Package 'alabaster.base'

May 8, 2026

Title Save Bioconductor Objects to File

Version 1.13.0

Date 2026-01-04

License MIT + file LICENSE

Description Save Bioconductor data structures into file artifacts, and load them back into memory. This is a more robust and portable alternative to serialization of such objects into RDS files. Each artifact is associated with metadata for further interpretation; downstream applications can enrich this metadata with context-specific properties.

Imports alabaster.schemas, methods, utils, S4Vectors, rhdf5 (>= 2.47.6), jsonlite, jsonvalidate, Rcpp

Suggests BiocStyle, rmarkdown, knitr, testthat, digest, Matrix, alabaster.matrix

LinkingTo Rcpp, assorthead (>= 1.1.2), Rhdf5lib

VignetteBuilder knitr

SystemRequirements C++17, GNU make

RoxygenNote 7.3.3

Encoding UTF-8

biocViews DataRepresentation, DataImport

URL https://github.com/ArtifactDB/alabaster.base

BugReports https://github.com/ArtifactDB/alabaster.base/issues

git_url https://git.bioconductor.org/packages/alabaster.base

git_branch devel

git_last_commit 1d9992b

git_last_commit_date 2026-04-28

Repository Bioconductor 3.24

Date/Publication 2026-05-08

Author Aaron Lun [aut, cre]

Maintainer Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>

Contents

absolutizePath *Make an absolute file path*

Description

Create an absolute file path from a relative file path. All processing is purely lexical; the path itself does not have to exist on the filesystem.

Usage

```
absolutizePath(path)
```

Arguments

path	String containing an absolute or relative file path.

Value

An absolute file path corresponding to path. This is cleaned to remove .., . and ~ components.

Author(s)

Aaron Lun

Examples

```
absolutizePath("alpha")
absolutizePath("../alpha")
absolutizePath("../../alpha/./bravo")
absolutizePath("/alpha/bravo")
```

acquireFile *Acquire file or metadata*

Description

WARNING: these functions are deprecated. Applications are expected to handle acquisition of files before loaders are called. Acquire a file or metadata for loading. As one might expect, these are typically used inside a load* function.

Usage

```
acquireFile(project, path)

acquireMetadata(project, path)

## S4 method for signature 'character'
acquireFile(project, path)

## S4 method for signature 'character'
acquireMetadata(project, path)
```

Arguments

project	Any value specifying the project of interest. The default methods expect a string containing a path to a staging directory, but other objects can be used to control dispatch.
path	String containing a relative path to a resource inside the staging directory.

Details

By default, files and metadata are loaded from the same staging directory that is written to by stageObject. alabaster applications can define custom methods to obtain the files and metadata from a different location, e.g., remote databases. This is achieved by dispatching on a different class of project.

Each custom acquisition method should take two arguments. The first argument is an R object representing some concept of a "project". In the default case, this is a string containing a path to the staging directory representing the project. However, it can be anything, e.g., a number containing a database identifier, a list of identifiers and versions, and so on - as long as the custom acquisition method is capable of understanding it, the load* functions don't care.

The second argument is a string containing the relative path to the resource inside that project. This should be the path to a specific file inside the project, not the subdirectory containing the file. More concretely, it should be equivalent to the path in the *output* of stageObject, not the path to the subdirectory used as the input to the same function.

The return value for each custom acquisition function should be the same as their local counterparts. That is, any custom file acquisition function should return a file path, and any custom metadata acquisition function should return a naamed list of metadata.

Value

acquireFile methods return a local path to the file corresponding to the requested resource.

acquireMetadata methods return a named list of metadata for the requested resource.

Author(s)

Aaron Lun

Examples

```
# Staging an example DataFrame:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
dir.create(tmp)
info <- stageObject(df, tmp, path="coldata")
writeMetadata(info, tmp)

# Retrieving the metadata:
meta <- acquireMetadata(tmp, "coldata/simple.csv.gz")
str(meta)

# Retrieving the file:
acquireFile(tmp, "coldata/simple.csv.gz")
```

`altReadObject` *Alter the reading function*

Description

Allow alabaster applications to specify an alternative reading function in `altReadObject`.

Usage

```
altReadObject(path, ...)

altReadObjectFunction(fun)
```

Arguments

`path, ...`	Further arguments to pass to `readObject` or its equivalent.
`fun`	Function that can serve as a drop-in replacement for `readObject`.

Details

By default, `altReadObject` is just a wrapper around `readObject`. However, if `altReadObjectFunction` is called, `altReadObject` calls the replacement `fun` instead. This allows alabaster applications to inject wholesale or class-specific customizations into the reading process, e.g., to add more metadata whenever an instance of a particular class is encountered. Developers of alabaster extensions should use `altReadObject` (instead of `readObject`) to read child objects when writing their own reading functions, to ensure that application-specific customizations are respected for the children.

To motivate the use of `altReadObject`, consider the following scenario.

1. We have created a reading function `readX` function to read an instance of class X in an alabaster extension. This function may be called by `readObject` if instances of X are children of other objects.
2. An alabaster application Y requires the addition of some custom metadata during the reading process for X. It defines an alternative reading function `readObject2` that, upon encountering a schema for X, redirects to a application-specific reader `readX2`. An example implementation for `readX2` would involve calling `readX` and decorating the result with the extra metadata.
3. When operating in the context of application Y, the `readObject2` generic is used to set `altReadObjectFunction`. Any calls to `altReadObject` in Y's context will subsequently call `readObject2`.
4. So, when writing a reading function in an alabaster extension for a class that might contain instances of X as children, we use `altReadObject` instead of directly using `readObject`. This ensures that, if a child instance of X is encountered *and* we are operating in the context of application Y, we correctly call `readObject2` and then ultimately `readX2`.

The application-specific `fun` is free to do anything it wants as long as it understands the representation. It is usually most convenient to leverage the existing functionality in `readObject`, but if the application-specific saver in `altSaveObject` does something unusual, then `fun` is responsible for the correct interpretation of any custom representation.

Value

For `altReadObject`, any R object similar to those returned by `readObject`.

For `altReadObjectFunction`, the alternative function (if any) is returned if `fun` is missing. If `fun` is provided, it is used to define the alternative, and the previous alternative is returned.

Author(s)

Aaron Lun

Examples

```
old <- altReadObjectFunction()

# Setting it to something.
altReadObjectFunction(function(...) {
    print("YAY")
    readObject(...)
})

# Staging an example DataFrame:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
saveObject(df, tmp)

# And now reading it - this should print our message.
altReadObject(tmp)

# Restoring the old reader:
altReadObjectFunction(old)
```

---

altSaveObject *Alter the saving generic*

Description

Allow alabaster applications to divert to a different saving generic instead of `saveObject`.

Usage

```
altSaveObject(x, path, ...)

altSaveObjectFunction(generic)
```

Arguments

`x, path, ...`	Further arguments to pass to `saveObject` or an equivalent generic.
`generic`	Generic function that can serve as a drop-in replacement for `saveObject`.

Details

By default, `altSaveObject` is just a wrapper around `saveObject`. However, if `altSaveObjectFunction` is called, `altSaveObject` calls the replacement `generic` instead. This allows alabaster applications to inject wholesale or class-specific customizations into the saving process, e.g., to save more metadata whenever an instance of a particular class is encountered. Developers of alabaster extensions should use `altSaveObject` to save child objects when implementing `saveObject` methods, to ensure that application-specific customizations are respected for the children.

To motivate the use of `altSaveObject`, consider the following scenario.

