Package 'zarr'

November 8, 2026

Title Native and Extensible R Driver for 'Zarr'

Version 1.0.1

Description The 'Zarr' specification is widely used to build libraries for the storage and retrieval of n-dimensional array data from data stores ranging from local file systems to the cloud. This package is a native 'Zarr' implementation in R with support for all required features of 'Zarr' version 3. It is designed to be extensible such that new stores, codecs and extensions can be added easily.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 8.0.0

Imports Rcpp (>= 1.0.0), blosc, jsonlite, methods, R6

Suggests bit64, cli, curl, digest, future, future.apply, paws.storage, qs2, testthat (>= 3.0.0), webfakes, zlib

Config/testthat/edition 3

Config/Needs/website rmarkdown

Depends R (>= 4.4)

LinkingTo Rcpp

URL https://github.com/R-CF/zarr, https://r-cf.github.io/zarr/

BugReports https://github.com/R-CF/zarr/issues

NeedsCompilation yes

Author Patrick Van Laake [aut, cre, cph]

Maintainer Patrick Van Laake <patrick@vanlaake.net>

Repository CRAN

Date/Publication 2026-11-08 16:20:02 UTC

Contents

array_builder 3
as_zarr 6

chunk_grid_regular . . . 7
chunk_grid_sharded . . . 9
create_zarr . . . 11
define_array . . . 11
is_valid_node_name . . . 12
open_zarr . . . 13
optimal_chunking . . . 14
reset_options . . . 14
s3_list_dir . . . 15
save_options . . . 16
str.chunk_grid_regular . . . 16
str.chunk_grid_sharded . . . 17
str.zarr . . . 17
str.zarr_array . . . 18
str.zarr_group . . . 18
zarr . . . 19
zarr_array . . . 22
zarr_codec . . . 26
zarr_codec_blosc . . . 27
zarr_codec_bytes . . . 29
zarr_codec_crc32c . . . 30
zarr_codec_gzip . . . 32
zarr_codec_sharding . . . 33
zarr_codec_transpose . . . 34
zarr_codec_ucs4 . . . 35
zarr_codec_vlenutf8 . . . 37
zarr_codec_zstd . . . 38
zarr_convention . . . 39
zarr_conventions . . . 41
zarr_convention_ref . . . 41
zarr_convention_uom . . . 43
zarr_data_type . . . 44
zarr_domain . . . 45
zarr_domains . . . 46
zarr_extension . . . 47
zarr_filter . . . 48
zarr_filter_astype . . . 49
zarr_filter_categorize . . . 50
zarr_filter_checksum . . . 51
zarr_filter_delta . . . 52
zarr_filter_fixedscaleoffset . . . 53
zarr_filter_packbits . . . 54
zarr_filter_shuffle . . . 55
zarr_group . . . 56
zarr_httpstore . . . 59
zarr_localstore . . . 64
zarr_memorystore . . . 69
zarr_node . . . 74

zarr_object 77
zarr_options 80
zarr_register_domain 81
zarr_s3store 81
zarr_store 88
zarr_unregister_domain 94
[.zarr_array 95
[[.zarr 96
[[.zarr_group 96

Index **98**

`array_builder` *Array builder*

Description

This class builds the metadata document for an array to be created or modified. It can also be used to inspect the metadata document of an existing Zarr array.

The Zarr core specification is quite complex for arrays, including codecs and storage transformers that are part optional, part mandatory, and dependent on each other. On top of that, extensions defined outside of the core specification must also be handled in the same metadata document. This class helps construct a valid metadata document, with support for (some) extensions. (If you need support for a specific extension, open an issue on Github.)

When creating array definitions, the default is to use R ordering, meaning that a transpose codec is added with the "order" parameters having the dimensions in reverse order. If you want to use a different ordering, for instance to have a Zarr store with maximum portability, delete the default transpose codec or set its ordering to the desired values. Note that the shape and chunk_shape parameters are set in reference to the ordering in the transpose codec (or the default 0, 1, ... is no transpose codec is present), but that within the R environment the shape and chunk shape are always set to R ordering. This is necessary to be able to apply array operations on the data in R.

This class does not care about the "chunk_key_encoding" parameter. This is addressed at the level of the store.

The "codecs" parameter has a default first codec of "transpose". This ensures that R matrices and arrays can be stored in native column-major order with the store still accessible to environments that use row-major order by default, such as Python. A second default codec is "bytes" that records the endianness of the data. Other codecs may be added by the user, such as a compression codec.

This class only handles the mandatory metadata in a Zarr array document. Optional arguments may be set directly on the Zarr array after it has been created.

Active bindings

`format` The Zarr format to build the metadata for. The value must be 3. After changing the format, many fields will have been reset to a default value.

portable Logical flag to indicate if the array is specified for maximum portability across environments (e.g. Python, Java, C++). Default is FALSE. Setting the portability to TRUE implies that R data will be permuted before writing the array to the store. A value of FALSE is therefore more efficient.

data_type The data type of the Zarr array. After changing the format, many fields will have been reset to a default value.

data_type_from_storage.mode Set the data type of the Zarr array from the storage.mode() of the data to be written to the array with this array_builder instance. This is effectively a write-only field: reading the value will just return the Zarr data type, not the R type - use zarr_v3_datatypes[[<zarr_data_type>]]$Rtype for that. After setting the format, many fields will have been reset to a default value.

fill_value The value in the array of uninitialized data elements. The fill_value has to agree with the data_type of the array.

shape The shape of the Zarr array, an integer vector of lengths along the dimensions of the array. Setting the shape will reset the chunking settings to their default values.

chunk_shape The shape of each individual chunk in which to store the Zarr array. When setting, pass in an integer vector of lengths of the same size as the shape of the array. The shape of the array must be set before setting this. When reading, returns an instance of class chunk_grid_regular if the shape field has been set, NULL otherwise.

codec_info (read-only) Retrieve a data.frame of registered codec modes and names for this array.

codecs (read-only) A list with validated and instantiated codecs for processing data associated with this array.

Methods

Public methods:

- array_builder$new()
- array_builder$print()
- array_builder$metadata()
- array_builder$add_codec()
- array_builder$remove_codec()
- array_builder$is_valid()

array_builder$new(): Create a new instance of the array_builder class. Optionally, a metadata document may be passed in as an argument to inspect the definition of an existing Zarr array, or to use as a template for a new metadata document.

Usage:

```
array_builder$new(metadata = NULL)
```

Arguments:

metadata Optional. A JSON metadata document or list of metadata from an existing Zarr array. This document will not be modified through any operation in this class.

Returns: An instance of this class.

array_builder$print(): Print the array metadata to the console.

Usage:

```
array_builder$print()
```

`array_builder$metadata()`: Retrieve the metadata document to create a Zarr array.

Usage:

```
array_builder$metadata(format = "list")
```

Arguments:

`format` Either "list" or "JSON".

Returns: The metadata document in the requested format.

`array_builder$add_codec()`: Adds a codec at the end of the currently registered codecs. Optionally, the `.position` argument may be used to indicate a specific position of the codec in the list. Codecs can only be added if their mode agrees with the mode of existing codecs - if this codec does not agree with the existing codecs, a warning will be issued and the new codec will not be registered.

Usage:

```
array_builder$add_codec(codec, configuration, .position = NULL)
```

Arguments:

`codec` The name of the codec. This must be a registered codec with an implementation that is available from this package.

`configuration` List with configuration parameters of the `codec`. May be `NULL` or `list()` for codecs that do not have configuration parameters.

`.position` Optional, the 1-based position where to insert the codec in the list. If the number is larger than the list, the codec will be appended at the end of the list of codecs.

Returns: Self, invisibly.

`array_builder$remove_codec()`: Remove a codec from the list of codecs for the array. A codec cannot be removed if the remaining codecs do not form a valid chain due to mode conflicts.

Usage:

```
array_builder$remove_codec(codec)
```

Arguments:

`codec` The name of the codec to remove, a single character string.

`array_builder$is_valid()`: This method indicates if the current specification results in a valid metadata document to create a Zarr array.

Usage:

```
array_builder$is_valid()
```

Returns: `TRUE` if a valid metadata document can be generated, `FALSE` otherwise.

as_zarr *Convert an R object into a Zarr array*

Description

This function creates a Zarr object from an R vector, matrix or array. Default settings will be taken from the R object (data type, shape). Data is chunked into chunks of length 100 (or less if the array is smaller) and compressed.

Usage

```
as_zarr(x, name = "", location = NULL)
```

Arguments

x	The R object to convert. Must be a vector, matrix or array of a numeric, character or logical type.
name	Optional. The name of the Zarr array to be created. If omitted, an array will be created at the root of the Zarr store.
location	Optional. If supplied, either an existing zarr_group in a Zarr object, or a character string giving the location on a local file system where to persist the data. If the argument is a zarr_group, argument name must be provided. If the argument gives the location for a new Zarr store then the location must be writable by the calling code. As per the Zarr specification, it is recommended to use a location that ends in ".zarr" when providing a location for a new store. If argument name is given then the Zarr array will be created in the root of the Zarr store with that name. If the name argument is not given, a single-array Zarr store will be created. If the location argument is not given, a Zarr object is created in memory.

Value

If the location argument is a zarr_group, the new Zarr array is returned. Otherwise, the zarr object that is newly created and which contains the Zarr array, or an error if the zarr object could not be created.

Examples

```
x <- array(1:400, c(5, 20, 4))
z <- as_zarr(x)
z
```

chunk_grid_regular *Chunk management*

Description

This class implements the regular chunk grid for Zarr arrays, in which all chunks have the same shape. Chunks along the high end of a dimension extend beyond the array when the array shape is not a multiple of the chunk shape. It manages reading from and writing to Zarr stores, using the codecs for data transformation.

Super classes

zarr_extension -> chunking -> chunk_grid_regular

Active bindings

chunk_shape (read-only) The dimensions of each chunk in the chunk grid of the associated array.

codecs The list of codecs used by the chunking scheme. These are set by the array when starting to use chunking for file I/O. Upon reading, the list of registered codecs.

Methods

Public methods:

- chunk_grid_regular$new()
- chunk_grid_regular$print()
- chunk_grid_regular$metadata_fragment()
- chunk_grid_regular$resize()
- chunk_grid_regular$read()
- chunk_grid_regular$write()
- chunk_grid_regular$flush()

chunk_grid_regular$new(): Initialize a new chunking scheme for an array.

Usage:

```
chunk_grid_regular$new(array_shape, chunk_shape)
```

Arguments:

array_shape Integer vector of the array dimensions. This may be NA for a scalar array.

chunk_shape Optional. Integer vector of the dimensions of each chunk. If omitted, the optimal chunking is automatically determined. Ignored for a scalar array.

Returns: An instance of chunk_grid_regular.

chunk_grid_regular$print(): Print a short description of this chunking scheme to the console.

Usage:

```
chunk_grid_regular$print()
```

Returns: Self, invisibly.

`chunk_grid_regular$metadata_fragment()`: Return the metadata fragment that describes this chunking scheme.

Usage:

```
chunk_grid_regular$metadata_fragment()
```

Returns: A list with the metadata of this chunking scheme.

`chunk_grid_regular$resize()`: Resize the array, growing or shrinking any combination of dimensions at either end in one pass. Chunks are renamed in the store when their grid index moves and erased when they fall outside the new grid. No chunk payload is re-encoded except for a chunk left straddling a shrinking, non-chunk-aligned high-end boundary, whose excess elements are set to the fill value.

Because chunks have a fixed position on the grid, `low` can only move in whole chunks: values are rounded outward to the nearest chunk (more space added when growing, less removed when shrinking). `high` is not constrained this way.

Usage:

```
chunk_grid_regular$resize(low, high)
```

Arguments:

`low, high` Integer vectors, one element per array dimension. Positive grows that end, negative shrinks it, `0` leaves it unchanged.

Details: An error is thrown when the resize would produce a non-positive extent along any dimension, or when the array would shrink to a high boundary that is not aligned to the chunk boundary while the chunking scheme cannot clip chunks.

Returns: Self, invisibly.

`chunk_grid_regular$read()`: Read data from the Zarr array into an R object. The read can span multiple chunks. Reads will be parallelised if `future::plan(multisession)` is set; by default the reading is sequential.

Usage:

```
chunk_grid_regular$read(start, stop)
```

Arguments:

`start, stop` Integer vectors of the same length as the dimensionality of the Zarr array, indicating the starting and ending (inclusive) indices of the data along each axis. These are ignored if the Zarr array is a scalar.

Returns: A vector, matrix or array of data.

`chunk_grid_regular$write()`: Write data to the array. Writing data always uses a sequential plan.

Usage:

```
chunk_grid_regular$write(data, start, stop, flush = TRUE)
```

Arguments:

`data` An R object with the same dimensionality as the Zarr array.

`start, stop` Integer vectors of the same length as the dimensionality of the Zarr array, indicating the starting and ending (inclusive) indices of the data along each axis. Ignored for a scalar array.

`flush` Logical, default is `TRUE`. Should the chunks that have been written to be flushed to the store (`TRUE`), or should they be left stale for further writes to the same chunk (`FALSE`). Leaving the chunks stale for further writing leads to better performance when chunks are written to multiple times. Call the `flush()` method to persist data in stale chunks.

Returns: Self, invisibly.

`chunk_grid_regular$flush()`**:** Persist the data in all chunks with pending edits to the store.

Usage:

```
chunk_grid_regular$flush()
```

`chunk_grid_sharded` *Sharding chunk management*

Description

This class implements the sharded chunk grid for Zarr arrays. It manages reading from Zarr stores, using the codecs for data transformation included in the sharding configuration. Writing is not supported with this codec. Storing a scalar array in a sharded grid is not possible either and totally useless anyway.

The sharding codec requires a regular grid of shards, so the geometry of the shard grid is inherited from chunk_grid_regular, with the chunk shape being the shape of a shard.

Super classes

`zarr_extension` -> `chunking` -> `chunk_grid_regular` -> `chunk_grid_sharded`

Active bindings

`inner_shape` (read-only) The dimensions of each chunk in the shard.

`codecs` (read-only) The list of codecs used by the sharding scheme.

`index_codecs` (read-only) The list of codecs used by the sharding scheme for the indexing of the internal chunks.

