pkgcache'

May 31, 2019

Title Cache 'CRAN'-Like Metadata and R Packages

Version 1.0.5

Description Metadata and package cache for CRAN-like repositories. This is a utility package to be used by package management tools that want to take advantage of caching.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

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

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

Imports assertthat, cli, cliapp, curl (>= 3.2), crayon, digest, filelock, glue, prettyunits, R6, rappdirs, rematch2, rlang, tibble, tools, utils, uuid, withr

Suggests callr, covr, debugme, desc, fs, jsonlite, mockery, pingr, rprojroot, testthat,

Depends R (>= 3.1)

RoxygenNote 6.1.1

NeedsCompilation no

Author Gábor Csárdi [aut, cre]

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

Repository CRAN

Date/Publication 2019-05-31 11:20:07 UTC

R topics documented:

`cranlike_metadata_cache` *Metadata cache for a CRAN-like repository*

Description

This is an R6 class that implements the metadata cache of a CRAN-like repository. For a higher level interface, see the `meta_cache_list()`, `meta_cache_deps()`, `meta_cache_revdeps()` and `meta_cache_update()` functions.

Usage

```
cranlike_metadata_cache
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

The cache has several layers:

- The data is stored inside the `cranlike_metadata_cache` object.
- It is also stored as an RDS file, in the session temporary directory. This ensures that the same data is used for all queries of a `cranlike_metadata_cache` object.
- It is stored in an RDS file in the user's cache directory.
- The downloaded raw `PACKAGES*` files are cached, together with HTTP ETags, to minimize downloads.

It has a synchronous and an asynchronous API.

Usage

```
cmc <- cranlike_metadata_cache$new(
  primary_path = NULL, replica_path = tempfile(),
  platforms = default_platforms(), r_version = current_r_version(),
  bioc = TRUE, cran_mirror = default_cran_mirror(),
  repos = getOption("repos"),
  update_after = as.difftime(7, units = "days"))

cmc$list(packages = NULL)
cmc$async_list(packages = NULL)

cmc$deps(packages, dependencies = NA, recursive = TRUE)
cmc$async_deps(packages, dependencies = NA, recursive = TRUE)

cmc$revdeps(packages, dependencies = NA, recursive = TRUE)
```

```
cmc$async_revdeps(packages, dependencies = NA, recursive = TRUE)

cmc$update()
cmc$async_update()
cmc$check_update()
cmc$asnyc_check_update()

cmc$summary()