1. We have created a staging method for class X, defined for the `saveObject` generic.
2. An alabaster application Y requires the addition of some custom metadata during the staging process for X. It defines an alternative staging generic `saveObject2` that, upon encountering an instance of X, redirects to an application-specific method (i.e., `saveObject2,X-method`). For example, the `saveObject2` method for X could call X's `saveObject` method and add the necessary metadata to the result.
3. When operating in the context of application Y, the `saveObject2` generic is used to set `altSaveObjectFunction`. Any calls to `altSaveObject` in Y's context will subsequently call `saveObject2`.
4. So, when writing a `saveObject` method for any objects that might contain an instance of X as a child, we call `altSaveObject` on that X object instead of directly using `saveObject`. This ensures that, if a child instance of X is encountered *and* we are operating in the context of application Y, we correctly call `saveObject2` and then ultimately the application-specific method.

The application-specific `generic` is free to do anything it wants as long as the custom representation is understood by the application-specific reader in `altReadObject`. However, it is usually most convenient to re-use the existing representations created by `saveObject`. This means that any customizations should not interfere with the validity of those representations, as defined by the **takane** specifications and enforced by `validateObject`. We recommend that any customizations should manifest as new files starting with an underscore, as this will not interfere by any **takane** file specification.

Value

For `altSaveObject`, files are created at the specified location, see `saveObject` for details.

For `altSaveObjectFunction`, the alternative generic (if any) is returned if `generic` is missing. If `generic` is provided, it is used to define the alternative, and the previous alternative is returned.

Author(s)

Aaron Lun

Examples

```
old <- altSaveObjectFunction()

# Creating a new generic for demonstration purposes:
setGeneric("superSaveObject", function(x, path, ...)
    standardGeneric("superSaveObject"))

setMethod("superSaveObject", "ANY", function(x, path, ...) {
    print("Falling back to the base method!")
```

```
    saveObject(x, path, ...)
})

altSaveObjectFunction(superSaveObject)

# Staging an example DataFrame. This should print our message.
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
altSaveObject(df, tmp)

# Restoring the old loader:
altSaveObjectFunction(old)
```

---

anyMissing *Find missing values*

Description

Find missing (NA) values. This is smart enough to distinguish them from NaN values in numeric x. For all other types, it just calls is.na or anyNA.

Usage

```
anyMissing(x)

is.missing(x)
```

Arguments

x	Vector or array of atomic values.

Value

For anyMissing, a logical scalar indicating whether any NA values were present in x.

For is.missing, a logical vector or array of shape equal to x, indicating whether each value is NA.

Author(s)

Aaron Lun

Examples

```
anyNA(c(NaN))
anyNA(c(NA))
anyMissing(c(NaN))
anyMissing(c(NA))

is.na(c(NA, NaN))
is.missing(c(NA, NaN))
```

`chooseMissingPlaceholderForHdf5` *Choose a missing value placeholder*

Description

In the **alabaster.*** framework, we mark missing entries inside HDF5 datasets with placeholder values. This function chooses a value for the placeholder that does not overlap with anything else in a vector.

Usage

```
chooseMissingPlaceholderForHdf5(x, .version = 3)
```

Arguments

`x`	An atomic vector to be saved to HDF5.
`.version`	Internal use only.

Details

For floating-point datasets, the placeholder will not be NA if there are mixtures of NAs and NaNs. We do not rely on the NaN payload to distinguish between these two values.

Placeholder values are typically saved as scalar attributes on the HDF5 dataset that they are used in. The usual name of this attribute is `"missing-value-placeholder"`, as encoding by `missingPlaceholderName`.

Value

A placeholder value for missing values in `x`, guaranteed to not be equal to any non-missing value in `x`.

Examples

```
chooseMissingPlaceholderForHdf5(c(TRUE, NA, FALSE))
chooseMissingPlaceholderForHdf5(c(1L, NA, 2L))
chooseMissingPlaceholderForHdf5(c("aaron", NA, "barry"))
chooseMissingPlaceholderForHdf5(c("aaron", NA, "barry", "NA"))
chooseMissingPlaceholderForHdf5(c(1.5, NA, 2.6))
chooseMissingPlaceholderForHdf5(c(1.5, NaN, NA, 2.6))
```

`cloneDirectory` *Clone an existing directory*

Description

Clone an existing directory to a new location. This is typically performed inside `saveObject` after detecting duplicated objects, see `?createDedupSession` for an example use case.

Usage

```
cloneDirectory(src, dest, action = c("link", "copy", "symlink", "relsymlink"))
```

Arguments

`src`	String containing the path to the source directory, typically generated by a prior `saveObject` call.
`dest`	String containing the path to the destination directory, typically the `path` in a subsequent `saveObject` call.
`action`	String specifying the action to use when cloning files from `src` to `dest`.

- `"copy"`: copy all files within `src` to their corresponding locations in `dest`.
- `"link"`: create a hard link from each file in `src` to its corresponding location in `dest`. If this fails, we silently fall back to a copy.
- `"symlink"`: create a symbolic link from each file in `src` to its corresponding location in `dest`. Each symbolic link contains the absolute path in the corresponding file in its original directory, which is useful when the contents of `dest` might be moved (but the original directory will not).
- `"relsymlink"`: create a symbolic link from each file in `src` to its corresponding location in `dest`. Each symbolic link contains the minimal relative path to the corresponding file in the original directory, which is useful when both `src` and `dest` are moved together, e.g., as they are part of the same parent object like a SummarizedExperiment.

Value

A new directory is created at `dest` with the contents of `src`, either copied or linked. `NULL` is invisibly returned.

Author(s)

Aaron Lun

See Also

`cloneFile`, to clone individual files.

Examples

```
tmp <- tempfile()
dir.create(tmp)

src <- file.path(tmp, "A")
dir.create(src)
write(file=file.path(src, "foobar"), LETTERS)

dest <- file.path(tmp, "B")
cloneDirectory(src, dest)
list.files(dest, recursive=TRUE)
```

---

cloneFile *Clone an existing file*

Description

Clone an existing file to a new location. This is typically performed inside `saveObject` methods for objects that contain a reference to a file.

Usage

```
cloneFile(src, dest, action = c("link", "copy", "symlink", "relsymlink"))
```

Arguments

`src`	String containing the path to the source file.
`dest`	String containing the destination file path, typically within the `path` supplied to `saveObject`.
`action`	String specifying the action to use when cloning `src` to `dest`.

- `"copy"`: copy `src` to `dest`.
- `"link"`: create a hard link from `src` to `dest`. If this fails, we silently fall back to a copy.
- `"symlink"`: create a symbolic link from `src` to `dest`. The symbolic link contains the absolute path to `src`, which is useful when `dest` might be moved but `src` will not.
- `"relsymlink"`: create a symbolic link from `src` to `dest`. Each symbolic link contains the minimal relative path to `src`, which is useful when both `src` and `dest` are moved together, e.g., as they are part of the same parent object like a SummarizedExperiment.

Value

A new file/link is created at `dest`. `NULL` is invisibly returned.

Author(s)

Aaron Lun

See Also

cloneDirectory, to clone entire directories.

Examples

```
tmp <- tempfile()
write(file=tmp, LETTERS)

dest <- tempfile()
cloneFile(tmp, dest)
readLines(dest)
```

---

createDedupSession *Deduplicate objects when saving*

Description

Utilities for deduplicating objects inside saveObject to save time and/or storage space.

Usage

```
createDedupSession()

checkObjectInDedupSession(x, session)

addObjectToDedupSession(x, session, path)
```

Arguments

x	Some object, typically S4.
session	Session object created by createDedupSession.
path	String containing the absolute path to the directory in which x is to be saved. This will be used for deduplication if another object is identified as a duplicate of x. If a relative path is provided, it will be converted to an absolute path.

Details

These utilities allow extension developers to support deduplication of objects in a top-level call to saveObject. For a given saveObject method, we can:

1. Accept a session object in an optional <PREFIX>.dedup.session= argument. We may also accept a <PREFIX>.dedup.action= argument to specify how any deduplication should be performed. Some <PREFIX> prefix should be chosen to avoid conflicts between multiple deduplication sessions.
2. If a session argument is provided, we call checkObjectInDedupSession(x, session) to see if the x is a duplicate of an existing object in the session. If a path is returned, we call cloneDirectory and return.
3. Otherwise, we save this object to disk, possibly passing the session argument as <PREFIX>.dedup.session= in further calls to saveObject for child objects. We call addObjectToDedupSession to add the current object to the session.