Methods

Public methods:

- `chunk_grid_sharded$new()`
- `chunk_grid_sharded$print()`
- `chunk_grid_sharded$metadata_fragment()`
- `chunk_grid_sharded$read()`
- `chunk_grid_sharded$write()`

`chunk_grid_sharded$new()`: Initialize a new sharded chunking scheme for an array.

Usage:

```
chunk_grid_sharded$new(
  array_shape,
  chunk_shape,
  inner_shape,
  index_loc,
  inner_codecs,
  index_codecs
)
```

Arguments:

`array_shape` Integer vector of the array dimensions.

`chunk_shape` Integer vector of the dimensions of each outer chunk, i.e. the size of a shard.

`inner_shape` Integer vector of the dimensions of each inner chunk, i.e. the size of a single chunk inside a shard.

`index_loc` Location of the shard index in the shard file, either "start" or "end".

`inner_codecs, index_codecs` List of `zarr_codec` instances to decode the inner chunks and the index, respectively.

Returns: An instance of `chunk_grid_sharded`.

`chunk_grid_sharded$print()`: Print a short description of this sharded chunking scheme to the console.

Usage:

```
chunk_grid_sharded$print()
```

Returns: Self, invisibly.

`chunk_grid_sharded$metadata_fragment()`: Return the metadata fragment that describes this chunking scheme.

Usage:

```
chunk_grid_sharded$metadata_fragment()
```

Returns: A list with the metadata of this chunking scheme.

`chunk_grid_sharded$read()`: Read data from the Zarr array into an R object.

Usage:

```
chunk_grid_sharded$read(start, stop)
```

Arguments:

`start, stop` Integer vectors of the same length as the dimensionality of the Zarr array, indicating the starting and ending (inclusive) indices of the data along each axis.

Returns: A vector, matrix or array of data.

`chunk_grid_sharded$write()`: Writing to a sharded array is not supported.

Usage:

```
chunk_grid_sharded$write(...)
```

Arguments:

`...` Ignored.

Details: This method always throws an error.

create_zarr *Create a Zarr store*

Description

This function creates a Zarr v.3 instance. The root of the Zarr store will be a group to which other groups or arrays can be added.

Usage

```
create_zarr(location)
```

Arguments

location	Optional. Character string that indicates a location on a file system where the data in the Zarr object will be persisted in a Zarr store in a directory. The character string may contain UTF-8 characters and/or use a file URI format. The Zarr specification recommends that the location use the ".zarr" extension to identify the location as a Zarr store. If missing, a Zarr store will be created in memory.

Value

A zarr object.

Examples

```
fn <- tempfile(fileext = ".zarr")
my_zarr_object <- create_zarr(fn)
my_zarr_object$store$root
unlink(fn)
```

define_array *Define the properties of a new Zarr array.*

Description

With this function you can create a skeleton Zarr array from some key properties and a number of derived properties. Compression of the data is set to a default algorithm and level. This function returns an array_builder instance with which you can create directly the Zarr array, or set further properties before creating the array.

Usage

```
define_array(data_type, shape)
```

Arguments

data_type	The data type of the Zarr array.
shape	An integer vector giving the length along each dimension of the array.

Value

A array_builder instance with which a Zarr array can be created.

Examples

```
x <- array(1:120, c(3, 8, 5))
def <- define_array("int32", dim(x))
def$chunk_shape <- c(4, 4, 4)
z <- create_zarr() # Creates a Zarr object in memory
arr <- z$add_array("/", "my_array", def)
arr$write(x)
arr
```

---

is_valid_node_name *Check if the name of a node is valid in Zarr*

Description

From the Zarr specification, the following constraints apply to node names:

- must not be the empty string (""), except for the root node
- must not be a string composed only of period characters, e.g. "." or ".."
- must not start with the reserved prefix "__".

Only punctuation characters in the set -, _, . are allowed. As an extension to the Zarr specification, characters and numbers can be any UTF-8 code point. When portability is an issue, restrict characters and numbers to the set A-Za-z0-9.

Usage

```
is_valid_node_name(name)
```

Arguments

name	Character string with a node name to check.

Value

TRUE if the node name is valid, FALSE otherwise.

Examples

```
is_valid_node_name("simple_name")
is_valid_node_name("no spaces allowed!")
```

`open_zarr` *Open a Zarr store*

Description

This function opens a Zarr object, connected to a store located on the local file system or on a remote server using the HTTP or S3 protocol. The Zarr object can be either v.2 or v.3.

Usage

```
open_zarr(location, read_only = NULL, protocol = NULL, ...)
```

Arguments

`location`	Character string that indicates a location on a file system or a HTTP or S3 server where the Zarr store is to be found. The character string may contain UTF-8 characters and/or use a file URI format.
`read_only`	Optional. Logical that indicates if the store is to be opened in read-only mode. Default is `NULL`, which implies `FALSE` for a local file system store, `TRUE` otherwise. A Zarr version 2 store is always opened in read-only mode.
`protocol`	Optional, character string. Override automatic protocol detection ('local', 'http', or 's3'). Needed for S3-compatible endpoints that aren't AWS and don't follow AWS's hostname conventions (MinIO, EMBASSY Cloud, Ceph RGW, etc.) - there's no reliable way to recognize these from the URL alone, you have to indicate so explicitly rather than have `open_zarr()` parse the location.
`...`	Additional protocol-specific parameters passed through to the underlying store constructor. For `s3://` and S3 `https://` locations, this includes `region`, `profile`, `access_key`/`secret_key`/ `session_token`, `endpoint`, and `anonymous` — see zarr_s3store. Ignored for local and plain HTTP locations.

Value

A zarr object.

Examples

```
fn <- system.file("extdata", "africa.zarr", package = "zarr")
africa <- open_zarr(fn)
africa
```

optimal_chunking *Get optimal chunking size for an array.*

Description

This function will determine the optimal chunking sizes of the array dimensions based on weights per dimension.

Usage

```
optimal_chunking(dim_sizes, weights, chunk_values = 4L * 1024L * 1024L)
```

Arguments

dim_sizes	Integer array of dimension lengths, corresponding to the shape of the array.
weights	Optional, numeric vector with weights per dimension, in the same order as dim_sizes. If omitted, each dimension will have a weight of 1L, i.e. no preferential chunking on any dimension.
chunk_values	Optional, integer value given the maximum number of array elements per chunk. Default is 4 million, meaning that the chunk size of float32 data is at most 16MB uncompressed.

Value

An integer vector with chunk length per dimension in the same order as argument dim_sizes.

Examples

```
shape <- c(x = 50000L, y = 350L, time = 8192)

# Default chunking, approaching the maximum chunk size
optimal_chunking(dim_sizes = shape)

# Prioritize extractions over the "time" dimension
optimal_chunking(dim_sizes = shape, weights = c(1, 1, 2))
```

reset_options *Reset all options to their default value*

Description

This functions resets the options of the zarr package to their default value.

Usage

```
reset_options()
```

Value

Nothing.

Examples

```
reset_options()
```

`s3_list_dir` *List objects/prefixes in an S3 bucket*

Description

Lists the immediate contents under a prefix in an S3 bucket, without requiring a valid Zarr store to be present at that prefix. Useful for discovering the actual store path(s) within a bucket that hosts multiple datasets, before calling `open_zarr()` or `zarr_s3store$new()`.

Usage

```
s3_list_dir(
  bucket,
  prefix = "",
  region = NULL,
  profile = NULL,
  access_key = NULL,
  secret_key = NULL,
  session_token = NULL,
  endpoint = NULL,
  path_style = NULL,
  anonymous = NULL
)
```

Arguments

`bucket`	Character string. The S3 bucket name.
`prefix`	Character string. The prefix to list under. Default `""`.
`region, profile, access_key, secret_key, session_token, endpoint, anonymous, path_style`	As in zarr_s3store; same credential resolution and anonymous default.

Value

A character vector of keys/prefixes found immediately below `prefix`.

Examples

```
# OME test data
if (requireNamespace("paws.storage", quietly = TRUE))
 (prefix <- s3_list_dir("ome-zarr-scivis", region = "us-east-1"))
```

save_options *Save the options to a file in the user config directory*

Description

This functions saves the options of the zarr package in a RDS file in the config directory of the user. This will persist modified options over multiple sessions. The file is automatically read when a new session starts. Note that saving the options is only necessary when any options have been modified and these changes need to apply to future sessions as well.

Usage

```
save_options()
```

Value

Nothing.

Examples

```
save_options()
```

str.chunk_grid_regular *Compact display of a regular chunk grid*

Description

Compact display of a regular chunk grid

Usage

```
## S3 method for class 'chunk_grid_regular'
str(object, ...)
```

Arguments

object	A chunk_grid_regular instance.
...	Ignored.

Examples

```
fn <- system.file("extdata", "africa.zarr", package = "zarr")
africa <- open_zarr(fn)
tas <- africa[["/tas"]]
str(tas$chunking)
```

str.chunk_grid_sharded *Compact display of a sharding chunk grid*

Description

Compact display of a sharding chunk grid

Usage

```
## S3 method for class 'chunk_grid_sharded'
str(object, ...)
```

Arguments

object	A chunk_grid_sharded instance.
...	Ignored.

str.zarr *Compact display of a Zarr object*

Description

Compact display of a Zarr object

Usage

```
## S3 method for class 'zarr'
str(object, ...)
```

Arguments

object	A zarr instance.
...	Ignored.

Examples

```
fn <- system.file("extdata", "africa.zarr", package = "zarr")
africa <- open_zarr(fn)
str(africa)
```

str.zarr_array *Compact display of a Zarr array*

Description

Compact display of a Zarr array

Usage

```
## S3 method for class 'zarr_array'
str(object, ...)
```

Arguments

object	A zarr_array instance.
...	Ignored.

Examples

```
fn <- system.file("extdata", "africa.zarr", package = "zarr")
africa <- open_zarr(fn)
tas <- africa[["/tas"]]
str(tas)
```

str.zarr_group *Compact display of a Zarr group*

Description

Compact display of a Zarr group

Usage

```
## S3 method for class 'zarr_group'
str(object, ...)
```

Arguments

object	A zarr_group instance.
...	Ignored.

Examples

```
fn <- system.file("extdata", "africa.zarr", package = "zarr")
africa <- open_zarr(fn)
root <- africa[["/"]]
str(root)
```

zarr *Zarr object*

Description

This class implements a Zarr object. A Zarr object is a set of objects that make up an instance of a Zarr data set, irrespective of where it is located. The Zarr object manages the hierarchy as well as the underlying store.

A Zarr object may contain multiple Zarr arrays in a hierarchy. The main class for managing Zarr arrays is zarr_array. The hierarchy is made up of zarr_group instances. Each `zarr_array` is located in a `zarr_group`.

Active bindings

`version` (read-only) The version of the Zarr object.

`root` The root node of the Zarr object, usually a zarr_group instance but it could also be a zarr_array instance. CAUTION: When setting the root node, the entire existing hierarchy is deleted. The hierarchy will likely be out of sync with the store after setting the root node.

`store` (read-only) The store of the Zarr object.

`domain` (read-only) The `zarr_domain` instance managing the data in this `zarr` object.

`groups` (read-only) Retrieve the paths to the groups of the Zarr object, starting from the root group, as a character vector.

`arrays` (read-only) Retrieve the paths to the arrays of the Zarr object, starting from the root group, as a character vector.

Methods

Public methods:

- `zarr$new()`
- `zarr$print()`
- `zarr$hierarchy()`
- `zarr$get_node()`
- `zarr$add_group()`
- `zarr$add_array()`
- `zarr$delete_group()`
- `zarr$delete_array()`
- `zarr$save_to()`
- `zarr$clone()`

`zarr$new()`: Create a new Zarr instance. The Zarr instance manages the groups and arrays in the Zarr store that it refers to. This instance provides access to all objects in the Zarr store.

This method can open any Zarr store located on systems with a supported protocol ('local', 's3', 'http').

This method can also create a new Zarr store on a local file system or in memory. It is not possible to create a new store on S3 or an a web server. The newly created store is uninitialised, it is an empty directory. Either assign a zarr_array to the `root` field, or create a root zarr_group to make the store valid and usable.

Usage:

```
zarr$new(store, read_only = NULL, protocol = NULL, ...)
```

Arguments:

`store` Optional. Either an instance of a zarr_store descendant class where the Zarr objects are located, or character string that indicates a location on a file system or a HTTP or S3 server where the Zarr store is to be found. The character string may contain UTF-8 characters and/or use a file URI format. On a local file system the Zarr store will be created if it does not exist. If omitted, an in-memory Zarr store will be created.

`read_only` Optional. Logical that indicates if the store is to be opened in read-only mode. Default is `NULL`, which implies `FALSE` for a local file system and memory store, `TRUE` otherwise.

`protocol` Optional, character string. Override automatic protocol detection ('local', 'http', or 's3'). Needed for S3-compatible endpoints that aren't AWS and don't follow AWS's hostname conventions (MinIO, EMBASSY Cloud, Ceph RGW, etc.) - there's no reliable way to recognize these from the URL alone, you have to indicate so explicitly rather than have this method parse the location.

`...` Additional protocol-specific parameters passed through to the underlying store constructor. For `s3://` and S3 `https://` locations, this includes `region`, `profile`, `access_key`/`secret_key`/`session_token`, `endpoint`, and `anonymous` — see zarr_s3store. Ignored for memory, local and plain HTTP locations.

Returns: A `zarr` object.

`zarr$print()`: Print a summary of the Zarr object to the console.

Usage:

```
zarr$print()
```

Returns: Self, invisibly.

`zarr$hierarchy()`: Print the Zarr hierarchy to the console.

Usage:

```
zarr$hierarchy()
```

`zarr$get_node()`: Retrieve the group or array represented by the node located at the path.

Usage:

```
zarr$get_node(path)
```

Arguments:

`path` The path to the node to retrieve. Must start with a forward-slash "/".

Returns: The zarr_group or zarr_array instance located at `path`, or `NULL` if the `path` was not found.

`zarr$add_group()`: Add a group below a given path.

Usage:

```
zarr$add_group(path, name)
```

Arguments:

`path` The path to the parent group of the new group, a single character string.

`name` The name for the new group, a single character string.

Returns: The newly created zarr_group, or `NULL` if the group could not be created.

`zarr$add_array()`**:** Add an array to the store.

Usage:

```
zarr$add_array(path, name, metadata)
```

Arguments:

`path` Optional. The path to the group of the new array, a single character string. If omitted, a single-array store is assumed and the store must be uninitialised (meaning that it cannot already hold any groups or arrays).