cmc$cleanup(force = FALSE)
```

Arguments

- `primary_path`: Path of the primary, user level cache. Defaults to the user level cache directory of the machine.
- `replica_path`: Path of the replica. Defaults to a temporary directory within the session temporary directory.
- `platforms`: Subset of `c("macos", "windows", "source")`, platforms to get data for.
- `r_version`: R version to create the cache for.
- `bioc`: Whether to include BioConductor packages.
- `cran_mirror`: CRAN mirror to use, this takes precedence over `repos`.
- `repos`: Repositories to use.
- `update_after`: `difftime` object. Automatically update the cache if it gets older than this. Set it to `Inf` to avoid updates. Defaults to seven days.
- `packages`: Packages to query, character vector.
- `dependencies`: Which kind of dependencies to include. Works the same way as the `dependencies` argument of `utils::install.packages()`.
- `recursive`: Whether to include recursive dependencies.
- `force`: Whether to force cleanup without asking the user.

Details

`cranlike_metadata_cache$new()` creates a new cache object. Creation does not trigger the population of the cache. It is only populated on demand, when queries are executed against it. In your package, you may want to create a cache instance in the `.onLoad()` function of the package, and store it in the package namespace. As this is a cheap operation, the package will still load fast, and then the package code can refer to the common cache object.

`cmc$list()` lists all (or the specified) packages in the cache. It returns a tibble, see the list of columns below.

`cmc$async_list()` is similar, but it is asynchronous, it returns a `deferred` object.

`cmc$deps()` returns a tibble, with the (potentially recursive) dependencies of `packages`.

`cmc$async_deps()` is the same, but it is asynchronous, it returns a `deferred` object.

`cmc$revdeps()` returns a tibble, with the (potentially recursive) reverse dependencies of `packages`.

`cmc$async_revdeps()` does the same, asynchronously, it returns an `deferred` object.

`cmc$update()` updates the the metadata (as needed) in the cache, and then returns a tibble with all packages, invisibly.

`cmc$async_update()` is similar, but it is asynchronous.

`cmc$check_update()` checks if the metadata is current, and if it is not, it updates it.

`cmc$async_check_update()` is similar, but it is asynchronous.

`cmc$summary()` lists metadata about the cache, including its location and size.

`cmc$cleanup()` deletes the cache files from the disk, and also from memory.

Columns

The metadata tibble contains all available versions (i.e. sources and binaries) for all packages. It usually has the following columns, some might be missing on some platforms.

- `package`: Package name.
- `title`: Package title.
- `version`: Package version.
- `depends`: `Depends` field from `DESCRIPTION`, or `NA_character_`.
- `suggests`: `Suggests` field from `DESCRIPTION`, or `NA_character_`.
- `built`: `Built` field from `DESCIPTION`, if a binary package, or `NA_character_`.
- `imports`: `Imports` field from `DESCRIPTION`, or `NA_character_`.
- `archs`: `Archs` entries from `PACKAGES` files. Might be missing.
- `repodir`: The directory of the file, inside the repository.
- `platform`: Possible values: `macos`, `windows`, `source`.
- `needscompilation`: Whether the package needs compilation.
- `type`: `bioc` or `cran` currently.
- `target`: The path of the package file inside the repository.
- `mirror`: URL of the CRAN/BioC mirror.
- `sources`: List column with URLs to one or more possible locations of the package file. For source CRAN packages, it contains URLs to the `Archive` directory as well, in case the package has been archived since the metadata was cached.
- `filesize`: Size of the file, if known, in bytes, or `NA_integer_`.
- `sha256`: The SHA256 hash of the file, if known, or `NA_character_`.
- `deps`: All package dependencies, in a tibble.
- `license`: Package license, might be `NA` for binary packages.
- `linkingto`: `LinkingTo` field from `DESCRIPTION`, or `NA_character_`.
- `enhances`: `Enhances` field from `DESCRIPTION`, or `NA_character_`.
- `os_type`: `unix` or `windows` for OS specific packages. Usually `NA`.
- `priority`: "optional", "recommended" or `NA`. (Base packages are normalliy not included in the list, so "base" should not appear here.)
- `md5sum`: MD5 sum, if available, may be `NA`.

- sysreqs: For CRAN packages, the SystemRequirements field, the required system libraries or other software for the package. For non-CRAN packages it is NA.
- published: The time the package was published at, in GMT, POSIXct class.

The tibble contains some extra columns as well, these are for internal use only.

Examples

```
dir.create(cache_path <- tempfile())
cmc <- cranlike_metadata_cache$new(cache_path, bioc = FALSE)
cmc$list()
cmc$list("pkgconfig")
cmc$deps("pkgconfig")
cmc$revdeps("pkgconfig", recursive = FALSE)
```

---

get_cranlike_metadata_cache *The R6 object that implements the global metadata cache*

Description

This is used by the meta_cache_deps(), meta_cache_list(), etc. functions.

Usage

```
get_cranlike_metadata_cache()
```

---

meta_cache_deps *Query CRAN(like) package data*

Description

It uses CRAN and BioConductor packages, for the current platform and R version, from the default repositories.

Usage

```
meta_cache_deps(packages, dependencies = NA, recursive = TRUE)

meta_cache_revdeps(packages, dependencies = NA, recursive = TRUE)

meta_cache_update()

meta_cache_list(packages = NULL)
```

```
meta_cache_cleanup(force = FALSE)

meta_cache_summary()
```

Arguments

packages	Packages to query.
dependencies	Dependency types to query. See the dependencies parameter of utils::install.packages().
recursive	Whether to query recursive dependencies.
force	Whether to force cleanup without asking the user.

Details

meta_cache_list() lists all packages.

meta_cache_update() updates all metadata. Note that metadata is automatically updated if it is older than seven days.

meta_cache_deps() queries packages dependencies.

meta_cache_revdeps() queries reverse package dependencies.

meta_cache_summary() lists data about the cache, including its location and size.

meta_cache_cleanup() deletes the cache files from the disk.

Value

A data frame (tibble) of the dependencies. For meta_cache_deps() and meta_cache_revdeps() it includes the queried packages as well.

Examples

```
meta_cache_deps("dplyr")
meta_cache_list(c("MASS", dplyr"))
meta_cache_update()
```

Examples

```
meta_cache_list("pkgdown")
meta_cache_deps("pkgdown", recursive = FALSE)
meta_cache_revdeps("pkgdown", recursive = FALSE)
```

package_cache *A simple package cache*

Description

This is an R6 class that implements a concurrency safe package cache.

Usage

```
package_cache
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

By default these fields are included for every package:

- `fullpath` Full package path.
- `path` Package path, within the repository.
- `package` Package name.
- `url` URL it was downloaded from.
- `etag` ETag for the last download, from the given URL.
- `sha256` SHA256 hash of the file.