A user can enable deduplication by passing the output of `createDedupSession` to `<PREFIX>.dedup.session=` in the top-level call to `saveObject`. This is most typically performed when saving SummarizedExperiment objects with multiple assays, where one assay consists of delayed operations on another assay.

Value

`createDedupSession` will return a deduplication session that can be modified in-place.

If `x` is a duplicate of an object in `session`, `checkObjectInDedupSession` will return a string containing the absolute path to a directory representing that object. Otherwise, it will return `NULL`.

`addObjectToDedupSession` will add `x` to `session` with the supplied `path`. It returns `NULL` invisibly.

Author(s)

Aaron Lun

Examples

```
test <- function(x, path, test.dedup.session=NULL, test.dedup.action="link") {
   if (!is.null(test.dedup.session)) {
       original <- checkObjectInDedupSession(x, test.dedup.session)
       if (!is.null(original)) {
           cloneDirectory(original, path, test.dedup.action)
           return(invisible(NULL))
       }
   }
   dir.create(path)
   saveRDS(x, file.path(path, "whee.rds")) # replace this with actual saving code.
   if (!is.null(test.dedup.session)) {
       addObjectToDedupSession(x, test.dedup.session, path)
   }
}

library(S4Vectors)
y <- DataFrame(A=1:10, B=1:10)
tmp <- tempfile()
dir.create(tmp)

# Saving the first instance of the object, which is now stored in the session.
session <- createDedupSession()
checkObjectInDedupSession(y, session) # no duplicates yet.
test(y, file.path(tmp, "first"), test.dedup.session=session)

# Saving it again will trigger the deduplication.
checkObjectInDedupSession(y, session)
test(y, file.path(tmp, "duplicate"), test.dedup.session=session)

list.files(tmp, recursive=TRUE)
```

createRedirection *Create a redirection file*

Description

WARNING: this function is deprecated. Redirection is no longer supported in the latest **alabaster** *framework.* Create a redirection to another path in the same staging directory. This is useful for creating short-hand aliases for resources that have inconveniently long paths.

Usage

```
createRedirection(dir, src, dest)
```

Arguments

dir	String containing the path to the staging directory.
src	String containing the source path relative to dir.
dest	String containing the destination path relative to dir. This may be any path that can also be used in acquireMetadata.

Details

src should not correspond to an existing file inside dir. This avoids ambiguity when attempting to load src via acquireMetadata. Otherwise, it would be unclear as to whether the user wants the file at src or the redirection target dest.

src may correspond to existing directories. This is because directories cannot be used in acquireMetadata, so no such ambiguity exists.

Value

A list of metadata that can be processed by writeMetadata.

Author(s)

Aaron Lun

Examples

```
# Staging an example DataFrame:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
dir.create(tmp)
info <- stageObject(df, tmp, path="coldata")
writeMetadata(info, tmp)

# Creating a redirection:
redirect <- createRedirection(tmp, "foobar", "coldata/simple.csv.gz")
writeMetadata(redirect, tmp)

# We can then use this redirect to pull out metadata:
info2 <- acquireMetadata(tmp, "foobar")
```

```
str(info2)
```

`getSaveEnvironment` *Track the environment used for saving objects*

Description

Utilities to write, load and access the R environment used by `saveObject` for any given object.

Usage

```
getSaveEnvironment()

formatSaveEnvironment()

useSaveEnvironment(use)

registerSaveEnvironment(info = NULL)

loadSaveEnvironment(path)
```

Arguments

`use`	Logical scalar specifying whether to use a save environment during reading/saving of objects.
`info`	Named list containing information about the environment used to save each object. If `NULL`, this is created by calling `formatSaveEnvironment`.
`path`	String containing the path to a directory representing an object, same as that used by `saveObject` and `readObject`.

Details

When saving an object, **saveObject** will automatically record some details about the current R environment. This facilitates trouble-shooting and provides some opportunities for corrective measures if any bugs are found in older `saveObject` methods. Information about the save environment is stored in an `_environment.json` file inside the directory containing the object. Subdirectories for child objects may also have separate `_environment.json` files (e.g., if they were created in a different environment), otherwise it is assumed that they inherit the save environment from the parent object.

Application or extension developers are expected to call `getSaveEnvironment` from inside a loading function used by `readObject` or `altReadObject`. This wil return the save environment that was used for the "current" object, i.e., the object that was previously saved at `path`. By accessing the historical save environment, developers can check if buggy versions of the corresponding `saveObject` or `altSaveObject` methods were used. Appropriate corrective measures can then be applied to recover the correct object, warn users, etc. `getSaveEnvironment` can also be called inside `saveObject` or `altSaveObject` methods, in which case the current object is the one being saved.

In most cases, `registerSaveEnvironment` does not need to be explicitly called by end-users or developers. It is automatically executed by the top-level calls to the `saveObject` or `altSaveObject`

generics. Methods can simply call `getSaveEnvironment` to access the save environment information. Similarly, `loadSaveEnvironment` does not usually need to be explicitly called by end-users or developers, as it is automatically executed by each `readObject` or `altReadObject` call. Individual reader functions can simply call `getSaveEnvironment` to access the save environment information.

Tracking of the save environment can be disabled by setting `useSaveEnvironment(FALSE)`.

Value

`getSaveEnvironment` returns a named list describing the environment used to save the "current" object (see Details). The list should have a `type` field specifying the type of environment, e.g., `"R"`. For objects created by `saveObject`, this will typically have the same format as the list returned by `formatSaveEnvironment`. Alternatively, `NULL` is returned if `useSaveEnvironment` is set to `FALSE` or no environment information was recorded for the current object.

`formatSaveEnvironment` returns a named list containing the current R environment, derived from the `sessionInfo`. This records the R `version`, the `platform` in which R is running, and the versions of all `packages` as a named list.

If `use` is not supplied, `useSaveEnvironment` returns a logical scalar indicating whether to use the save environment information. If `use` is supplied, it is used to define the save environment usage policy, and the previous setting of this value is invisibly returned.

`registerSaveEnvironment` registers the current environment information in memory so that it can be returned by `getSaveEnvironment`. It returns a list containing a `restore` function that should be called `on.exit` to (i) restore the previous environment information; and a `write` function that accepts a `path` to a directory in which to create an `_environment.json` file with the environment information. Both functions are no-ops if `useSaveEnvironment` is set to `FALSE` or if a save environment has already been registered.

`loadSaveEnvironment` loads the environment information from a `_environment.json` file in `path`. It also registers the environment information in memory so that it is returned when `getSaveEnvironment` is called. It returns a function that should be called `on.exit` to restore the previous environment information. This function is a no-op if `useSaveEnvironment` is set to `FALSE`, or if the environment information is not parsable (in which case a warning will be emitted).

Author(s)

Aaron Lun

Examples

```
str(formatSaveEnvironment())

prev <- useSaveEnvironment(TRUE)
tmp <- tempfile()
dir.create(tmp)

wfun <- registerSaveEnvironment(tmp)
getSaveEnvironment()
wfun$restore()

useSaveEnvironment(prev)
```

hdf5 *HDF5 utilities*

Description

Basically just better versions of those in **rhdf5**, dedicated to **alabaster.base** and its dependents. Intended for **alabaster.*** developers only.

listObjects *List objects in a directory*

Description

List all objects in a directory, along with their types.

Usage

```
listObjects(dir, include.children = FALSE)
```

Arguments

dir	String containing a path to a directory containing objects saved by saveObject, possibly in separate subdirectories.
include.children	Logical scalar indicating whether to include child objects.