`name` The name for the new array, a single character string.

`metadata` A `list` with the metadata for the new array, or a valid array_builder instance.

Returns: The newly created zarr_array, or `NULL` if the array could not be created.

`zarr$delete_group()`**:** Delete a group from the Zarr object. This will also delete the group from the Zarr store. The root group cannot be deleted but it can be specified through `path = "/"` in which case the root group loses any specific group metadata (with only the basic parameters remaining), as well as any arrays and sub-groups if `recursive = TRUE`. **Warning:** this operation is irreversible for many stores!

Usage:

```
zarr$delete_group(path, recursive = FALSE)
```

Arguments:

`path` The path to the group.

`recursive` Logical, default `FALSE`. If `FALSE`, the operation will fail if the group has any arrays or sub-groups. If `TRUE`, the group and all Zarr objects contained by it will be deleted.

Returns: Self, invisible.

`zarr$delete_array()`**:** Delete an array from the Zarr object. If the array is the root of the Zarr object, the Zarr object will become uninitialised. **Warning:** this operation is irreversible for many stores!

Usage:

```
zarr$delete_array(path)
```

Arguments:

`path` The path to the array.

Returns: Self, invisible.

`zarr$save_to()`**:** This method can be used to save a Zarr store to a different location. The entire store is persisted to the new location.

Note that saving a Zarr store on a local file system to another location on the same local file system is usually much faster with disk management tools of the operating system (e.g. copy/paste).

The new Zarr store is always written as Zarr version 3. This method can thus be used to quickly convert from Zarr version 2 to version 3 stores.

Usage:

```
zarr$save_to(location, verbose = FALSE)
```

Arguments:

`location` Character string that indicates a location where to save this Zarr object to. Any writable location may be used, such as a directory on a local file system or a bucket on a S3 server. The character string may contain UTF-8 characters and/or use a file URI format. The Zarr specification recommends that the location use the ".zarr" extension to identify the location as a Zarr store.

`verbose` Should feedback be provided? Default is `FALSE`. When `TRUE` and in interactive mode and the `cli` package is installed then that package will be used, otherwise standard text messages will be printed to the console.

Returns: Self, invisibly.

`zarr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
zarr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`zarr_array` *Zarr Array*

Description

This class implements a Zarr array. A Zarr array is stored in a node in the hierarchy of a Zarr data set. The array contains the data for an object.

Super classes

`zarr_object` -> `zarr_node` -> `zarr_array`

Active bindings

`data_type` (read-only) Retrieve the data type of the array.

`shape` (read-only) Retrieve the shape of the array, an integer vector.

`chunking` (read-only) The chunking engine for this array.

`chunk_separator` (read-only) Retrieve the separator to be used for creating store keys for chunks.

`codecs` The list of codecs that this array uses for encoding data (and decoding in inverse order).

`raw_read` Set or retrieve the logical flag that indicates if data read from a Zarr store should have cells with the `fill_value` of the array be converted to `NA` (`FALSE`) or kept as-is (`TRUE`). The value applies to the array for all subsequent reads.

Methods

Public methods:

- zarr_array$new()
- zarr_array$print()
- zarr_array$details()
- zarr_array$hierarchy_nodes()
- zarr_array$read()
- zarr_array$write()
- zarr_array$flush()
- zarr_array$copy_to()
- zarr_array$resize()
- zarr_array$promote()

`zarr_array$new()`: Initialize a new array in a Zarr hierarchy. The array will be created in the store if it does not yet exist, opened otherwise.

Usage:

```
zarr_array$new(name, metadata, parent, no_create_check = FALSE)
```

Arguments:

`name` The name of the array. Ignored for a single-array store.

`metadata` List with the metadata of the array, or an instance of class array_builder.

`parent` The parent zarr_group instance of this new array, or the zarr object for a single-array store.

`no_create_check` Optional, logical flag to indicate if a check for existence of the group in the store should be made. Default is `FALSE`. Set to `TRUE` only when existence has been established before calling this method.

Returns: An instance of `zarr_array`.

`zarr_array$print()`: Print a summary of the array to the console.

Usage:

```
zarr_array$print()
```

`zarr_array$details()`: Produce a named character vector with a summary of the array including its name, data type, shape (with dimension names if available) and the chunking.

Usage:

```
zarr_array$details()
```

Returns: Named character vector with details of the array.

`zarr_array$hierarchy_nodes()`: Prints this array to a hierarchy. Usually called from the Zarr object or a group to display the full group hierarchy.

Usage:

```
zarr_array$hierarchy_nodes(idx, total)
```

Arguments:

`idx, total` Arguments to control indentation. Ignored if package `cli` is installed.

`zarr_array$read()`**:** Read some or all of the array data for the array. For all types other than logical, any data elements with the `fill_value` of the Zarr data type are set to `NA` if the `raw_read` field is `TRUE`.

Usage:

```
zarr_array$read(selection)
```

Arguments:

`selection` A list as long as the array has dimensions where each element is a range of indices along the dimension to read. If missing or `NULL`, the entire array will be read.

Returns: A vector, matrix or array of data.

`zarr_array$write()`**:** Write data for the array. The data will be chunked, encoded and persisted in the store that the array is using. Prior to writing, any `NA` values are assigned the `fill_value` of the array. Note that the logical type cannot encode `NA` in Zarr and any `NA` values are set to `FALSE`.

Usage:

```
zarr_array$write(data, selection, flush = TRUE)
```

Arguments:

`data` An R vector, matrix or array with the data to write. The data in the R object has to agree with the data type and rank of the array.

`selection` Optional. A `list` as long as the array has dimensions where each element is a range of indices along the dimension to write. If missing, the `data` object must have the same size as the array. Ignored when the array is scalar.

`flush` Logical, default is `TRUE`. Should the chunks that have been written to be flushed to the store (`TRUE`), or should they be left stale for further writes to the same chunk (`FALSE`). Leaving the chunks stale for further writing leads to better performance when chunks are written to multiple times. Call the `flush()` method to persist data in stale chunks.

Details: An error is thrown when the selection extends beyond the shape of the array, including when `selection` is missing and `data` is larger than the array. Use `resize()` first to grow the array.

Returns: Self, invisibly.

`zarr_array$flush()`**:** Persist the data in all chunks with pending edits to the store.

Usage:

```
zarr_array$flush()
```

`zarr_array$copy_to()`**:** Copy this array to another store. The chunks of this array are copied as stored and without decoding, so the copy is byte-identical. For a sharded array, whole shards are copied.

Usage:

```
zarr_array$copy_to(dest, verbose)
```

Arguments:

`dest` The `zarr` or `zarr_group` instance to copy this array into.

`verbose` A `list` with details for providing feedback; empty if no feedback is requested.

Details: This is a method for internal use. It is called by the zarr_group `copy_to()` method, which itself is called by the zarr `save_to()` method.
An error is raised if this array uses storage transformers.

Returns: The object in argument `dest` with this array copied into it.

`zarr_array$resize()`**:** Resize the array, growing or shrinking any combination of dimensions at either end in one pass. Existing chunk payload is never rewritten, except for a chunk left straddling a shrinking, non-chunk- aligned high-end boundary (its excess elements become `NA`).

Because the chunk grid is fixed, `low` can only move in whole chunks: values are rounded outward to the nearest chunk (more space added when growing, less removed when shrinking), so the array's origin may land ahead of where the actual data starts; those cells read as `NA`. `high` is not constrained this way.

Usage:

```
zarr_array$resize(low, high)
```

Arguments:

`low, high` Integer vectors, one element per array dimension. Positive grows that end, negative shrinks it, `0` (default) leaves it unchanged.

Returns: Self, invisibly.

`zarr_array$promote()`**:** Insert a new dimension into the array, increasing its rank by one. This operation doesn't grow an existing dimension, it creates one where there wasn't one before — the typical case being promoting a 0-d scalar array to rank 1, or giving an existing array a new leading dimension (e.g. turning a `(lat, lon)` array into a `(time, lat, lon)` array once a second file/time step becomes available).

Existing chunk payload is moved, never re-encoded: inserting a size-`length` dimension whose chunk size equals `length` never changes the relative order of elements in the encoded byte stream, for any array rank or transpose order, because a size-1-chunk dimension never contributes more than a single (vacuous) index to the enumeration. So every existing chunk is just renamed with a "0" grid index inserted at `dimension`. The new dimension therefore always starts out as exactly one full chunk; grow it afterwards with `resize()`.

Usage:

```
zarr_array$promote(dimension, length = 1L)
```

Arguments:

`dimension` Integer. 1-based position of the new dimension in the resulting shape; `1` prepends it, `rank + 1` appends it.

`length` The size of the new dimension, and also its chunk size. Default `1L`.

Returns: Self, invisibly.

zarr_codec *Zarr codecs*

Description

Zarr codecs encode data from the user data to stored data, using one or more transformations, such as compression. Decoding of stored data is the inverse process, whereby the codecs are applied in reverse order.

Super class

zarr_extension -> zarr_codec

Active bindings

`mode` (read-only) Retrieve the operating mode of the encoding operation of the codec in form of a string "array -> array", "array -> bytes" or "bytes -> bytes".

`from` (read-only) Character string that indicates the source data type of this codec, either "array" or "bytes".

`to` (read-only) Character string that indicates the output data type of this codec, either "array" or "bytes".

`configuration` (read-only) A list with the configuration parameters of the codec, exactly like they are defined in Zarr. This field is read-only but each codec class has fields to set individual parameters.

Methods

Public methods:

- zarr_codec$new()
- zarr_codec$copy()
- zarr_codec$print()
- zarr_codec$metadata_fragment()
- zarr_codec$encode()
- zarr_codec$decode()

`zarr_codec$new()`: Create a new codec object.

Usage:

```
zarr_codec$new(name, configuration)
```

Arguments:

`name` The name of the codec, a single character string.

`configuration` A list with the configuration parameters for this codec.

Returns: An instance of this class.

`zarr_codec$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec$copy()
```

Returns: This method always throws an error.

`zarr_codec$print()`: Print a summary of the codec to the console.

Usage:

```
zarr_codec$print()
```

`zarr_codec$metadata_fragment()`: Return the metadata fragment that describes this codec.

Usage:

```
zarr_codec$metadata_fragment()
```

Returns: A list with the metadata of this codec.

`zarr_codec$encode()`: This method encodes a data object but since this is the base codec class the "encoding" is a no-op.

Usage:

```
zarr_codec$encode(data)
```

Arguments:

`data` The data to be encoded.

Returns: The encoded data object, unaltered.

`zarr_codec$decode()`: This method decodes a data object but since this is the base codec class the "decoding" is a no-op.

Usage:

```
zarr_codec$decode(data)
```

Arguments:

`data` The data to be decoded.

Returns: The decoded data object, unaltered.

`zarr_codec_blosc` *Zarr blosc codec*

Description

The Zarr "blosc" codec offers a number of compression options to reduce the size of a raw vector prior to storing, and decompressing when reading.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_blosc`

Active bindings

`cname` Set or retrieve the name of the compression algorithm. Must be one of "blosclz", "lz4", "lz4hc", "zstd" or "zlib".

`clevel` Set or retrieve the compression level. Must be an integer between 0 (no compression) and 9 (maximum compression).

`shuffle` Set or retrieve the data shuffling of the compression algorithm. Must be one of "shuffle", "noshuffle" or "bitshuffle".

`typesize` Set or retrieve the size in bytes of the data type being compressed. It is highly recommended to leave this at the automatically determined value.

`blocksize` Set or retrieve the size in bytes of the blocks being compressed. It is highly recommended to leave this at a value of 0 such that the blosc library will automatically determine the optimal value.

Methods

Public methods:

- `zarr_codec_blosc$new()`
- `zarr_codec_blosc$copy()`
- `zarr_codec_blosc$encode()`
- `zarr_codec_blosc$decode()`

`zarr_codec_blosc$new()`: Create a new "blosc" codec object. The typesize argument is taken from the data type of the array passed in through the `data_type` argument and the shuffle argument is chosen based on the `data_type`.

Usage:

```
zarr_codec_blosc$new(data_type, configuration = NULL)
```

Arguments:

`data_type` The zarr_data_type instance of the Zarr array that this codec is used for.

`configuration` Optional. A list with the configuration parameters for this codec. If not given, the default compression of "zstd" with level 1 will be used.

Returns: An instance of this class.

`zarr_codec_blosc$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec_blosc$copy()
```

Returns: An instance of `zarr_codec_blosc`.

`zarr_codec_blosc$encode()`: This method compresses a data object using the "blosc" compression library.

Usage:

```
zarr_codec_blosc$encode(data)
```

Arguments:

`data` The raw vector to be compressed.

Returns: A raw vector with compressed data.

`zarr_codec_blosc$decode()`: This method decompresses a data object using the "blosc" compression library.

Usage:

```
zarr_codec_blosc$decode(data)
```

Arguments:

`data` The raw vector to be decoded.

Returns: A raw vector with the decoded data.

`zarr_codec_bytes` *Zarr bytes codec*

Description

The Zarr "bytes" codec encodes an R data object to a raw byte string, and decodes a raw byte string to a R object, possibly inverting the endianness of the data in the operation.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_bytes`

Active bindings

`endian` Set or retrieve the endianness of the storage of the data with this codec. A string with value of "big" or "little".

Methods

Public methods:

- `zarr_codec_bytes$new()`
- `zarr_codec_bytes$copy()`
- `zarr_codec_bytes$metadata_fragment()`
- `zarr_codec_bytes$encode()`
- `zarr_codec_bytes$decode()`

`zarr_codec_bytes$new()`: Create a new "bytes" codec object.

Usage:

```
zarr_codec_bytes$new(data_type, chunk_shape, configuration = NULL)
```

Arguments:

`data_type` The zarr_data_type instance of the Zarr array that this codec is used for.

`chunk_shape` The shape of a chunk of data of the array, an integer vector.

`configuration` Optional. A list with the configuration parameters for this codec. The element `endian` specifies the byte ordering of the data type of the Zarr array. A string with value "big" or "little". If not given, the default endianness of the platform is used.

Returns: An instance of this class.

`zarr_codec_bytes$copy()`**:** Create a new, independent copy of this codec.

Usage:

```
zarr_codec_bytes$copy()
```

Returns: An instance of `zarr_codec_bytes`.

`zarr_codec_bytes$metadata_fragment()`**:** Return the metadata fragment that describes this codec.