Additional fields can be added as needed.

For a simple API to a session-wide instance of this class, see `pkg_cache_summary()` and the other functions listed there.

Usage

```
pc <- package_cache$new(path = NULL)

pc$list()
pc$find(..., .list = NULL)
pc$copy_to(..., .list = NULL)
pc$add(file, path, sha256 = shasum256(file), ..., .list = NULL)
pc$add_url(url, path, ..., .list = NULL, on_progress = NULL,
  http_headers = NULL)
pc$async_add_url(url, path, ..., .list = NULL, on_progress = NULL,
  http_headers = NULL)
pc$copy_or_add(target, urls, path, sha256 = NULL, ..., .list = NULL,
               on_progress = NULL, http_headers = NULL)
pc$async_copy_or_add(target, urls, path, ..., sha256 = NULL, ...,
               .list = NULL, on_progress = NULL, http_headers = NULL)
```

```
pc$update_or_add(target, urls, path, ..., .list = NULL,
                on_progress = NULL, http_headers = NULL)
pc$async_update_or_add(target, urls, path, ..., .list = NULL,
                on_progress = NULL, http_headers = NULL)
pc$delete(..., .list = NULL)
```

Arguments

- `path`: For `package_cache$new()` the location of the cache. For other functions the location of the file inside the cache.
- `...`: Extra attributes to search for. They have to be named.
- `.list`: Extra attributes to search for, they have to in a named list.
- `file`: Path to the file to add.
- `url`: URL attribute. This is used to update the file, if requested.
- `sha256`: SHA256 hash of the file.
- `on_progress`: Callback to create progress bar. Passed to internal function `http_get()`.
- `target`: Path to copy the (first) to hit to.
- `urls`: Character vector or URLs to try to download the file from.
- `http_headers`: HTTP headers to add to all HTTP queries.

Details

`package_cache$new()` attaches to the cache at `path`. (By default a platform dependent user level cache directory.) If the cache does not exists, it creates it.

`pc$list()` lists all files in the cache, returns a tibble with all the default columns, and potentially extra columns as well.

`pc$find()` list all files that match the specified criteria (`fullpath`, `path`, `package`, etc.). Custom columns can be searched for as well.

`pc$copy_to()` will copy the first matching file from the cache to `target`. It returns the tibble of *all* matching records, invisibly. If no file matches, it returns an empty (zero-row) tibble.

`pc$add()` adds a file to the cache.

`pc$add_url()` downloads a file and adds it to the cache.

`pc$async_add_url()` is the same, but it is asynchronous.

`pc$copy_or_add()` works like `pc$copy_to()`, but if the file is not in the cache, it tries to download it from one of the specified URLs first.

`pc$async_copy_or_add()` is the same, but asynchronous.

`pc$update_or_add()` is like `pc$copy_to_add()`, but if the file is in the cache it tries to update it from the urls, using the stored ETag to avoid unnecessary downloads.

`pc$async_update_or_add()` is the same, but it is asynchronous.

`pc$delete()` deletes the file(s) from the cache.

Examples

```
## Although package_cache usually stores packages, it may store
## arbitrary files, that can be search by metadata
pc <- package_cache$new(path = tempfile())
pc$list()

cat("foo\n", file = f1 <- tempfile())
cat("bar\n", file = f2 <- tempfile())
pc$add(f1, "/f1")
pc$add(f2, "/f2")
pc$list()
pc$find(path = "/f1")
pc$copy_to(target = f3 <- tempfile(), path = "/f1")
readLines(f3)
```

---

pkg_cache_summary *Functions to query and manipulate the package cache*

Description

pkg_cache_summary() returns a short summary of the state of the cache, e.g. the number of files and their total size. It returns a named list.

Usage

```
pkg_cache_summary(cachepath = NULL)

pkg_cache_list(cachepath = NULL)

pkg_cache_find(cachepath = NULL, ...)

pkg_cache_get_file(cachepath = NULL, target, ...)

pkg_cache_delete_files(cachepath = NULL, ...)

pkg_cache_add_file(cachepath = NULL, file, relpath = dirname(file),
  ...)
```

Arguments

cachepath	Path of the cache. By default the cache directory is in R-pkg, within the user's cache directory. See rappdirs::user_cache_dir().
...	Extra named arguments to select the package file.
target	Path where the selected file is copied.
file	File to add.
relpath	The relative path of the file within the cache.

See Also

The package_cache R6 class for a more flexible API.

Index