Value

A DFrame where each row corresponds to an object. It contains the following columns:

- path, the relative path to the object's subdirectory inside dir.
- type, the type of the object based on its OBJECT file (see ?readObjectFile).
- child, whether the object is a child of another object.

If include.children=FALSE, metadata is only returned for non-child objects.

Author(s)

Aaron Lun

Examples

```
tmp <- tempfile()
dir.create(tmp)

library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
saveObject(df, file.path(tmp, "whee"))

ll <- list(A=1, B=LETTERS, C=DataFrame(X=1:5))
saveObject(ll, file.path(tmp, "stuff"))
```

```
listObjects(tmp)
listObjects(tmp, include.children=TRUE)
```

`loadDirectory` *Load all non-child objects in a directory*

Description

WARNING: this function is deprecated, use `listObjects` *and loop over entries with* `readObject` *instead.* As the title suggests, this function loads all non-child objects in a staging directory. All loading is performed using `altLoadObject` to respect any application-specific overrides. Children are used to assemble their parent objects and are not reported here.

Usage

```
loadDirectory(dir, redirect.action = c("from", "to", "both"))
```

Arguments

`dir`	String containing a path to a staging directory.
`redirect.action`	String specifying how redirects should be handled: • `"to"` will report an object at the redirection destination, not the redirection source. • `"from"` will report an object at the redirection source(s), not the destination. • `"both"` will report an object at both the redirection source(s) and destination.

Value

A named list is returned containing all (non-child) R objects in `dir`.

Author(s)

Aaron Lun

Examples

```
tmp <- tempfile()
dir.create(tmp)

library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
meta <- stageObject(df, tmp, path="whee")
writeMetadata(meta, tmp)

ll <- list(A=1, B=LETTERS, C=DataFrame(X=1:5))
meta <- stageObject(ll, tmp, path="stuff")
writeMetadata(meta, tmp)
```

```
redirect <- createRedirection(tmp, "whoop", "whee/simple.csv.gz")
writeMetadata(redirect, tmp)

all.meta <- loadDirectory(tmp)
str(all.meta)
```

`moveObject` *Move a non-child object in the staging directory*

Description

WARNING: this function is deprecated, as directories of non-child objects can just be moved with regular methods (e.g., `file.rename`*) in the latest version of* **alabaster**. Pretty much as it says in the title. This only works with non-child objects as children are referenced by their parents and cannot be safely moved in this manner.

Usage

```
moveObject(dir, from, to, rename.redirections = TRUE)
```

Arguments

`dir`	String containing the path to the staging directory.
`from`	String containing the path to a non-child object inside `dir`, as used in `acquireMetadata`. This can also be a redirection to such an object.
`to`	String containing the new path inside `dir`.
`rename.redirections`	Logical scalar specifying whether redirections pointing to `from` should be renamed as `to`.

Details

This function will look around `path` for JSON files containing redirections to `from`, and update them to point to `to`. More specifically, if `path` is a subdirectory, it will search in the same directory containing `path`; otherwise, it will search in the directory containing `dirname(path)`. Redirections in other locations will not be removed automatically - these will be caught by `checkValidDirectory` and should be manually updated.

If `rename.redirections=TRUE`, this function will additionally move the redirection files so that they are named as `to`. In the unusual case where `from` is the target of multiple redirection files, the renaming process will clobber all of them such that only one of them will be present after the move.

Value

The object represented by `path` is moved, along with any redirections to it. A `NULL` is invisibly returned.

Safety of moving operations

In general, **alabaster.*** representations are safe to move as only the parent object's `resource.path` metadata properties will contain links to the children's paths. These links are updated with the new `to` path after running `moveObject` on the parent `from`.

However, alabaster applications may define custom data structures where the paths are present elsewhere, e.g., in the data file itself or in other metadata properties. If so, applications are reponsible for updating those paths to reflect the naming to `to`.

Author(s)

Aaron Lun

Examples

```
tmp <- tempfile()
dir.create(tmp)

library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
meta <- stageObject(df, tmp, path="whee")
writeMetadata(meta, tmp)

ll <- list(A=1, B=LETTERS, C=DataFrame(X=1:5))
meta <- stageObject(ll, tmp, path="stuff")
writeMetadata(meta, tmp)

redirect <- createRedirection(tmp, "whoop", "whee/simple.csv.gz")
writeMetadata(redirect, tmp)

list.files(tmp, recursive=TRUE)
moveObject(tmp, "whoop", "YAY")
list.files(tmp, recursive=TRUE)
```

---

quickLoadObject *Convenience helpers for handling local directories*

Description

WARNING: these functions are deprecated as the saving/reading functions are already simple enough in the newer versions of the **alabaster** *framework.* Read and write objects from a local staging directory. These are just convenience wrappers around functions like `loadObject`, `stageObject` and `writeMetadata`.

Usage

```
quickLoadObject(dir, path, ...)

quickStageObject(x, dir, path, ...)
```

Arguments

dir	String containing a path to the directory.
path	String containing a relative path to the object of interest inside dir.
...	Further arguments to pass to loadObject (for quickLoadObject) or stageObject (for quickStageObject).
x	Object to be saved.

Value

For quickLoadObject, the object at path.

For quickStageObject, the object is saved to path inside dir. All necessary directories are created if they are not already present. A NULL is returned invisibly.

Author(s)

Aaron Lun

Examples

```
local <- tempfile()

# Creating a slightly complicated object:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
df$C <- DataFrame(D=letters[1:10], E=runif(10))

# Saving it:
quickStageObject(df, local, "FOOBAR")

# Reading it back:
quickLoadObject(local, "FOOBAR")
```

---

quickReadCsv *Quickly read and write a CSV file*

Description

Quickly read and write a CSV file, usually as a part of staging or loading a larger object. This assumes that all files follow the comservatory specification.

Usage

```
quickReadCsv(
  path,
  expected.columns,
  expected.nrows,
  compression,
  row.names,
  parallel = TRUE
)
```

```
quickWriteCsv(
  df,
  path,
  ...,
  row.names = FALSE,
  compression = "gzip",
  validate = TRUE
)
```

Arguments

path	String containing a path to a CSV to read/write.
expected.columns	Named character vector specifying the type of each column in the CSV (excluding the first column containing row names, if row.names=TRUE).
expected.nrows	Integer scalar specifying the expected number of rows in the CSV.
compression	String specifying the compression that was/will be used. This should be either "none", "gzip".
row.names	For .quickReadCsv, a logical scalar indicating whether the CSV contains row names. For .quickWriteCsv, a logical scalar indicating whether to save the row names of df.
parallel	Whether reading and parsing should be performed concurrently.
df	A DFrame or data.frame object, containing only atomic columns.
...	Further arguments to pass to write.csv.
validate	Whether to double-check that the generated CSV complies with the comservatory specification.

Value

For .quickReadCsv, a DFrame containing the contents of path.