Usage:

```
zarr_codec_bytes$metadata_fragment()
```

Returns: A list with the metadata of this codec.

`zarr_codec_bytes$encode()`**:** This method writes an R object to a raw vector in the data type of the Zarr array.

Usage:

```
zarr_codec_bytes$encode(data)
```

Arguments:

`data` The data to be encoded.

Returns: A raw vector with the encoded data object.

`zarr_codec_bytes$decode()`**:** This method takes a raw vector and converts it to an R object of an appropriate type.

Usage:

```
zarr_codec_bytes$decode(data)
```

Arguments:

`data` The data to be decoded.

Returns: An R object with the shape of a chunk from the array.

`zarr_codec_crc32c` *Zarr CRC32C codec*

Description

The Zarr "CRC32C" codec computes a 32-bit checksum of a raw vector. Upon encoding the codec appends the checksum to the end of the vector. When decoding, the final 4 bytes from the raw vector are extracted and compared to the checksum of the remainder of the raw vector - if the two don't match a warning is generated.

Super classes

zarr_extension -> zarr_codec -> zarr_codec_crc32c

Methods

Public methods:

- zarr_codec_crc32c$new()
- zarr_codec_crc32c$copy()
- zarr_codec_crc32c$encode()
- zarr_codec_crc32c$decode()

zarr_codec_crc32c$new(): Create a new "crc32c" codec object.

Usage:

```
zarr_codec_crc32c$new()
```

Returns: An instance of this class.

zarr_codec_crc32c$copy(): Create a new, independent copy of this codec.

Usage:

```
zarr_codec_crc32c$copy()
```

Returns: An instance of zarr_codec_crc32c.

zarr_codec_crc32c$encode(): This method computes the CRC32C checksum of a data object and appends it to the data object.

Usage:

```
zarr_codec_crc32c$encode(data)
```

Arguments:

data A raw vector whose checksum to compute.

Returns: The input data raw vector with the 32-bit checksum appended to it.

zarr_codec_crc32c$decode(): This method extracts the CRC32C checksum from the trailing 32-bits of a data object. It then computes the CRC32C checksum from the data object (less the trailing 32-bits) and compares the two values. If the values differ, a warning will be issued.

Usage:

```
zarr_codec_crc32c$decode(data)
```

Arguments:

data The raw vector whose checksum to verify.

Returns: The data raw vector with the trailing 32-bits removed.

zarr_codec_gzip *Zarr gzip codec*

Description

The Zarr "gzip" codec compresses a raw vector prior to storing, and uncompresses the raw vector when reading.

Super classes

zarr_extension -> zarr_codec -> zarr_codec_gzip

Active bindings

`level` The compression level of the gzip codec, an integer value between 0L (no compression) and 9 (maximum compression).

Methods

Public methods:

- zarr_codec_gzip$new()
- zarr_codec_gzip$copy()
- zarr_codec_gzip$encode()
- zarr_codec_gzip$decode()

`zarr_codec_gzip$new()`: Create a new "gzip" codec object.

Usage:

```
zarr_codec_gzip$new(configuration = NULL)
```

Arguments:

`configuration` Optional. A list with the configuration parameters for this codec. The element `level` specifies the compression level of this codec, ranging from 0 (no compression) to 9 (maximum compression).

Returns: An instance of this class.

`zarr_codec_gzip$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec_gzip$copy()
```

Returns: An instance of `zarr_codec_gzip`.

`zarr_codec_gzip$encode()`: This method encodes a data object.

Usage:

```
zarr_codec_gzip$encode(data)
```

Arguments:

`data` The data to be encoded.

Returns: The encoded data object.

`zarr_codec_gzip$decode()`: This method decodes a data object.

Usage:

```
zarr_codec_gzip$decode(data)
```

Arguments:

`data` The data to be decoded.

Returns: The decoded data object.

`zarr_codec_sharding` *Zarr sharding codec*

Description

The Zarr sharding codec is not a true codec in the sense that it does not encode or decode - that is left up to regular codec defined inside this "codec" configuration. This implementation can read from a store using sharding, writing is not supported.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_sharding`

Methods

Public methods:

- `zarr_codec_sharding$new()`
- `zarr_codec_sharding$copy()`

`zarr_codec_sharding$new()`: Create a new "crc32c" codec object.

Usage:

```
zarr_codec_sharding$new(configuration)
```

Arguments:

`configuration` Optional. A list with the configuration parameters for this codec but since this codec doesn't have any the argument is always ignored.

Returns: An instance of this class.

`zarr_codec_sharding$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec_sharding$copy()
```

Returns: An instance of `zarr_codec_sharding`.

`zarr_codec_transpose` *Zarr transpose codec*

Description

The Zarr "transpose" codec registers the storage order of a data object relative to the canonical row-major ordering of Zarr. If the registered ordering is different from the native ordering on the platform where the array is being read, the data object will be permuted upon reading.

R data is arranged in column-major order. The most efficient storage arrangement between Zarr and R is thus column-major ordering, avoiding encoding to the canonical row-major ordering during storage and decoding to column-major ordering during a read. If the storage arrangement is not row-major ordering, a transpose codec must be added to the array definition. Note that within R, both writing and reading are no-ops when data is stored in column-major ordering. On the other hand, when no transpose codec is defined for the array, there will be an automatic transpose of the data on writing and reading to maintain compatibility with the Zarr specification. Using the array_builder will automatically add the transpose codec to the array definition.

For maximum portability (e.g. with Zarr implementations outside of R that do not implement the transpose codec), data should be stored in row-major order, which can be achieved by not including this codec in the array definition.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_transpose`

Active bindings

`order` Set or retrieve the 0-based ordering of the dimensions of the array when storing

Methods

Public methods:

- `zarr_codec_transpose$new()`
- `zarr_codec_transpose$copy()`
- `zarr_codec_transpose$encode()`
- `zarr_codec_transpose$decode()`

`zarr_codec_transpose$new()`: Create a new "transpose" codec object.

Usage:

```
zarr_codec_transpose$new(shape_length, configuration = list())
```

Arguments:

`shape_length` The length of the shape of the array that this codec operates on.

`configuration` Optional. A list with the configuration parameters for this codec. The element `order` specifies the ordering of the dimensions of the shape relative to the Zarr canonical arrangement. An integer vector with a length equal to argument `shape_length`. The ordering must be 0-based. If not given, the default R ordering is used.

Returns: An instance of this class.

`zarr_codec_transpose$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec_transpose$copy()
```

Returns: An instance of `zarr_codec_transpose`.

`zarr_codec_transpose$encode()`: This method permutes a data object to match the desired dimension ordering.

Usage:

```
zarr_codec_transpose$encode(data)
```

Arguments:

`data` The data to be permuted, an R matrix or array.

Returns: The permuted data object, a matrix or array in Zarr store dimension order.

`zarr_codec_transpose$decode()`: This method permutes a data object from a Zarr store to an R matrix or array.

Usage:

```
zarr_codec_transpose$decode(data)
```

Arguments:

`data` The data to be permuted, from a Zarr store.

Returns: The permuted data object, an R matrix or array.

`zarr_codec_ucs4` *Numpy UCS-4 codec*

Description

The Numpy UCS-4 format is a fixed-length character string format where shorter string are padded on the right with 0's. This is not a Zarr codec but specific to Numpy. It is included here because many Zarr v.2 stores have been written with this formatting of character strings. Since it does not use a codec in Zarr v.2, it is an invalid configuration in Zarr v.3 and this package because it embodies the `array -> bytes` step that is mandatory in Zarr v.3 - this is why this mock codec is included. This "codec" encodes an R character object to a raw byte string, and decodes a raw byte string to a R character object. The character object, typically a vector but possibly a matrix or array, should use UTF-8 encoding, which is the standard on modern platforms running R.

This codec is not part of the Zarr v.3 core specification but a commonly used process in Zarr v.2 on Python to serialize character strings into Zarr chunks. This implementation enables the use of the Zarr v.2 `"<U*"` data type. As a consequence, this codec can only decode data - new data is not written in this format.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_ucs4`

Active bindings

`endian` (read-only) Retrieve the endianness of the storage of the data with this codec. A string with value of "big" or "little".

Methods

Public methods:

- `zarr_codec_ucs4$new()`
- `zarr_codec_ucs4$copy()`
- `zarr_codec_ucs4$metadata_fragment()`
- `zarr_codec_ucs4$decode()`

`zarr_codec_ucs4$new()`: Create a new UCS-4 codec object.

Usage:

```
zarr_codec_ucs4$new(chunk_shape, configuration)
```

Arguments:

`chunk_shape` The shape of a chunk of data of the array, an integer vector.

`configuration` A list with the configuration parameters for this codec. This is a list created in this package, it does not exist in the Zarr store as the Numpy UCS-4 method is not a real codec. The element `endian` specifies the byte ordering of the data type of the Zarr array. A string with value "big" or "little". The element `width` given the fixed padded string width.

Returns: An instance of this class.

`zarr_codec_ucs4$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec_ucs4$copy()
```

Returns: An instance of `zarr_codec_ucs4`.

`zarr_codec_ucs4$metadata_fragment()`: Return the metadata fragment that describes this codec.

Usage:

```
zarr_codec_ucs4$metadata_fragment()
```

Returns: A list with the metadata of this codec, just the name.

`zarr_codec_ucs4$decode()`: This method takes a raw UCS-4 vector and converts it to an R character object.

Usage:

```
zarr_codec_ucs4$decode(data)
```

Arguments:

`data` The data to be decoded.

Returns: An R character object with the shape of a chunk from the array.

zarr_codec_vlenutf8 *Zarr vlen-utf8 codec*

Description

The Zarr "vlen-utf8" codec encodes an R character object to a raw byte string, and decodes a raw byte string to a R character object. The character object, typically a vector but possibly a matrix or array, should use UTF-8 encoding, which is the standard on modern platforms running R.

This codec is not part of the Zarr v.3 core specification but a commonly used codec to serialize character strings into Zarr chunks. It is defined for Zarr v.2. This implementation enables the use of the Zarr v.3 registered `"string"` data type, as well as Zarr v.2 `"|O"` data type.

The codec does not handle `NA` values. On encoding, `NA` values become empty strings (`""`); on decoding empty strings are preserved (not set to `NA`). This behaviour is adopted from Python, making it the most interoperable arrangement. If support for `NA` values is needed at the application level, use should be made of a sentinel character string (like `"NO_DATA"`) which then gets set to `NA` in the application. This will obviously not be interoperable, at least not outside of the application ecosystem.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_vlenutf8`

Methods

Public methods:

- `zarr_codec_vlenutf8$new()`
- `zarr_codec_vlenutf8$copy()`
- `zarr_codec_vlenutf8$metadata_fragment()`
- `zarr_codec_vlenutf8$encode()`
- `zarr_codec_vlenutf8$decode()`

`zarr_codec_vlenutf8$new()`**:** Create a new "vlen-utf8" codec object.

Usage:

```
zarr_codec_vlenutf8$new()
```

Returns: An instance of this class.

`zarr_codec_vlenutf8$copy()`**:** Create a new, independent copy of this codec.

Usage:

```
zarr_codec_vlenutf8$copy()
```

Returns: An instance of `zarr_codec_vlenutf8`.

`zarr_codec_vlenutf8$metadata_fragment()`**:** Return the metadata fragment that describes this codec.

Usage:

`zarr_codec_vlenutf8$metadata_fragment()`

Returns: A list with the metadata of this codec, just the name.

`zarr_codec_vlenutf8$encode()`: This method writes an R character object to a raw vector. Prior to writing, any `NA` values are converted to an empty string.

Usage:

```
zarr_codec_vlenutf8$encode(data)
```

Arguments:

`data` The data to be encoded.

Returns: A raw vector with the encoded data object.

`zarr_codec_vlenutf8$decode()`: This method takes a raw vector and converts it to an R character object.

Usage:

```
zarr_codec_vlenutf8$decode(data)
```

Arguments:

`data` The data to be decoded.

Returns: An R character object with the shape of a chunk from the array.

`zarr_codec_zstd` *Zarr "zstd" codec*

Description

This class provides the codec for "zstd" compression.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_codec_zstd`

Active bindings

`level` The compression level of the zstd codec, an integer value between 1L (fast) and 20 (maximum compression).

Methods

Public methods:

- `zarr_codec_zstd$new()`
- `zarr_codec_zstd$copy()`
- `zarr_codec_zstd$encode()`
- `zarr_codec_zstd$decode()`

`zarr_codec_zstd$new()`: Create a new "zstd" codec object.

Usage:

```
zarr_codec_zstd$new(configuration = NULL)
```

Arguments:

`configuration` Optional. A list with the configuration parameters for this codec. The element `level` specifies the compression level of this codec, ranging from 1 (no compression) to 20 (maximum compression).

Returns: An instance of this class.

`zarr_codec_zstd$copy()`: Create a new, independent copy of this codec.

Usage:

```
zarr_codec_zstd$copy()
```

Returns: An instance of `zarr_codec_zstd`.

`zarr_codec_zstd$encode()`: This method encodes a raw data object.

Usage:

```
zarr_codec_zstd$encode(data)
```

Arguments:

`data` The raw data to be encoded.

Returns: The encoded raw data object.

`zarr_codec_zstd$decode()`: This method decodes a raw data object.

Usage:

```
zarr_codec_zstd$decode(data)
```

Arguments:

`data` The raw data to be decoded.

Returns: The decoded raw data object.

`zarr_convention` *Zarr convention*

Description

This class implements a basic Zarr convention attribute factory. A convention is a set of attributes specific to a certain domain of application. These attributes are included in Zarr group and array attributes and are interpreted by application code. Conventions may be grouped in domains and combined with other conventions.

Application-specific conventions need to inherit from this base class and redefine relevant methods. Descendant conventions may have additional methods specific to the domain of the data. Descendants of this class take the elements that define it and then return them formatted for inclusion in the attributes of a Zarr node.

It is not useful to directly instantiate this class, use a descendant convention instead. It is recommended that descendant classes use the "zarr_convention_*" naming pattern and that they are included in a R package with similar conventions and/or domain(s).

Active bindings

`name` (read-only) The name of the convention, possibly with a trailing semi-colon ":".

`schema` (read-only) The URL to the schema of the convention.

`uuid` (read-only) The UUID of the convention, in string format.

`spec` The URL to the specification of the convention.