For .quickWriteCsv, df is written to path and a NULL is invisibly returned.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(A=1, B="Aaron")

temp <- tempfile()
.quickWriteCsv(df, path=temp, row.names=FALSE, compression="gzip")

.quickReadCsv(temp, c(A="numeric", B="character"), 1, "gzip", FALSE)
```

readAtomicVector *Read an atomic vector from disk*

Description

Read a vector consisting of atomic elements from its on-disk representation. This is usually not directly called by users, but is instead called by dispatch in readObject.

Usage

```
readAtomicVector(path, metadata, ...)
```

Arguments

path	Path to a directory created with any of the vector methods for saveObject.
metadata	Named list containing metadata for the object, see readObjectFile for details.
...	Further arguments, ignored.

Value

The vector described by info.

Author(s)

Aaron Lun

See Also

"saveObject,integer-method", for one of the staging methods.

Examples

```
tmp <- tempfile()
saveObject(setNames(runif(26), letters), tmp)
readObject(tmp)
```

readBaseFactor *Read a factor from disk*

Description

Read a base R factor from its on-disk representation. This is usually not directly called by users, but is instead called by dispatch in readObject.

Usage

```
readBaseFactor(path, metadata, ...)
```

Arguments

path	String containing a path to a directory, itself created with the saveObject method for factors.
metadata	Named list containing metadata for the object, see readObjectFile for details.
...	Further arguments, ignored.

Value

The vector described by info.

Author(s)

Aaron Lun

See Also

"saveObject,factor-method", for the staging method.

Examples

```
tmp <- tempfile()
saveObject(factor(letters[1:10], letters), tmp)
readObject(tmp)
```

---

readBaseList *Read a base list from disk*

Description

Read a list from its on-disk representation. This is usually not directly called by users, but is instead called by dispatch in readObject.

Usage

```
readBaseList(path, metadata, simple_list.parallel = TRUE, ...)
```

Arguments

path	String containing a path to a directory, itself created with the list method for stageObject.
metadata	Named list containing metadata for the object, see readObjectFile for details.
simple_list.parallel	Whether to perform reading and parsing in parallel for greater speed. Only relevant for lists stored in the JSON format.
...	Further arguments to be passed to altReadObject for complex child objects.

Details

The **uzuki2** specification (see https://github.com/ArtifactDB/uzuki2) allows length-1 vectors to be stored as-is or as a scalar. If the file stores a length-1 vector as-is, `readBaseList` will read the list element as a length-1 vector with the AsIs class. If the file stores a length-1 vector as a scalar, `readBaseList` will read the list element as a length-1 vector without this class. This allows downstream users to distinguish between the storage modes in the rare cases that it is necessary.

Value

The list represented by `path`.

Author(s)

Aaron Lun

See Also

"`stageObject,list-method`", for the staging method.

Examples

```
library(S4Vectors)
ll <- list(A=1, B=LETTERS, C=DataFrame(X=letters))

tmp <- tempfile()
saveObject(ll, tmp)
readObject(tmp)
```

`readDataFrame` *Read a DataFrame from disk*

Description

Read a DFrame from its on-disk representation. This is usually not directly called by users, but is instead called by dispatch in `readObject`.

Usage

```
readDataFrame(path, metadata, ...)
```

Arguments

`path`	String containing a path to the directory, itself created with `saveObject` method for DFrames.
`metadata`	Named list containing metadata for the object, see `readObjectFile` for details.
`...`	Further arguments, passed to `altLoadObject` for complex nested columns.

Value

The DFrame represented by `path`.

Author(s)

Aaron Lun

See Also

"saveObject,DataFrame-method", for the staging method.

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
saveObject(df, tmp)
readObject(tmp)
```

---

readDataFrameFactor *Read a DataFrame factor from disk*

Description

Read a DataFrameFactor from its on-disk representation. This is usually not directly called by users, but is instead called by dispatch in readObject.

Usage

```
readDataFrameFactor(path, metadata, ...)
```

Arguments

path	String containing a path to a directory, itself created with the saveObject method for DataFrameFactors.
metadata	Named list containing metadata for the object, see readObjectFile for details.
...	Further arguments to pass to internal altSaveObject calls.

Value

A DataFrameFactor represented by path.

Author(s)

Aaron Lun

See Also

"saveObject,DataFrameFactor-method", for the staging method.

Examples

```
library(S4Vectors)
df <- DataFrame(X=LETTERS[1:5], Y=1:5)
out <- DataFrameFactor(df[sample(5, 100, replace=TRUE),,drop=FALSE])

tmp <- tempfile()
saveObject(out, tmp)
readObject(tmp)
```

`readMetadata` *Read R-level metadata*

Description

Read `metadata` and `mcols` for a Annotated or Vector object, respectively. This is typically used inside loading functions for concrete subclasses.

Usage

```
readMetadata(x, metadata.path, mcols.path, ...)
```

Arguments

`x`	An Vector or Annotated object.
`metadata.path`	String containing a path to a directory, itself containing an on-disk representation of a base R list to be used as the `metadata`. Alternatively `NULL` to skip loading.
`mcols.path`	String containing a path to a directory, itself containing an on-disk representation of a DataFrame to be used as the `mcols`. Alternatively `NULL` to skip loading.
`...`	Further arguments to be passed to `altReadObject`.

Value

`x` is returned, possibly with `mcols` and `metadata` added to it.

Author(s)

Aaron Lun

See Also

`saveMetadata`, which does the staging.

`readObject` *Read an object from disk*

Description

Read an object from its on-disk representation. This is done by dispatching to an appropriate loading function based on the type in the `OBJECT` file.

Usage

```
readObject(path, metadata = NULL, ...)

readObjectFunctionRegistry()

registerReadObjectFunction(type, fun, existing = c("old", "new", "error"))
```

Arguments

`path`	String containing a path to a directory, itself created with a `saveObject` method.
`metadata`	Named list containing metadata for the object - most importantly, the `type` field that controls dispatch to the correct loading function. If `NULL`, this is automatically read by `readObjectFile(path)`.
`...`	Further arguments to pass to individual methods.
`type`	String specifying the name of type of the object.
`fun`	A loading function that accepts `path`, `metadata` and `...` (in that order), and returns the associated object. This may also be `NULL` to delete an existing entry in the registry.
`existing`	Logical scalar indicating the action to take if a function has already been registered for `type` - keep the old or new function, or throw an error.

Value

For `readObject`, an object created from the on-disk representation in `path`.

For `readObjectFunctionRegistry`, a named list of functions used to load each object type.

For `registerReadObjectFunction`, the function is added to the registry.

Comments for extension developers

`readObject` uses an internal registry of functions to decide how an object should be loaded into memory. Developers of alabaster extensions can add extra functions to this registry, usually in the `.onLoad` function of their packages. Alternatively, extension developers can request the addition of their packages to default registry.

If a loading function makes use of additional arguments in `...`, those arguments should be prefixed by the name of the object type for each method, e.g., `simple_list.parallel`. This avoids problems with conflicts in the interpretation of identically named arguments between different functions. Unlike the `...` arguments in `saveObject`, we prefix by the object type instead of the output class, as the former is used for dispatch here.

When writing loading functions for complex classes, extension developers may need to load child objects to compose the output object. In such cases, developers should use `altReadObject` on the

child subdirectories, rather than calling `readObject` directly. This ensures that any application-level overrides of the loading functions are respected. It is also expected that arguments in `...` are forwarded to internal `altReadObject` calls.

Developers can manually control `readObject` dispatch by suppling a `metadata` list where `metadata$type` is set to the desired object type. This pattern is commonly used inside the loading function for a subclass - an instance of the base class is first constructed by an internal `readObject` call with the modified `metadata$type`, after which the subclass-specific slots are added. (In practice, base construction should be done using `altReadObject` so as to respect application-specific overrides.)

Comments for application developers

Application developers can override `readObject` by specifying a custom function in `altReadObject`. This can be used to point to a different registry of reading functions, to perform pre- or post-reading actions, etc. If customization is type-specific, the custom `altReadObject` function can read the type from the `OBJECT` file to determine the most appropriate course of action; the `OBJECT` metadata can then be passed to the `metadata` argument of any internal `readObject` calls to avoid a redundant read from the same file.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
saveObject(df, tmp)
readObject(tmp)
```

`readObjectFile` *Utilities to read and save the object file*

Description

The OBJECT file inside each directory provides some high-level metadata of the object represented by that directory. It is guaranteed to have a `type` property that specifies the object type; individual objects may add their own information to this file. These methods are intended for developers to easily read and load information in the OBJECT file.

Usage

```
readObjectFile(path)

saveObjectFile(path, type, extra = list())
```

Arguments

path	Path to the directory representing an object.
type	String specifying the type of the object.
extra	Named list containing extra metadata to be written to the OBJECT file in path. Names should be unique, and any element named "type" will be overwritten by type.

Value

readObjectFile returns a named list of metadata for path.

saveObjectFile saves metadata to the OBJECT file inside path

Author(s)

Aaron Lun

Examples

```
tmp <- tempfile()
dir.create(tmp)
saveObjectFile(tmp, "foo", list(bar=list(version="1.0")))
readObjectFile(tmp)
```

---

removeObject *Remove a non-child object from the staging directory*

Description

WARNING: this function is deprecated, as directories of non-child objects can just be deleted with regular methods (e.g., file.rename*) in the latest version of* **alabaster**. Pretty much as it says in the title. This only works with non-child objects as children are referenced by their parents and cannot be safely removed in this manner.

Usage

```
removeObject(dir, path)
```

Arguments

dir	String containing the path to the staging directory.
path	String containing the path to a non-child object inside dir, as used in acquireMetadata. This can also be a redirection to such an object.

Details

This function will search around path for JSON files containing redirections to path, and remove them. More specifically, if path is a subdirectory, it will search in the same directory containing path; otherwise, it will search in the directory containing dirname(path). Redirections in other locations will not be removed automatically - these will be caught by checkValidDirectory and should be manually removed.

Value

The object represented by path is removed, along with any redirections to it. A NULL is invisibly returned.

Author(s)

Aaron Lun

Examples

```
tmp <- tempfile()
dir.create(tmp)

library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
meta <- stageObject(df, tmp, path="whee")
writeMetadata(meta, tmp)

ll <- list(A=1, B=LETTERS, C=DataFrame(X=1:5))
meta <- stageObject(ll, tmp, path="stuff")
writeMetadata(meta, tmp)

redirect <- createRedirection(tmp, "whoop", "whee/simple.csv.gz")
writeMetadata(redirect, tmp)

list.files(tmp, recursive=TRUE)
removeObject(tmp, "whoop")
list.files(tmp, recursive=TRUE)
```

Rfc3339 *Representing Internet date/times*

Description

The Rfc3339 class is a character vector that stores Internet Date/time timestamps, formatted as described in RFC3339. It provides a faithful representation of any RFC3339-compliant string in an R session.

Usage

```
as.Rfc3339(x)

## S3 method for class 'character'
as.Rfc3339(x)

## Default S3 method:
as.Rfc3339(x)

## S3 method for class 'POSIXt'
as.Rfc3339(x)
```

```
## S3 method for class 'Rfc3339'
as.character(x, ...)

is.Rfc3339(x)

## S3 method for class 'Rfc3339'
as.POSIXct(x, tz = "", ...)

## S3 method for class 'Rfc3339'
as.POSIXlt(x, tz = "", ...)

## S3 method for class 'Rfc3339'
x[i]

## S3 method for class 'Rfc3339'
x[[i]]

## S3 replacement method for class 'Rfc3339'
x[i] <- value

## S3 replacement method for class 'Rfc3339'
x[[i]] <- value

## S3 method for class 'Rfc3339'
c(..., recursive = TRUE)

## S4 method for signature 'Rfc3339'
saveObject(x, path, ...)
```

Arguments

x	For `as.Rfc3339` methods, object to be coerced to an Rfc3339 instance. For the subset and combining methods, an Rfc3339 instance. For `as.character`, `as.POSIXlt` and `as.POSIXct` methods, an Rfc3339 instance. For `is.Rfc3339`, any object to be tested for Rfc3339-ness.
tz, recursive, ...	Further arguments to be passed to individual methods.
i	Indices specifying elements to extract or replace.
value	Replacement values, either as another Rfc3339 instance, a character vector or something that can be coerced into one.
path	String containing the path to a directory in which to save `x`.

Details

This class is motivated by the difficulty in using the various POSIXt classes to faithfully represent any RFC3339-compliant string. In particular:

- The POSIXt classes do not automatically capture the string's timezone offset, instead converting all times to the local timezone. This is problematic as it discards information about the original timezone. Technically, the POSIXlt class is capable of holding this information in the `gmtoff` field but it is not clear how to set this.

- There is no way to distinguish between the timezones `Z` and `+00:00`. These are functionally the same but will introduce differences in the checksums of saved files and thus interfere with deduplication mechanisms in storage backends.
- Coercion of POSIXt classes to strings may print more or fewer digits in the fractional seconds than what was present in the original string. Functionally, this is probably unimportant but will still introduce differences in the checksums.

By comparison, the Rfc3339 class preserves all information in the original string, avoiding unexpected modifications from a roundtrip through `readObject` and `saveObject`. This is especially relevant for strings that were created from other languages, e.g., Node.js Date's ISO string conversion uses `Z` by default.

That said, users should not expect too much from this class. It is only used to provide a faithful representation of RFC3339 strings, and does not support any time-related arithmetic. Users are advised to convert to POSIXct or similar if such operations are required.

Value

For `as.Rfc3339`, the subset and combining methods, an Rfc3339 instance is returned.

For the other `as.*` methods, an instance of the corresponding type generated from an Rfc3339 instance.

Author(s)

Aaron Lun

Examples

```
out <- as.Rfc3339(Sys.time() + 1:10)
out

out[2:5]
out[2] <- "2"
c(out, out)

as.character(out)
as.POSIXct(out)
```

---

saveAtomicVector *Save atomic vectors to disk*

Description

Save vectors containing atomic elements (or values that can be cast as such, e.g., dates and times) to an on-disk representation.

Usage

```
## S4 method for signature 'integer'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'character'
saveObject(x, path, character.vls = FALSE, ...)
```

```
## S4 method for signature 'logical'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'double'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'numeric'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'Date'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'POSIXlt'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'POSIXct'
saveObject(x, path, character.vls = FALSE, ...)

## S4 method for signature 'numeric_version'
saveObject(x, path, ...)
```

Arguments

x	Any of the atomic vector types, or Date objects, or time objects, e.g., POSIXct.
path	String containing the path to a directory in which to save x.
character.vls	Logical scalar indicating whether to save character vectors in the custom variable length string (VLS) array format. If NULL, this is determined based on a comparison of the expected storage against a fixed length array.
...	Further arguments that are ignored.

Value

x is saved inside path. NULL is invisibly returned.

Author(s)

Aaron Lun

See Also

readAtomicVector, to read the files back into the session.

Examples

```
tmp <- tempfile()
dir.create(tmp)
saveObject(LETTERS, file.path(tmp, "foo"))
saveObject(setNames(runif(26), letters), file.path(tmp, "bar"))
list.files(tmp, recursive=TRUE)
```

saveBaseFactor *Save a factor to disk*

Description

Pretty much as it says, let's save a base R factor to an on-disk representation.

Usage

```
## S4 method for signature 'factor'
saveObject(x, path, ...)
```

Arguments

x	A factor.
path	String containing the path to a directory in which to save x.
...	Further arguments that are ignored.

Value

x is saved inside path. NULL is invisibly returned.

Author(s)

Aaron Lun

See Also

readBaseFactor, to read the files back into the session.

Examples

```
tmp <- tempfile()
saveObject(factor(1:10, 1:30), tmp)
list.files(tmp, recursive=TRUE)
```

saveBaseList *Save a base list to disk*

Description

Save a list or List to a JSON or HDF5 file, with extra files created for any of the more complex list elements (e.g., DataFrames, arrays). This uses the uzuki2 specification to ensure that appropriate types are declared.

Usage

```
## S4 method for signature 'list'
saveObject(
  x,
  path,
  list.format = saveBaseListFormat(),
  list.character.vls = NULL,
  ...
)