`description` A short description of the convention.

Methods

Public methods:

- `zarr_convention$new()`
- `zarr_convention$register()`
- `zarr_convention$set()`
- `zarr_convention$as_list()`
- `zarr_convention$clear()`

`zarr_convention$new()`: Create a new instance of a Zarr convention agent. This is a "virtual" ancestor class that should not be instantiated directly - instead use one of the descendant classes.

Usage:

```
zarr_convention$new(name, schema, uuid)
```

Arguments:

`name` String value with the name of the convention.

`schema` String value with the URL to the schema of the convention.

`uuid` String value with the UUID of the convention.

Returns: A new instance of a Zarr convention agent.

`zarr_convention$register()`: Register the use of a convention in the attributes of a Zarr object.

Usage:

```
zarr_convention$register(attributes, brief = FALSE)
```

Arguments:

`attributes` A `list` with Zarr attributes for a group or array.

`brief` Logical flag to indicate if the registration should only include the name and the schema URL or all details (default).

Returns: The updated attributes.

`zarr_convention$set()`: Set the attributes for this convention for use in a Zarr node. This is a stub that descendant classes can implement, using a specific set of arguments. More complex conventions can use other arrangements to set the more complex attributes.

Usage:

```
zarr_convention$set()
```

`zarr_convention$as_list()`: Format the elements of a convention instance in a list suitable for the attributes of a Zarr object. Descendant classes should implement their specific solutions.

Usage:

```
zarr_convention$as_list()
```

Returns: The convention attributes in a `list`.

`zarr_convention$clear()`: Clear any attributes that may have been set. Only the properties of the convention itself will remain in place.

Usage:

```
zarr_convention$clear()
```

`zarr_conventions` *List the Zarr conventions supported by this release*

Description

List the Zarr conventions supported by this release

Usage

```
zarr_conventions()
```

Value

A `data.frame` with descriptions of supported conventions.

`zarr_convention_ref` *Convention "ref"*

Description

This class implements the "ref" convention. This convention provides a standard way of referring to objects from a referring group or array in a Zarr store. The referenced object may be located in the same Zarr store or in an external Zarr store. In particular, the following convention is implemented here:

```
{
 "schema_url": "https://raw.githubusercontent.com/R-CF/zarr_convention_ref/refs/tags/v2.0.0/schema.j
 "spec_url": "https://github.com/R-CF/zarr_convention_ref/blob/v2.0.0/README.md",
 "uuid": "d89b30cf-ed8c-43d5-9a16-b492f0cd8786",
 "name": "ref",
 "description": "External Reference Convention"
}
```

Super class

zarr_convention -> zarr_convention_ref

Methods

Public methods:

- zarr_convention_ref$new()
- zarr_convention_ref$set()
- zarr_convention_ref$clear()
- zarr_convention_ref$as_list()

zarr_convention_ref$new(): Create a new instance of a "ref" convention agent.

Usage:

```
zarr_convention_ref$new()
```

Returns: A new instance of a "ref" convention agent.

zarr_convention_ref$set(): Set the attributes for this convention for use in a Zarr node. Any attributes set previously are replaced.

Usage:

```
zarr_convention_ref$set(node, uri, attribute)
```

Arguments:

node Character string. Path to the Zarr node containing the data of interest. The path is relative to the referring node when argument uri is missing, absolute from the root of the Zarr store otherwise.

uri Optional, character string. URI of an external Zarr store. Omit for nodes that are in the same local store as the referring node.

attribute Optional, a character string with a JSON pointer to a referenced attribute in the metadata of the referenced node.

Details: The node path must be relative (not starting with "/") when uri is not given, and absolute (starting with "/") when it is. An error is thrown when this is not the case, when node or uri is not a non-empty character string, or when attribute is not a valid JSON pointer.

Returns: Self, invisibly.

zarr_convention_ref$clear(): Clear any attributes that may have been set. Only the properties of the convention itself will remain in place.

Usage:

```
zarr_convention_ref$clear()
```

zarr_convention_ref$as_list(): Return the data of this instance for inclusion in the attributes of a Zarr object.

Usage:

```
zarr_convention_ref$as_list()
```

Returns: A list with Zarr attributes for a group or array.

zarr_convention_uom *Convention "uom"*

Description

This class implements the "uom" convention. This convention provides a standard way of describing the unit-of-measure of Zarr array data or an attribute. In particular, the following convention is implemented here:

```
{
 "schema_url": "https://raw.githubusercontent.com/clbarnes/zarr-convention-uom/refs/tags/v1/schema.
 "spec_url": "https://github.com/clbarnes/zarr-convention-uom/blob/v1/README.md",
  "uuid": "3bbe438d-df37-49fe-8e2b-739296d46dfb",
  "name": "uom",
  "description": "Units of measurement for Zarr arrays"
}
```

Super class

zarr_convention -> zarr_convention_uom

Methods

Public methods:

- zarr_convention_uom$new()
- zarr_convention_uom$set()
- zarr_convention_uom$clear()
- zarr_convention_uom$as_list()

zarr_convention_uom$new(): Create a new instance of a "uom" convention agent.

Usage:

```
zarr_convention_uom$new()
```

Returns: A new instance of a "uom" convention agent.

zarr_convention_uom$set(): Set the attributes for this convention for use in a Zarr node.

Usage:

```
zarr_convention_uom$set(unit, version, description)
```

Arguments:

`unit` Character string. The "unit" attribute under "ucum", giving the unit-of-measure in UCUM notation.

`version` Optional, character string. The "version" attribute under "ucum", indicating the UCUM version that is being used.

`description` Optional, a character string with the "description" attribute, giving a free-text description of the unit-of-measure.

`zarr_convention_uom$clear()`: Reset any attributes that may have been set to their default values. Only the properties of the convention itself will remain in place.

Usage:

```
zarr_convention_uom$clear()
```

`zarr_convention_uom$as_list()`: Return the data of this instance for inclusion in the attributes of a Zarr object.

Usage:

```
zarr_convention_uom$as_list()
```

Returns: A `list` with Zarr attributes for a group or array.

`zarr_data_type` *Zarr data types*

Description

This class implements a Zarr data type as an extension point. This class also manages the 'fill_value' attribute associated with the data type.

Super class

`zarr_extension` -> `zarr_data_type`

Active bindings

`data_type` The data type for the Zarr array, a single character string. Setting the data type will also set the fill value to its default value.

`Rtype` (read-only) The R data type corresponding to the Zarr data type.

`signed` (read-only) Flag that indicates if the Zarr data type is signed or not.

`size` (read-only) The size of the data type, in bytes.

`fill_value` The fill value for the Zarr array, a single value that agrees with the range of the `data_type`.

Methods

Public methods:

- `zarr_data_type$new()`
- `zarr_data_type$print()`
- `zarr_data_type$metadata_fragment()`

`zarr_data_type$new()`: Create a new data type object.

Usage:

```
zarr_data_type$new(data_type, fill_value = NULL)
```

Arguments:

`data_type` The name of the data type, a single character string.

`fill_value` Optionally, the fill value for the data type.

Returns: An instance of this class.

`zarr_data_type$print()`**:** Print a summary of the data type to the console.

Usage:

```
zarr_data_type$print()
```

`zarr_data_type$metadata_fragment()`**:** Return the metadata fragment for this data type and its fill value.

Usage:

```
zarr_data_type$metadata_fragment()
```

Returns: A list with the metadata fragment.

`zarr_domain` *Zarr domain*

Description

This class implements a basic domain object for Zarr stores. Domains are specific encodings for a certain domain of application. These encodings are included in the attributes of the Zarr groups and arrays and need custom code (usually in the form of another package) to interpret the attributes and properly process array data. Domains can be fully application-specific or they can implement one or more published Zarr conventions.

Domains have to be registered with this package to become available to the processing pipeline. Domain code is then called automatically when a Zarr store is opened for all groups and arrays in the store.

New domains need to inherit from this base class and implement all of the relevant methods. New domains may have additional methods specific to the domain of the data. It is generally not useful to directly instantiate this class: for a Zarr store without a registered domain an instance of this class is initialized automatically.

Active bindings

`name` (read-only) The name of the domain.

`can_read` (read-only) Flag to indicate if this domain can read a Zarr store using this domain. Should always be `TRUE`.

`can_write` (read-only) Flag to indicate if this domain can write a Zarr store using this domain. `TRUE` for a generic Zarr store; other domains may report 'FALSE.

Methods

Public methods:

- zarr_domain$new()
- zarr_domain$build()

`zarr_domain$new()`: Create a new instance of a Zarr domain. This method should not be called directly (as in `zarr_domain$new()`); instead, descendant classes will call this method in their initialization code.

Usage:

```
zarr_domain$new(name)
```

Arguments:

`name` Character string giving the name of the domain. This may be any sensible string value. The name of the domain must be unique in the session or an error will be thrown upon registering the domain.

Returns: A new instance of a domain class.

`zarr_domain$build()`: This method will be called when the domain is requested to asses the Zarr node for domain properties. This method must be implemented by descendant classes and return an appropriate node if it will manage the node. The code should be agile and return swiftly so any non-trivial operations should be left to a later moment, for instance when the node is accessed by the application or end-user.

Usage:

```
zarr_domain$build(name, metadata, parent, store)
```

Arguments:

`name` The name of the node.

`metadata` List with the metadata of the node.

`parent` The parent node of this new node. May be `NULL` for a root node.

`store` The store to persist data in.

Returns: A `zarr_node` descendant, typically an instance of a domain-specific descendant class of `zarr_array` or `zarr_group`. If the domain does not want to manage the node, return `FALSE`.

`zarr_domains` *List the Zarr domains registered in this session*

Description

List the Zarr domains registered in this session

Usage

```
zarr_domains()
```

Value

A list with instances of `zarr_domain` descendant classes.

zarr_extension *Zarr extension support*

Description

Many aspects of a Zarr array are implemented as extensions. More precisely, all core properties of a Zarr array except for its shape are defined as extension points, down to its data type. This class is the basic ancestor for extensions. It supports generation of the appropriate metadata for the extension, as well as processing in more specialized descendant classes.

Extensions can be nested. For instance, a sharding object contains one or more codecs, with both the sharding object and the codec being extension points.

Active bindings

`name` The name of the extension. Setting the name may be restricted by descendant classes.

Methods

Public methods:

- `zarr_extension$new()`
- `zarr_extension$metadata_fragment()`

`zarr_extension$new()`**:** Create a new extension object.

Usage:

```
zarr_extension$new(name)
```

Arguments:

`name` The name of the extension, a single character string.

Returns: An instance of this class.

`zarr_extension$metadata_fragment()`**:** Return the metadata fragment that describes this extension point object. This includes the metadata of any nested extension objects.

Usage:

```
zarr_extension$metadata_fragment()
```

Returns: A list with the metadata of this extension point object.

zarr_filter *Zarr v.2 filters*

Description

Zarr v.2 filters are transformations of chunk data that were not specified by Zarr v.2 itself but by the Numcodecs package that zarr-python uses. When a Zarr v.2 store is opened, the filters become codecs in the Zarr v.3 codec pipeline of an array, with a name of `numcodecs.<id>`. Because Zarr v.2 stores are read-only in this package, filters only decode data.

This class is the ancestor of all filters. It also implements the lossy "quantize" and "bitround" filters: these reduce the precision of floating point data upon encoding but they leave the data type unchanged, so decoding is the identity.

The following filters are supported: "delta", "fixedscaleoffset", "quantize", "bitround", "astype", "categorize", "packbits", "shuffle", "crc32", "adler32" and "fletcher32". The "vlen-utf8" filter is supported through zarr_codec_vlenutf8.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_filter`

Active bindings

`encoded_type` (read-only) The zarr_data_type of the encoded data, or `NULL` if the filter does not change the data type. This is the data type that the "bytes" codec must deserialize.

Methods

Public methods:

- `zarr_filter$new()`
- `zarr_filter$copy()`
- `zarr_filter$encode()`

`zarr_filter$new()`: Create a new filter object.

Usage:

```
zarr_filter$new(id, configuration, from = "array", to = "array")
```

Arguments:

`id` The Numcodecs identifier of the filter, such as "delta".

`configuration` A list with the configuration parameters of the filter, as found in the Zarr v.2 metadata but without the "id" element.

`from, to` The input and output data object for the encoding operation, either "array" or "bytes".

Returns: An instance of this class.

`zarr_filter$copy()`: Create a new, independent copy of this filter.

Usage:

`zarr_filter$copy()`

Returns: An instance of the same class as this filter.

`zarr_filter$encode()`: Filters can only decode data: Zarr v.2 stores are read-only.

Usage:

```
zarr_filter$encode(data)
```

Arguments:

`data` Ignored.

Returns: This method always throws an error.

`zarr_filter_astype` *Zarr v.2 "astype" filter*

Description

The "astype" filter converts data to another data type. Decoding converts the data back to the decoded data type, truncating towards zero when converting floating point values to an integer type.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_filter` -> `zarr_filter_astype`

Methods

Public methods:

- `zarr_filter_astype$new()`
- `zarr_filter_astype$decode()`

`zarr_filter_astype$new()`: Create a new "astype" filter.

Usage:

```
zarr_filter_astype$new(configuration)
```

Arguments:

`configuration` A list with elements `encode_dtype` and `decode_dtype`.

Returns: An instance of this class.

`zarr_filter_astype$decode()`: Decode the data.

Usage:

```
zarr_filter_astype$decode(data)
```

Arguments:

`data` The encoded data.

Returns: The decoded data.

zarr_filter_categorize *Zarr v.2 "categorize" filter*

Description

The "categorize" filter encodes character strings as the 1-based index into a vector of labels. Strings that are not a label are encoded as 0, which decodes to an empty string, like with Numcodecs.

Super classes

zarr_extension -> zarr_codec -> zarr_filter -> zarr_filter_categorize

Methods

Public methods:

- zarr_filter_categorize$new()
- zarr_filter_categorize$decode()

zarr_filter_categorize$new(): Create a new "categorize" filter.

Usage:

```
zarr_filter_categorize$new(configuration)
```

Arguments:

configuration A list with elements labels, dtype and, optionally, astype.

Returns: An instance of this class.

zarr_filter_categorize$decode(): Decode the data.

Usage:

```
zarr_filter_categorize$decode(data)
```

Arguments:

data The encoded data.

Returns: A character vector with the decoded data.

`zarr_filter_checksum` *Zarr v.2 checksum filters*

Description

The "crc32", "adler32" and "fletcher32" filters add a 32-bit checksum to the data, in little-endian byte order. The "crc32" and "adler32" checksums are placed at the start of the data unless the configuration has a `location` of "end"; the "fletcher32" checksum is always placed at the end. Decoding strips the checksum and verifies it against the data: if the two don't match a warning is generated.

The "fletcher32" checksum is the HDF5 variant. The two 16-bit sums are verified modulo 65535 because the HDF5 algorithm does not reduce a sum of 65535 to 0.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_filter` -> `zarr_filter_checksum`

Methods

Public methods:

- `zarr_filter_checksum$new()`
- `zarr_filter_checksum$decode()`

`zarr_filter_checksum$new()`: Create a new checksum filter.

Usage:

```
zarr_filter_checksum$new(id, configuration = list())
```

Arguments:

`id` The Numcodecs identifier of the filter, one of "crc32", "adler32" or "fletcher32".

`configuration` A list with the configuration parameters of the filter, optionally with element `location` for "crc32" and "adler32".

Returns: An instance of this class.

`zarr_filter_checksum$decode()`: Strip the checksum from the data and verify it.

Usage:

```
zarr_filter_checksum$decode(data)
```

Arguments:

`data` The encoded data, a raw vector.

Returns: A raw vector with the decoded data.

zarr_filter_delta *Zarr v.2 "delta" filter*

Description

The "delta" filter encodes data as the difference between adjacent elements, with the first element stored as is. Decoding is a cumulative sum in the decoded data type, wrapping around on overflow for integer data types up to 32 bits, like Numpy does. 64-bit integers do not wrap around: an overflow yields NA.

Super classes

zarr_extension -> zarr_codec -> zarr_filter -> zarr_filter_delta

Methods

Public methods:

- zarr_filter_delta$new()
- zarr_filter_delta$decode()

zarr_filter_delta$new(): Create a new "delta" filter.

Usage:

```
zarr_filter_delta$new(configuration)
```

Arguments:

configuration A list with elements dtype and, optionally, astype.

Returns: An instance of this class.

zarr_filter_delta$decode(): Decode the data.

Usage:

```
zarr_filter_delta$decode(data)
```

Arguments:

data The encoded data.

Returns: The decoded data.

zarr_filter_fixedscaleoffset

Zarr v.2 "fixedscaleoffset" filter

Description

The "fixedscaleoffset" filter encodes data as `round((x - offset) * scale)`, typically in a smaller integer data type. Decoding computes `x / scale + offset`, truncated towards zero when the decoded data type is an integer type, like Numpy does.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_filter` -> `zarr_filter_fixedscaleoffset`

Methods

Public methods:

- `zarr_filter_fixedscaleoffset$new()`
- `zarr_filter_fixedscaleoffset$decode()`

`zarr_filter_fixedscaleoffset$new()`: Create a new "fixedscaleoffset" filter.

Usage:

```
zarr_filter_fixedscaleoffset$new(configuration)
```

Arguments:

`configuration` A list with elements `scale`, `offset`, `dtype` and, optionally, `astype`.

Returns: An instance of this class.

`zarr_filter_fixedscaleoffset$decode()`: Decode the data.

Usage:

```
zarr_filter_fixedscaleoffset$decode(data)
```

Arguments:

`data` The encoded data.

Returns: The decoded data.

`zarr_filter_packbits` *Zarr v.2 "packbits" filter*

Description

The "packbits" filter packs logical values into bits, most significant bit first. The first byte of the encoded data gives the number of bits that pad the final byte.

Super classes

`zarr_extension` -> `zarr_codec` -> `zarr_filter` -> `zarr_filter_packbits`

Methods

Public methods:

- `zarr_filter_packbits$new()`
- `zarr_filter_packbits$decode()`

`zarr_filter_packbits$new()`**:** Create a new "packbits" filter.

Usage:

```
zarr_filter_packbits$new(configuration = list())
```

Arguments:

`configuration` Ignored, the filter has no configuration parameters.

Returns: An instance of this class.

`zarr_filter_packbits$decode()`**:** Decode the data.

Usage:

```
zarr_filter_packbits$decode(data)
```

Arguments:

`data` The encoded data, an integer vector of byte values.

Returns: A logical vector with the decoded data.

zarr_filter_shuffle *Zarr v.2 "shuffle" filter*

Description

The "shuffle" filter rearranges the bytes of the data such that the first bytes of all elements come first, then all second bytes, and so on. This improves compression of the data.

Super classes

zarr_extension -> zarr_codec -> zarr_filter -> zarr_filter_shuffle

Methods

Public methods:

- zarr_filter_shuffle$new()
- zarr_filter_shuffle$decode()

zarr_filter_shuffle$new(): Create a new "shuffle" filter.

Usage:

```
zarr_filter_shuffle$new(configuration)
```

Arguments:

configuration A list with element elementsize, the size of the data elements in bytes, by default 4.

Returns: An instance of this class.

zarr_filter_shuffle$decode(): Decode the data.

Usage:

```
zarr_filter_shuffle$decode(data)
```

Arguments:

data The encoded data, a raw vector.

Returns: A raw vector with the decoded data.

zarr_group *Zarr Group*

Description

This class implements a Zarr group. A Zarr group is a node in the hierarchy of a Zarr object. A group is a container for other groups and arrays.

A Zarr group is identified by a JSON file having required metadata, specifically the attribute `"node_type": "group"`.

Super classes

`zarr_object` -> `zarr_node` -> `zarr_group`

Active bindings

`children` (read-only) The children of the group. This is a list of `zarr_group` and `zarr_array` instances, or the empty list if the group has no children.

`groups` (read-only) Retrieve the paths to the sub-groups of the hierarchy starting from the current group, as a character vector.

`arrays` (read-only) Retrieve the paths to the arrays of the hierarchy starting from the current group, as a character vector.

Methods

Public methods:

- `zarr_group$new()`
- `zarr_group$post_open()`
- `zarr_group$print()`
- `zarr_group$hierarchy_nodes()`
- `zarr_group$hierarchy()`
- `zarr_group$build_hierarchy()`
- `zarr_group$get_node()`
- `zarr_group$set_node()`
- `zarr_group$count_arrays()`
- `zarr_group$add_group()`
- `zarr_group$add_array()`
- `zarr_group$copy_to()`
- `zarr_group$delete()`
- `zarr_group$delete_all()`

`zarr_group$new()`**:** Open a group in a Zarr hierarchy. The group must already exist in the store.

Usage:

```
zarr_group$new(name, metadata, parent, no_create_check = FALSE)
```

Arguments:

`name` The name of the group. For a root group, this is the empty string `""`.

`metadata` Optional. List with the metadata of the group. If omitted it will default to a simple Zarr v.3 group.

`parent` The parent `zarr_group` instance of this new group, can be a `zarr` instance for the root group.

`no_create_check` Optional, logical flag to indicate if a check for existence of the group in the store should be made. Default is `FALSE`. Set to `TRUE` only when existence has been established before calling this method.

Returns: An instance of `zarr_group`.

`zarr_group$post_open()`**:** This method is called automatically after a Zarr store is opened to allow for operations after the full hierarchy has been established. All contained children will be called similarly.

Usage:

```
zarr_group$post_open()
```

Returns: Self, invisibly.

`zarr_group$print()`**:** Print a summary of the group to the console.

Usage:

```
zarr_group$print()
```

`zarr_group$hierarchy_nodes()`**:** Collects the hierarchy of the group and its subgroups and arrays in a character vector. Usually called from the Zarr object or a group to display the full group hierarchy.

Usage:

```
zarr_group$hierarchy_nodes(idx = 1L, total = 1L)
```

Arguments:

`idx, total` Optional arguments to control indentation. Should both be 1 (the default) when called interactively. The values will be updated during recursion when there are groups below the current group. Ignored if package `cli` is installed.

`zarr_group$hierarchy()`**:** Print the Zarr hierarchy from the current group to the console.

Usage:

```
zarr_group$hierarchy()
```

`zarr_group$build_hierarchy()`**:** Return the hierarchy contained in the store as a tree of group and array nodes. This method only has to be called after opening an existing Zarr store - this is done automatically by user-facing code. After that, users can access the `children` property of this class.

Usage:

```
zarr_group$build_hierarchy()
```

Returns: This zarr_group instance with all of its children linked.

`zarr_group$get_node()`: Retrieve the group or array represented by the node located at the path relative from the current group.

Usage:

```
zarr_group$get_node(path)
```

Arguments:

`path` The path to the node to retrieve. The path is relative to the group, it must not start with a slash "/". The path may start with any number of double dots ".." separated by slashes "/" to denote groups higher up in the hierarchy.

Returns: The zarr_group or zarr_array instance located at `path`, or `NULL` if the `path` was not found.

`zarr_group$set_node()`: Set a group or array in the current group. CAUTION: The node must have been persisted to the store for a reliable functioning. All that this method does is update the in-memory representation of the Zarr hierarchy.

Usage:

```
zarr_group$set_node(node)
```

Arguments:

`node` The group or array to insert in this group. CAUTION: If a node with an identical name already exists in this group it will be replaced by the object in this argument.

Returns: The `node` object.

`zarr_group$count_arrays()`: Count the number of arrays in this group, optionally including arrays in sub-groups.

Usage:

```
zarr_group$count_arrays(recursive = TRUE)
```

Arguments:

`recursive` Logical flag that indicates if arrays in sub-groups should be included in the count. Default is `TRUE`.

`zarr_group$add_group()`: Add a group to the Zarr hierarchy under the current group.

Usage:

```
zarr_group$add_group(name)
```

Arguments:

`name` The name of the new group.

Returns: The newly created `zarr_group` instance.

`zarr_group$add_array()`: Add an array to the Zarr hierarchy in the current group.

Usage:

```
zarr_group$add_array(name, metadata)
```

Arguments:

`name` The name of the new array.

`metadata` A `list` with the metadata for the new array, or an instance of class array_builder whose data make a valid array definition.

Returns: The newly created `zarr_array` instance, or `NULL` if the array could not be created.

`zarr_group$copy_to()`: Copy this group to another group. Child objects, sub-groups and arrays, are recursively visited to be copied too.

Usage:

```
zarr_group$copy_to(dest, verbose)
```

Arguments:

`dest` A `zarr` or `zarr_group` instance to copy this group into.

`verbose` A `list` with details for providing feedback; empty if no feedback is requested.

Details: This is a method for internal use. It is called by the zarr `save_to()` method.

Returns: The object in argument `dest` with this group and descendant objects copied into it.

`zarr_group$delete()`: Delete a group or an array contained by this group. When deleting a group it cannot contain other groups or arrays. **Warning:** this operation is irreversible for many stores!

Usage:

```
zarr_group$delete(name)
```

Arguments:

`name` The name of the group or array to delete. This will also accept a path to a group or array but the group or array must be a node directly under this group.

Returns: Self, invisibly.

`zarr_group$delete_all()`: Delete all the groups and arrays contained by this group, including any sub-groups and arrays. Any specific metadata attached to this group is deleted as well - only a basic metadata document is maintained. **Warning:** this operation is irreversible for many stores!

Usage:

```
zarr_group$delete_all()
```

Returns: Self, invisibly.

`zarr_httpstore` *Zarr Store for HTTP access*

Description

This class implements a Zarr HTTP store. With this class Zarr stores on web servers can be read. For Zarr v.2 HTTP stores there exists a standard for publishing arrays on the store, using consolidated metadata. This class will look for such metadata in the root of the store. If no consolidated metadata is found then a regular group or array is searched for. Note that if a group is found that there is no standard process to determine what arrays are available in the store and where they are located relative to the root. Typically such information is found in the attributes of the group and you are advised to inspect those attributes and refer to the documentation of the store publisher.

This class performs no sanity checks on any of the arguments passed to the methods, for performance reasons. Since this class should be accessed through group and array objects, it is up to that code to ensure that arguments are valid, in particular keys and prefixes.

Super class

zarr_store -> zarr_httpstore

Active bindings

friendlyClassName (read-only) Name of the class for printing.

root (read-only) The root of the HTTP store, identical to its URL.

uri (read-only) The URI of the store location.

separator (read-only) The default chunk separator of the store, usually a slash '/'.

Methods

Public methods:

- zarr_httpstore$new()
- zarr_httpstore$brief()
- zarr_httpstore$exists()
- zarr_httpstore$clear()
- zarr_httpstore$erase()
- zarr_httpstore$erase_prefix()
- zarr_httpstore$list_dir()
- zarr_httpstore$list_prefix()
- zarr_httpstore$set()
- zarr_httpstore$set_if_not_exists()
- zarr_httpstore$get()
- zarr_httpstore$get_many()
- zarr_httpstore$get_metadata()
- zarr_httpstore$set_metadata()
- zarr_httpstore$is_group()
- zarr_httpstore$create_group()
- zarr_httpstore$create_array()

zarr_httpstore$new(): Create an instance of this class.
HTTP stores are read-only. Currently two types of Zarr store can be accessed. A Zarr v.2 consolidated metadata file at the root of the store (immediately below the URL) can identify a hierarchy of groups and arrays. Alternatively, a store with a group or a single array, either v.2 or v.3.

Usage:

```
zarr_httpstore$new(url)
```

Arguments:

url The path to the HTTP store to be opened. The URL may use UTF-8 code points.

Returns: An instance of this class.

zarr_httpstore$brief(): Print a brief summary of the HTTP store to the console. This will use cli if available.

Usage:

```
zarr_httpstore$brief()
```

`zarr_httpstore$exists()`: Check if a key exists in the store. The key can point to a group, an array, or a metadata file. This check is only relevant for HTTP stores with consolidated metadata. In other cases the single group or array will be at the root.

Usage:

```
zarr_httpstore$exists(key)
```

Arguments:

`key` Character string. The key that the store will be searched for.

Returns: `TRUE` if argument `key` is found, `FALSE` otherwise.

`zarr_httpstore$clear()`: Clearing the store is not supported.

Usage:

```
zarr_httpstore$clear()
```

Returns: `FALSE`.

`zarr_httpstore$erase()`: Removing a key from the store is not supported.

Usage:

```
zarr_httpstore$erase(key)
```

Arguments:

`key` Ignored.

Returns: `FALSE`.

`zarr_httpstore$erase_prefix()`: Removing keys from the store is not supported.