## S4 method for signature 'List'
saveObject(x, path, list.format = saveBaseListFormat(), ...)

saveBaseListFormat(list.format)
```

Arguments

x	An ordinary R list, named or unnamed. Alternatively, a List to be coerced into a list.
path	String containing the path to a directory in which to save x.
list.format	String specifying the format in which to save the list.
list.character.vls	Logical scalar indicating whether to save character vectors in the custom variable length string (VLS) array format. If NULL, this is determined based on a comparison of the expected storage against a fixed length array. Only used if list.format="hdf5".
...	Further arguments, passed to altSaveObject for complex child objects.

Value

For the saveObject method, x is saved inside dir. NULL is invisibly returned.

For saveBaseListFormat; if list.format is missing, a string containing the current format is returned. If list.format is supplied, it is used to define the current format, and the *previous* format is returned.

File formats

If list.format="json.gz" (default), the list is saved to a Gzip-compressed JSON file (the default). This is an easily parsed format with low storage overhead.

If list.format="hdf5", x is saved into a HDF5 file instead. This format is most useful for random access and for preserving the precision of numerical data.

Storing scalars

The **uzuki2** specification (see https://github.com/ArtifactDB/uzuki2) allows length-1 vectors to be stored as-is or as a scalar. If a list element is of length 1, saveBaseList will store it as a scalar on-disk, effectively "unboxing" it for languages with a concept of scalars. Users can override this behavior by adding the AsIs class to the affected list element, which will force storage as a length-1 vector. This reflects the decisions made by readBaseList and mimics the behavior of packages like **jsonlite**.

Author(s)

Aaron Lun

See Also

https://github.com/ArtifactDB/uzuki2 for the specification.

readBaseList, to read the list back into the R session.

Examples

```
library(S4Vectors)
ll <- list(A=1, B=LETTERS, C=DataFrame(X=1:5))

tmp <- tempfile()
saveObject(ll, tmp)
list.files(tmp, recursive=TRUE)
```

---

saveFormats *Choose the format for certain objects*

Description

Alter the format used to save DataFrames in its stageObject methods.

Usage

```
saveDataFrameFormat(format)
```

Arguments

format	String containing the format to use. Tbe "csv", "csv.gz" (default) or "hdf5". Alternatively NULL, to use the default format.

Details

stageObject methods will treat a format=NULL in the same manner as the default format. The distinction exists to allow downstream applications to set their own defaults while still responding to user specification. For example, an application can detect if the existing format is NULL, and if so, apply another default via .saveDataFrameFormat. On the other hand, if the format is not NULL, this is presumably specified by the user explicitly and should be respected by the application.

Value

If format is missing, a string containing the current format is returned, or NULL to use the default format.

If format is supplied, it is used to define the current format, and the *previous* format is returned.

Author(s)

Aaron Lun

Examples

```
(old <- .saveDataFrameFormat())

.saveDataFrameFormat("hdf5")
.saveDataFrameFormat()

# Setting it back.
.saveDataFrameFormat(old)
```

---

saveMetadata *Save R-level metadata to disk*

Description

Save metadata and mcols for Annotated or Vector objects, respectively, to disk. These are typically used inside saveObject methods for concrete subclasses.

Usage

```
saveMetadata(x, metadata.path, mcols.path, ...)
```

Arguments

x	A Vector or Annotated object.
metadata.path	String containing the path in which to save the metadata. If NULL, no metadata is saved.
mcols.path	String containing the path in which to save the mcols. If NULL, no mcols is saved.
...	Further arguments to be passed to altSaveObject.

Details

If mcols(x) has no columns, nothing is saved by saveMcols. Similarly, if metadata(x) is an empty list, nothing is saved by saveMetadata. This avoids creating unnecessary files with no meaningful content.

If mcols(x) has non-NULL row names, these are removed prior to staging. These names are usually redundant with the names associated with elements of x itself.

Value

The metadata for x is saved to metadata.path, and similarly for the mcols.

Author(s)

Aaron Lun

See Also

readMetadata, which restores metadata to the object.

`saveObject` *Save objects to disk*

Description

Generic to save assorted R objects into appropriate on-disk representations. More methods may be defined by other packages to extend the **alabaster.base** framework to new classes.

Usage

```
saveObject(x, path, ...)
```

Arguments

`x`	A Bioconductor object of the specified class.
`path`	String containing the path to a directory in which to save `x`.
`...`	Additional named arguments to pass to specific methods.

Value

`dir` is created and populated with files containing the contents of `x`. `NULL` should be invisibly returned.

Comments for extension developers

Methods for the `saveObject` generic should create a directory at `path` in which the contents of `x` are to be saved. The files may consist of any format, though language-agnostic formats like HDF5, CSV, JSON are preferred. For more complex objects, multiple files and subdirectories may be created within `path`. The only strict requirements are:

- There must be an `OBJECT` file inside `path`, containing a JSON object with a `"type"` string property that specifies the class of the object, e.g., `"data_frame"`, `"summarized_experiment"`. This will be used by loading functions to determine how to load the files into memory.
- The names of files and subdirectories should not start with `_` or `..` These are reserved for applications, e.g., to build manifests or to store additional metadata.

Callers can pass optional parameters to specific `saveObject` methods via `....` Any options recognized by a method should be prefixed by the name of the class used in the method's signature, e.g., any options for `saveObject,DataFrame-method` should start with `DataFrame.`. This scoping avoids conflicts between otherwise identically-named options of different methods.

When developing `saveObject` methods of complex objects, a simple approach is to decompose `x` into its "child" components. Each component can then be saved into a subdirectory of `path`, levering the existing `saveObject` methods for the component classes. In such cases, extension developers should actually call `altSaveObject` on each child component, rather than calling saveObject directly. This ensures that any application-level overrides of the loading functions are respected. It is expected that each method will forward `...` (possibly after modification) to any internal `altSaveObject` calls.

Comments for application developers

Application developers can override `saveObject` by specifying a custom function in `altSaveObject`. This can be used to point to a different function to handle the saving process for each class. The custom function can be as simple as a wrapper around `saveObject` with some additional actions (e.g., to save more metadata), or may be as complex as a full-fledged generic with its own methods for class-specific customizations.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
X <- DataFrame(X=LETTERS, Y=sample(3, 26, replace=TRUE))

tmp <- tempfile()
saveObject(X, tmp)
list.files(tmp, recursive=TRUE)
```

`saveObject,DataFrame-method` *Save a DataFrame to disk*

Description

Stage a DataFrame by saving it to a HDF5 file.

Usage

```
## S4 method for signature 'DataFrame'
saveObject(x, path, DataFrame.character.vls = NULL, ...)

## S4 method for signature 'data.frame'
saveObject(x, path, DataFrame.character.vls = NULL, ...)
```

Arguments

`x`	A DataFrame or data.frame.
`path`	String containing the path to a directory in which to save `x`.
`DataFrame.character.vls`	Logical scalar indicating whether to save character vectors in the custom variable length string (VLS) array format. If `NULL`, this is determined based on a comparison of the expected storage against a fixed length array.
`...`	Additional named arguments to pass to specific methods.

Details

This method creates a `basic_columns.h5` file that contains columns for atomic vectors, factors, dates and date-times. Dates and date-times are converted to character vectors and saved as such inside the file. Factors are saved as a HDF5 group with both the codes and the levels as separate datasets.

Any non-atomic columns are saved to a `other_columns` subdirectory inside `path` via `saveObject`, named after its zero-based positional index within `x`.

If `metadata` or `mcols` are present, they are saved to the `other_annotations` and `column_annotations` subdirectories, respectively, via `saveObject`.

In the on-disk representation, no distinction is made between DataFrame and data.frame instances of `x`. Calling `readDataFrame` will always produce a DFrame regardless of the class of `x`.