Usage:

```
zarr_httpstore$erase_prefix(prefix)
```

Arguments:

`prefix` Ignored.

Returns: `FALSE`.

`zarr_httpstore$list_dir()`: Retrieve all keys and prefixes with a given prefix and which do not contain the character "/" after the given prefix. In other words, this retrieves all the nodes in the store below the node indicated by the prefix.

Usage:

```
zarr_httpstore$list_dir(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character array with all keys found in the store immediately below the `prefix`, both for groups and arrays.

`zarr_httpstore$list_prefix()`: Retrieve all keys and prefixes with a given prefix.

Usage:

```
zarr_httpstore$list_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character vector with all paths found in the store below the `prefix` location, both for groups and arrays.

`zarr_httpstore$set()`: Storing a `(key, value)` pair is not supported.

Usage:

```
zarr_httpstore$set(key, value)
```

Arguments:

`key` Ignored.

`value` Ignored.

Returns: Self, invisibly.

`zarr_httpstore$set_if_not_exists()`: Storing a `(key, value)` pair is not supported.

Usage:

```
zarr_httpstore$set_if_not_exists(key, value)
```

Arguments:

`key` Ignored.

`value` Ignored.

Returns: Self, invisibly.

`zarr_httpstore$get()`: Retrieve the value associated with a given key.

Usage:

```
zarr_httpstore$get(key, prototype = NULL, byte_range = NULL)
```

Arguments:

`key` Character string. The key for which to get data.

`prototype` Ignored. The only buffer type that is supported maps directly to an R raw vector.

`byte_range` If `NULL`, all data associated with the key is retrieved. If a single positive integer, all bytes starting from a given byte offset to the end of the object are returned. If a single negative integer, the final bytes are returned. If an integer vector of length 2, request a specific range of bytes where the end is exclusive. If the range ends after the end of the object, the entire remainder of the object will be returned. If the given range is zero-length or starts after the end of the object, an error will be returned.

Returns: A raw vector with the data pointed at by the key.

`zarr_httpstore$get_many()`: Retrieve the values of several keys concurrently.

Usage:

```
zarr_httpstore$get_many(keys)
```

Arguments:

`keys` Character vector of keys.

Returns: A list as long as `keys` with a raw vector for each key, or `NULL` for a key that is not present in the store.

`zarr_httpstore$get_metadata()`: Retrieve the metadata document of the node at the location indicated by the `prefix` argument. The metadata will always be presented to the caller in the Zarr v.3 format. Attributes, if present, will be added.

Usage:

```
zarr_httpstore$get_metadata(prefix)
```

Arguments:

`prefix` The prefix of the node whose metadata document to retrieve.

Returns: A list with the metadata, or `NULL` if the prefix is not pointing to a Zarr group or array.

`zarr_httpstore$set_metadata()`: Setting metadata is not supported.

Usage:

```
zarr_httpstore$set_metadata(prefix, metadata)
```

Arguments:

`prefix` Ignored.

`metadata` Ignored.

Returns: Self, invisible

`zarr_httpstore$is_group()`: Test if `path` is pointing to a Zarr group.

Usage:

```
zarr_httpstore$is_group(path)
```

Arguments:

`path` The path to test.

Returns: `TRUE` if the `path` points to a Zarr group, `FALSE` otherwise.

`zarr_httpstore$create_group()`: Creating a new group in the store is not supported.

Usage:

```
zarr_httpstore$create_group(path, name)
```

Arguments:

`path, name` Ignored.

Returns: An error indicating that the group could not be created.

`zarr_httpstore$create_array()`: Creating a new array in the store is not supported.

Usage:

```
zarr_httpstore$create_array(parent, name, metadata)
```

Arguments:

`parent, name, metadata` Ignored.

Returns: An error indicating that the array could not be created.

zarr_localstore *Zarr Store for the Local File System*

Description

This class implements a Zarr store for the local file system. With this class Zarr stores on devices accessible through the local file system can be read and written to. This includes locally attached drives, removable media, NFS mounts, etc.

The chunking pattern is to locate all the chunks of an array in a single directory. That means that chunks have names like "c0.0.0" in the array directory.

This class performs no sanity checks on any of the arguments passed to the methods, for performance reasons. Since this class should be accessed through group and array objects, it is up to that code to ensure that arguments are valid, in particular keys and prefixes.

Super class

zarr_store -> zarr_localstore

Active bindings

friendlyClassName (read-only) Name of the class for printing.

root (read-only) The root directory of the file system store.

uri (read-only) The URI of the store location.

separator (read-only) The default chunk separator of the local file store, usually a dot '.'.

supports_prefix_rename (read-only) Flag to indicate whether the store can rename an entire prefix (e.g. a directory) as a single operation, moving everything below it regardless of how many chunk keys that represents.

Methods

Public methods:

- zarr_localstore$new()
- zarr_localstore$brief()
- zarr_localstore$exists()
- zarr_localstore$clear()
- zarr_localstore$erase()
- zarr_localstore$erase_prefix()
- zarr_localstore$list_dir()
- zarr_localstore$list_prefix()
- zarr_localstore$list_chunks()
- zarr_localstore$rename()
- zarr_localstore$rename_prefix()
- zarr_localstore$set()

- zarr_localstore$set_if_not_exists()
- zarr_localstore$get()
- zarr_localstore$get_metadata()
- zarr_localstore$set_metadata()
- zarr_localstore$is_group()
- zarr_localstore$create_group()
- zarr_localstore$create_array()

`zarr_localstore$new()`: Create an instance of this class.

If the root location does not exist, it will be created. The location on the file system must be writable by the process creating the store. The store is not yet functional in the sense that it is just an empty directory. Write a root group with `.$create_group('/', '')` or an array with `.$create_array('/', '', metadata)` for a single-array store before any other operations on the store.

If the root location does exist on the file system it must be a valid Zarr store, as determined by the presence of a "zarr.json" file. It is an error to try to open a Zarr store on an existing location where this metadata file is not present.

Usage:

```
zarr_localstore$new(root, read_only = FALSE)
```

Arguments:

`root` The path to the local store to be created or opened. The path may use UTF-8 code points. Following the Zarr specification, it is recommended that the root path has an extension of ".zarr" to easily identify the location as a Zarr store. When creating a file store, the root directory cannot already exist.

`read_only` Flag to indicate if the store is opened read-only. Default `FALSE`.

Returns: An instance of this class.

`zarr_localstore$brief()`: Print a brief summary of the local file system store to the console. This will use `cli` if available.

Usage:

```
zarr_localstore$brief()
```

`zarr_localstore$exists()`: Check if a key exists in the store. The key can point to a group, an array, or a chunk.

Usage:

```
zarr_localstore$exists(key)
```

Arguments:

`key` Character string. The key that the store will be searched for.

Returns: `TRUE` if argument `key` is found, `FALSE` otherwise.

`zarr_localstore$clear()`: Clear the store. Remove all keys and values from the store. WARNING: Invoking this method deletes affected files on the file system and this action can not be undone.

Usage:

`zarr_localstore$clear()`

Returns: `TRUE` if the operation proceeded, `FALSE` otherwise.

`zarr_localstore$erase()`**:** Remove a key from the store. The key must point to an array chunk or an empty group. The location of the key and all of its values are removed.

Usage:

```
zarr_localstore$erase(key)
```

Arguments:

`key` Character string. The key to remove from the store.

Returns: `TRUE` if the operation proceeded, `FALSE` otherwise.

`zarr_localstore$erase_prefix()`**:** Remove all keys in the store that begin with a given prefix. The last location in the prefix is preserved while all keys below are removed from the store. Any metadata extensions added to the group pointed to by the prefix will be deleted as well - only a basic group-identifying metadata file will remain.

Usage:

```
zarr_localstore$erase_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix to groups or arrays to remove from the store, including in child groups.

Returns: `TRUE` if the operation proceeded, `FALSE` otherwise.

`zarr_localstore$list_dir()`**:** Retrieve all keys and prefixes with a given prefix and which do not contain the character "/" after the given prefix. In other words, this retrieves all the nodes in the store below the node indicated by the prefix.

Usage:

```
zarr_localstore$list_dir(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character array with all keys found in the store immediately below the `prefix`, both for groups and arrays.

`zarr_localstore$list_prefix()`**:** Retrieve all keys and prefixes with a given prefix.

Usage:

```
zarr_localstore$list_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character vector with all paths found in the store below the `prefix` location, both for groups and arrays.

`zarr_localstore$list_chunks()`**:** Retrieve all chunk (and shard) keys stored for the array at the given prefix. Used internally to support resizing and promoting arrays.

Usage:

```
zarr_localstore$list_chunks(prefix)
```

Arguments:

`prefix` The prefix of the array whose chunk keys to retrieve.

Returns: A character vector of full store keys under `prefix`. This may include keys that are not chunks, such as the metadata document of the array; the chunking scheme of the array selects the chunk keys.

`zarr_localstore$rename()`: Rename a single key on the local file system. A thin wrapper over `rename_prefix()` — a single chunk file is, as far as the file system is concerned, no different from a directory.

Usage:

```
zarr_localstore$rename(old_key, new_key)
```

Arguments:

`old_key, new_key` Character strings, the source and destination keys. `old_key` must exist; `new_key` is overwritten if it exists.

`zarr_localstore$rename_prefix()`: Rename a whole subtree of the local file system in one operation. Works identically whether `old_prefix` resolves to a directory (an outer chunk-grid dimension) or to a single chunk file (the innermost dimension) — both are just directory entries.

Usage:

```
zarr_localstore$rename_prefix(old_prefix, new_prefix)
```

Arguments:

`old_prefix, new_prefix` Character strings, relative to the store root.

`zarr_localstore$set()`: Store a `(key, value)` pair. The key points to a specific file (shard or chunk of an array) in a store, rather than a group or an array. The key must be relative to the root of the store (so not start with a "/") and may be composite. It must include the name of the file. An example would be "group/subgroup/array/c0.0.0". The group hierarchy and the array must have been created before. If the `value` exists, it will be overwritten.

Usage:

```
zarr_localstore$set(key, value)
```

Arguments:

`key` The key whose value to set.

`value` The value to set, a complete chunk of data, a `raw` vector.

Returns: Self, invisibly, or an error.

`zarr_localstore$set_if_not_exists()`: Store a `(key, value)` pair. The key points to a specific file (shard or chunk of an array) in a store, rather than a group or an array. The key must be relative to the root of the store (so not start with a "/") and may be composite. It must include the name of the file. An example would be "group/subgroup/array/c0.0.0". The group hierarchy and the array must have been created before. If the `key` exists, nothing will be written.

Usage:

```
zarr_localstore$set_if_not_exists(key, value)
```

Arguments:

`key` The key whose value to set.

`value` The value to set, a complete chunk of data.

Returns: Self, invisibly, or an error.

`zarr_localstore$get()`**:** Retrieve the value associated with a given key.

Usage:

```
zarr_localstore$get(key, prototype = NULL, byte_range = NULL)
```

Arguments:

`key` Character string. The key for which to get data.

`prototype` Ignored. The only buffer type that is supported maps directly to an R raw vector.

`byte_range` If `NULL`, all data associated with the key is retrieved. If a single positive integer, all bytes starting from a given byte offset to the end of the object are returned. If a single negative integer, the final bytes are returned. If an integer vector of length 2, request a specific range of bytes where the end is exclusive. If the range ends after the end of the object, the entire remainder of the object will be returned. If the given range is zero-length or starts after the end of the object, an error will be returned.

Returns: An raw vector of data, or `NULL` if no data was found.

`zarr_localstore$get_metadata()`**:** Retrieve the metadata document of the node at the location indicated by the `prefix` argument. The metadata will always be presented to the caller in the Zarr v.3 format.

Usage:

```
zarr_localstore$get_metadata(prefix)
```

Arguments:

`prefix` The prefix of the node whose metadata document to retrieve.

Returns: A list with the metadata, or `NULL` if the prefix is not pointing to a Zarr group or array.

`zarr_localstore$set_metadata()`**:** Set the metadata document of the node at the location indicated by the `prefix` argument. The formatting of the metadata should always use the Zarr v.3 format, it will be converted internally if the store is Zarr v.2. If the metadata does not include an element

Usage:

```
zarr_localstore$set_metadata(prefix, metadata)
```

Arguments:

`prefix` The prefix of the node whose metadata document to set.

`metadata` The metadata to persist, either a `list` or an instance of array_builder.

Returns: Self, invisible

`zarr_localstore$is_group()`**:** Test if `path` is pointing to a Zarr group.

Usage:

```
zarr_localstore$is_group(path)
```

Arguments:

`path` The path to test.

Returns: `TRUE` if the `path` points to a Zarr group, `FALSE` otherwise.

`zarr_localstore$create_group()`: Create a new group in the store under the specified path.

Usage:

```
zarr_localstore$create_group(parent, name)
```

Arguments:

`parent` The path to the parent group of the new group. Ignored when creating a root group.

`name` The name of the new group. This may be an empty string `""` to create a root group. It is an error to supply an empty string if a root group or array already exists.

Returns: A list with the metadata of the group, or an error if the group could not be created.

`zarr_localstore$create_array()`: Create a new array in the store under the specified path to the `parent` argument.

Usage:

```
zarr_localstore$create_array(parent, name, metadata)
```

Arguments:

`parent` The path to the parent group of the new array. Ignored when creating a root array.

`name` The name of the new array. This may be an empty string `""` to create a root array. It is an error to supply an empty string if a root group or array already exists.

`metadata` A `list` with the metadata for the array. The list has to be valid for array construction. Use the array_builder class to create and or test for validity. An element "chunk_key_encoding" will be added to the metadata if not already present or with a value other than a dot "." or a slash "/".

Returns: A list with the metadata of the array, or an error if the array could not be created.

References

https://zarr-specs.readthedocs.io/en/latest/v3/stores/filesystem/index.html

---

`zarr_memorystore` *In-memory Zarr Store*

Description

This class implements a Zarr store in RAM memory. With this class Zarr stores can be read and written to. Obviously, any data is not persisted after the memory store is de-referenced and garbage-collected.

All data is stored in a list. The Zarr array itself has a list with the metadata, its chunks have names like "c.0.0.0" and they have an R array-like value.

This class performs no sanity checks on any of the arguments passed to the methods, for performance reasons. Since this class should be accessed through group and array objects, it is up to that code to ensure that arguments are valid, in particular keys and prefixes.

Super class

zarr_store -> zarr_memorystore

Active bindings

friendlyClassName (read-only) Name of the class for printing.

separator (read-only) The separator of the memory store, always a dot '.'.

keys (read-only) The defined keys in the store.

Methods

Public methods:

- zarr_memorystore$new()
- zarr_memorystore$exists()
- zarr_memorystore$clear()
- zarr_memorystore$erase()
- zarr_memorystore$erase_prefix()
- zarr_memorystore$list_dir()
- zarr_memorystore$list_prefix()
- zarr_memorystore$list_chunks()
- zarr_memorystore$rename()
- zarr_memorystore$set()
- zarr_memorystore$set_if_not_exists()
- zarr_memorystore$get()
- zarr_memorystore$get_metadata()
- zarr_memorystore$set_metadata()
- zarr_memorystore$create_group()
- zarr_memorystore$create_array()

zarr_memorystore$new(): Create an instance of this class.

Usage:

```
zarr_memorystore$new()
```

Returns: An instance of this class.

zarr_memorystore$exists(): Check if a key exists in the store. The key can point to a group, an array (having a metadata list as its value) or a chunk.

Usage:

```
zarr_memorystore$exists(key)
```

Arguments:

key Character string. The key that the store will be searched for.

Returns: TRUE if argument key is found, FALSE otherwise.

zarr_memorystore$clear(): Clear the store. Remove all keys and values from the store. Invoking this method deletes all data and this action can not be undone.

Usage:

```
zarr_memorystore$clear()
```

Returns: `TRUE`. This operation always proceeds successfully once invoked.

`zarr_memorystore$erase()`: Remove a key from the store. The key must point to an array or a chunk. If the key points to an array, the key and all of subordinated keys are removed.

Usage:

```
zarr_memorystore$erase(key)
```

Arguments:

`key` Character string. The key to remove from the store.

Returns: `TRUE`. This operation always proceeds successfully once invoked, even if argument `key` does not point to an existing key.

`zarr_memorystore$erase_prefix()`: Remove all keys in the store that begin with a given prefix.

Usage:

```
zarr_memorystore$erase_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix to groups or arrays to remove from the store, including in child groups.