Value

A named list containing the metadata for `x`. `x` itself is written to a HDF5 file inside `path`. Additional files may also be created inside `path` and referenced from the metadata.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
saveObject(df, tmp)
list.files(tmp, recursive=TRUE)
```

---

saveObject,DataFrameFactor-method

Stage a DataFrameFactor object

Description

Stage a DataFrameFactor object, a generalization of the base factor where each level is a row of a DataFrame.

Usage

```
## S4 method for signature 'DataFrameFactor'
saveObject(x, path, ...)
```

Arguments

`x`	A DataFrameFactor object.
`path`	String containing the path to a directory in which to save `x`.
`...`	Further arguments, to pass to internal `altSaveObject` calls.

Value

x is saved to an on-disk representation inside path.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(X=LETTERS[1:5], Y=1:5)
out <- DataFrameFactor(df[sample(5, 100, replace=TRUE),,drop=FALSE])

tmp <- tempfile()
saveObject(out, tmp)
list.files(tmp, recursive=TRUE)
```

---

transformVectorForHdf5

Transform a vector to save in a HDF5 file

Description

This handles type casting and missing placeholder value selection/substitution. It is primarily intended for developers of **alabaster.*** extensions.

Usage

```
transformVectorForHdf5(x, .version = 3)
```

Arguments

x	An atomic vector to be saved to HDF5.
.version	Internal use only.

Value

A list containing:

- transformed, the transformed vector. This may be the same as x if no NA values were detected. Note that logical vectors are cast to integers.
- placeholder, the placeholder value used to represent NA values. This is NULL if no NA values were detected in x, otherwise it is the same as the output of chooseMissingPlaceholderForHdf5.

Author(s)

Aaron Lun

Examples

```
transformVectorForHdf5(c(TRUE, NA, FALSE))
transformVectorForHdf5(c(1L, NA, 2L))
transformVectorForHdf5(c(1L, NaN, 2L))
transformVectorForHdf5(c("FOO", NA, "BAR"))
transformVectorForHdf5(c("FOO", NA, "NA"))
```

`validateDirectory` *Validate a directory of objects*

Description

Check whether each object in a directory is valid by calling `validateObject` on each non-child object.

Usage

```
validateDirectory(dir, legacy = NULL, ...)
```

Arguments

`dir`	String containing the path to a directory with subdirectories populated by `saveObject`.
`legacy`	Logical scalar indicating whether to validate a directory with legacy objects (created by the old `stageObject`). If `NULL`, this is auto-detected from the contents of `dir`.
`...`	Further arguments to use when `legacy=TRUE`, for back-compatibility only.

Details

We assume that the process of validating an object will call `validateObject` on any child objects. This allows us to skip explicit calls to `validateObject` on each component of a complex object.

Value

Character vector of the paths inside `dir` that were validated, invisibly. If any validation failed, an error is raised.

Author(s)

Aaron Lun

Examples

```
# Mocking up an object:
library(S4Vectors)
ncols <- 123
df <- DataFrame(
    X = rep(LETTERS[1:3], length.out=ncols),
    Y = runif(ncols)
)
df$Z <- DataFrame(AA = sample(ncols))
```

```
# Mocking up the directory:
tmp <- tempfile()
dir.create(tmp, recursive=TRUE)
saveObject(df, file.path(tmp, "foo"))

# Checking that it's valid:
validateDirectory(tmp)

# Adding an invalid object:
dir.create(file.path(tmp, "bar"))
write(file=file.path(tmp, "bar", "OBJECT"), '[ "WHEEE" ]')
try(validateDirectory(tmp))
```

`validateObject` *Validate an object's on-disk representation*

Description

Validate an object's on-disk representation against the **takane** specifications. This is done by dispatching to an appropriate validation function based on the type in the `OBJECT` file.

Usage

```
validateObject(path, metadata = NULL)

registerValidateObjectFunction(type, fun, existing = c("old", "new", "error"))

registerValidateObjectHeightFunction(
  type,
  fun,
  existing = c("old", "new", "error")
)

registerValidateObjectDimensionsFunction(
  type,
  fun,
  existing = c("old", "new", "error")
)

registerValidateObjectSatisfiesInterface(
  type,
  interface,
  action = c("add", "remove")
)

registerValidateObjectDerivedFrom(type, parent, action = c("add", "remove"))
```

Arguments

`path`	String containing a path to a directory, itself created with a `saveObject` method.

metadata	List containing metadata for the object. If this is not supplied, it is automatically read from the OBJECT file inside path.
type	String specifying the name of type of the object.
fun	For registerValidateObjectFunction, a function that accepts path and metadata, and raises an error if the object at path is invalid. It can be assumed that metadata is a list created by reading OBJECT. For registerValidateObjectHeightFunction, a function that accepts path and metadata, and returns an integer specifying the "height" of the object. This is usually the length for vector-like or 1-dimensional objects, and the extent of the first dimension for higher-dimensional objects. For registerValidateObjectDimensionsFunction, a function that accepts path and metadata, and returns an integer vector specifying the dimensions of the object. This may also be NULL to delete an existing registry from any of the functions mentioned above.
existing	Logical scalar indicating the action to take if a function has already been registered for type - keep the old or new function, or throw an error.
interface	String specifying the name of the interface that is represented by type.
action	String specifying whether to add or remove type from the list of types that implements interface or is derived from parent.
parent	String specifying the parent object from which type is derived.

Value

For validateObject, NULL is returned invisibly upon success, otherwise an error is raised.

For the registerValidObject*Function functions, the supplied fun is added to the corresponding registry for type. If fun = NULL, any existing entry for type is removed; a logical scalar is returned indicating whether removal was performed.

For the registerValidateObjectSatisfiesInterface and registerValidateObjectDerivedFrom functions, type is added to or removed from relevant list of types. A logical scalar is returned indicating whether the type was added or removed - this may be FALSE if type was already present or absent, respectively.

Author(s)

Aaron Lun

See Also

https://github.com/ArtifactDB/takane, for detailed specifications of the on-disk representation for various Bioconductor objects.

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
saveObject(df, tmp)
validateObject(tmp)
```

vls *VLS saving utilities*

Description

Utilities for saving our custom variable length string array format in HDF5. Intended for **alabaster.*** developers only.

writeMetadata *Saving the metadata*

Description

WARNING: this function is deprecated as newer versions of **alabaster** *do not need to write metadata.* Helper function to write metadata from a named list to a JSON file. This is commonly used inside `stageObject` methods to create the metadata file for a child object.

Usage

```
writeMetadata(meta, dir, ignore.null = TRUE)
```

Arguments

`meta`	A named list containing metadata. This should contain at least the `"$schema"` and `"path"` elements.
`dir`	String containing a path to the staging directory.
`ignore.null`	Logical scalar indicating whether `NULL` values should be ignored during coercion to JSON.

Details

Any `NULL` values in `meta` are pruned out prior to writing when `ignore.null=TRUE`. This is done recursively so any `NULL` values in sub-lists of `meta` are also ignored.

Any scalars are automatically unboxed so array values should be explicitly specified as such with `I()`.

Any starting `"./"` in `meta$path` will be automatically removed. This allows staging methods to save in the current directory by setting `path="."`, without the need to pollute the paths with a `"./"` prefix.

The JSON-formatted metadata is validated against the schema in `meta[["$schema"]]` using **jsonvalidate**. The location of the schema is taken from the `package` attribute in that string, if one exists; otherwise, it is assumed to be in the **alabaster.schemas** package. (All schemas are assumed to live in the `inst/schemas` subdirectory of their indicated packages.)

We also use the schema to determine whether `meta` refers to an actual artifact or is a metadata-only document. If it refers to an actual file, we compute its MD5 sum and store it in the metadata for saving. We also save its associated metadata into a JSON file at a location obtained by appending `".json"` to `meta$path`.

For artifacts, the MD5 sum calculation will be skipped if the `meta` already contains a `md5sum` field. This can be useful on some occasions, e.g., to improve efficiency when the MD5 sum was already computed during staging, or if the artifact does not actually exist in its full form on the file system.

Value

A JSON file containing the metadata is created at `path`. A list of resource metadata is returned, e.g., for inclusion as the `"resource"` property in parent schemas.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
dir.create(tmp)
info <- stageObject(df, tmp, path="coldata")
writeMetadata(info, tmp)
cat(readLines(file.path(tmp, "coldata/simple.csv.gz.json")), sep="\n")
```

Index