Returns: `TRUE`. This operation always proceeds successfully once invoked, even if argument `prefix` does not point to any existing keys.

`zarr_memorystore$list_dir()`: Retrieve all keys with a given prefix and which do not contain the character "/" after the given prefix. In other words, this retrieves all the keys in the store below the key indicated by the prefix.

Usage:

```
zarr_memorystore$list_dir(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character array with all keys found in the store immediately below the `prefix`.

`zarr_memorystore$list_prefix()`: Retrieve all keys and prefixes with a given prefix.

Usage:

```
zarr_memorystore$list_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix to nodes to list.

Returns: A character vector with all paths found in the store below the `prefix` location.

`zarr_memorystore$list_chunks()`: Retrieve all chunk (and shard) keys stored for the array at the given prefix. Used internally to support resizing and promoting arrays.

Usage:

```
zarr_memorystore$list_chunks(prefix)
```

Arguments:

`prefix` The prefix of the array whose chunk keys to retrieve.

Returns: A character vector of full store keys under `prefix`. This may include keys that are not chunks, such as the metadata document of the array; the chunking scheme of the array selects the chunk keys.

`zarr_memorystore$rename()`**:** Rename a key in the store, moving its value without necessarily reading or re-encoding it. Used internally to support resizing and promoting arrays. The default implementation is a plain copy, for stores without a cheaper native operation.

Usage:

```
zarr_memorystore$rename(old_key, new_key)
```

Arguments:

`old_key`, `new_key` Character strings, the source and destination keys. `old_key` must exist; `new_key` is overwritten if it exists.

Returns: Self, invisibly.

`zarr_memorystore$set()`**:** Store a `(key, value)` pair. If the `value` exists, it will be overwritten.

Usage:

```
zarr_memorystore$set(key, value)
```

Arguments:

`key` The key whose value to set.

`value` The value to set, typically a complete chunk of data, a `raw` vector.

Returns: Self, invisibly.

`zarr_memorystore$set_if_not_exists()`**:** Store a `(key, value)` pair. If the `key` exists, nothing will be written.

Usage:

```
zarr_memorystore$set_if_not_exists(key, value)
```

Arguments:

`key` The key whose value to set.

`value` The value to set, a complete chunk of data.

Returns: Self, invisibly, or an error.

`zarr_memorystore$get()`**:** Retrieve the value associated with a given key.

Usage:

```
zarr_memorystore$get(key, prototype = NULL, byte_range = NULL)
```

Arguments:

`key` Character string. The key for which to get data.

`prototype` Ignored. The only buffer type that is supported maps directly to an R raw vector.

`byte_range` If `NULL`, all data associated with the key is retrieved. If a single positive integer, all bytes starting from a given byte offset to the end of the object are returned. If a single negative integer, the final bytes are returned. If an integer vector of length 2, request a specific range of bytes where the end is exclusive. If the range ends after the end of the object, the entire remainder of the object will be returned. If the given range is zero-length or starts after the end of the object, an error will be returned.

Returns: An raw vector of data, or `NULL` if no data was found.

`zarr_memorystore$get_metadata()`**:** Retrieve the metadata document at the location indicated by the `prefix` argument.

Usage:

```
zarr_memorystore$get_metadata(prefix)
```

Arguments:

`prefix` The prefix whose metadata document to retrieve.

Returns: A list with the metadata, or `NULL` if the prefix is not pointing to a Zarr array.

`zarr_memorystore$set_metadata()`**:** Set the metadata document of the node at the location indicated by the `prefix` argument. The formatting of the metadata should always use the Zarr v.3 format.

Usage:

```
zarr_memorystore$set_metadata(prefix, metadata)
```

Arguments:

`prefix` The prefix of the node whose metadata document to set.

`metadata` The metadata to persist, either a `list` or an instance of array_builder.

Returns: Self, invisible

`zarr_memorystore$create_group()`**:** Create a new group in the store under the specified path.

Usage:

```
zarr_memorystore$create_group(parent, name)
```

Arguments:

`parent` The path to the parent group of the new group. Ignored when creating a root group.

`name` The name of the new group. This may be an empty string `""` to create a root group. It is an error to supply an empty string if a root group or array already exists.

Returns: A list with the metadata of the group, or an error if the group could not be created.

`zarr_memorystore$create_array()`**:** Create a new array in the store under key constructed from the specified path to the `parent` argument and the `name`. The key may not already exist in the store.

Usage:

```
zarr_memorystore$create_array(parent, name, metadata)
```

Arguments:

`parent` The path to the parent group of the new array. This is ignored if the `name` argument is the empty string.

`name` The name of the new array.

`metadata` A `list` with the metadata for the array. The list has to be valid for array construction. Use the array_builder class to create and or test for validity. An element "chunk_key_encoding" will be added to the metadata if it not already there or contains an invalid separator.

Returns: A list with the metadata of the array, or an error if the array could not be created.

`zarr_node` *Zarr Hierarchy node*

Description

This class implements a Zarr node. The node is an element in the hierarchy of the Zarr object. As per the Zarr specification, the node is either a group or an array. Thus, this class is the ancestor of the zarr_group and zarr_array classes. This class manages common features such as names, key, prefixes and paths, as well as the hierarchy between nodes and the zarr_store for persistent storage.

This class should never have to be instantiated or accessed directly. Instead, use instances of `zarr_group` or `zarr_array`. Function arguments are largely not checked, the group and array instances should do so prior to calling methods here. The big exception is checking the validity of node names.

Super class

`zarr_object` -> `zarr_node`

Active bindings

`zarr` (read-only) Retrieve the zarr object that manages this node.

`parent` The parent of the node. For a root node this returns the `zarr` object, otherwise this `zarr_group` or `zarr_array` instance. CAUTION: Setting the parent of a node can invalidate the Zarr hierarchy

- expert use only.

`store` (read-only) The store of the node.

`path` (read-only) The path of this node, relative to the root node of the hierarchy.

`prefix` (read-only) The prefix of this node, relative to the root node of the hierarchy.

`metadata` The metadata document of this node, a `list`. CAUTION: Setting a list that is not properly describing this object will render the object invalid.

`dirty` Flag to indicate that there are unsaved edits in the node. Such edits always concern the metadata and then in particular the user-level attributes. Setting the field to `TRUE` will ensure that the edits are saved to the Zarr store.

`attributes` (read-only) Retrieve the list of attributes of this object. Attributes can be added or modified with the `set_attribute()` and `append_array_attribute()` methods or removed with the `delete_attributes()` method.

Methods

Public methods:

- zarr_node$new()
- zarr_node$post_open()
- zarr_node$relative_path()
- zarr_node$absolute_path()
- zarr_node$walk_path()
- zarr_node$attribute()
- zarr_node$set_attribute()
- zarr_node$append_array_attribute()
- zarr_node$delete_attribute()
- zarr_node$save()

`zarr_node$new()`: Initialize a new node in a Zarr hierarchy.

Usage:

```
zarr_node$new(name, metadata, parent)
```

Arguments:

`name` The name of the node.

`metadata` List with the metadata of the node.

`parent` The parent node of this new node, either a zarr_group or a zarr instance.

Returns: A new instance of this class.

`zarr_node$post_open()`: This method is called automatically after a Zarr store is opened to allow for operations after the full hierarchy has been established. This is a no-op here, descendant classes with specific requirements should implement this method.

Usage:

```
zarr_node$post_open()
```

Returns: Self, invisibly.

`zarr_node$relative_path()`: Retrieve the relative path from the current node to the indicated node or path. In the relative path parent nodes are indicated with `..`, child nodes start with the child node name down to the target node.

Usage:

```
zarr_node$relative_path(to)
```

Arguments:

`to` Either a `zarr_node` instance or a character string giving the object to which the relative path reference is sought.

Returns: A character string with the relative path from this node. If the `to` argument is empty or points to this node '.' is returned. Note that the relative path has no leading slash (as regular Zarr paths do).

`zarr_node$absolute_path()`: Retrieve the absolute path from the indicated relative path from this node. In the relative path parent nodes are indicated with `..`, child nodes start with the child node name down to the target node.

Usage:

```
zarr_node$absolute_path(dest)
```

Arguments:

`dest` A character string giving the relative path to be made absolute calculated from the current node.

Returns: A character string with the absolute Zarr path from this node.

`zarr_node$walk_path()`: Walk from the current node to a target node indicated by the vector of relative path references. A call to this method moves one step closer to the target node, followed by a call of this method on the next node with the `node_names` vector with its first element removed. The target is thus iteratively found by walking the Zarr hierarchy. The method is called by `zarr_group$get_node()` and the `[[` indexing method of `zarr`, which are both easier to work with than this method.

Usage:

```
zarr_node$walk_path(node_names)
```

Arguments:

`node_names` A character vector of node names to walk. The first element in the vector is resolved: a ".." value moves up to the parent of the current node, all other values must resolve to a child of the current node. The `node_names` vector is easily constructed from a relative path to the target node using `strsplit(., "/", fixed = TRUE)[[1L]]`. With an absolute path the first element ("") must be removed.

Returns: The `zarr_node` that is found when the `node_names` vector is fully walked. If the node is not found, an error will be raised.

`zarr_node$attribute()`: Retrieve a specific attribute by path.

Usage:

```
zarr_node$attribute(name)
```

Arguments:

`name` The name (path) of the attribute to retrieve, using `/` as separator for nested attributes. Numeric path segments index into array attributes (1-based), e.g. `"zarr_conventions/2/name"` retrieves the `name` field of the second convention object.

Returns: The attribute value, or `NULL` if not found.

`zarr_node$set_attribute()`: Add an attribute to the metadata of the object. If an attribute `name` already exists, it will be overwritten.

Usage:

```
zarr_node$set_attribute(name, value)
```

Arguments:

`name` The name of the attribute. The name may be a compound path, relative to the "attributes" entry in the metadata, using a slash "/" as path separator. Each of the elements in the path (between slashes) must begin with a letter and be composed of letters, digits, and underscores and can be at most 255 characters long. Missing path elements will be created.

`value` The value of the attribute. This can be of any supported type, including a vector or list of values. In general, an attribute should be a character value, a numeric value, a logical value, or a short vector or list of any of these.

Returns: Self, invisibly.

`zarr_node$append_array_attribute()`: Append an attribute to an array in the metadata of the object. If an attribute `name` already exists, it will be overwritten.

Usage:

```
zarr_node$append_array_attribute(name, value, after = NULL)
```

Arguments:

`name` The name of the attribute. The name may be a compound path, relative to the "attributes" entry in the metadata, using a slash "/" as path separator. Each of the elements in the path (between slashes) must begin with a letter and be composed of letters, digits, and underscores and can be at most 255 characters long. Missing path elements will be created.

`value` The value of the attribute. This can be of any supported type, including a vector or list of values. In general, an attribute should be a character value, a numeric value, a logical value, or a short vector or list of any of these.

`after` A subscript, after which `value` is to be appended. The default is `NULL`, meaning that `value` will be placed after the existing values. Specifying `after = 0L` will place `value` before the existing values.

Returns: Self, invisibly.

`zarr_node$delete_attribute()`: Delete an attribute or array element. If the attribute is not present, this method simply returns.

Usage:

```
zarr_node$delete_attribute(name)
```

Arguments:

`name` Character. The name (path) of the attribute to delete, using `/` as separator for nested attributes, e.g. `"first/second/my_att"`. The `name` is relative to the `attributes` entry in the metadata of the node. To target an element of a JSON array attribute, append the 1-based index as the path segment, e.g. `"first/second/my_arr/2"` to delete the second element in the array, or `"first/second/my_arr/2/description"` to delete only the `description` field inside it. This nesting can be arbitrarily deep, including over multiple JSON arrays.

Returns: Self, invisibly.

`zarr_node$save()`: Persist any edits to the group or array to the store.

Usage:

```
zarr_node$save()
```

---

zarr_object *Zarr base object*

Description

This class implements the common features of objects that make up Zarr. This class is the ancestor of zarr_node and thus the zarr_group and zarr_array classes. This class manages common features such as names and attribute management. This class may also be used by other descendant classes that need a name and attributes (such as the coordinate system classes in package `geozarr`).

The main function of this class is to manage attributes of a Zarr object. Groups and arrays can have attributes, but other object descending from this class may have attributes as well. In order to accommodate the various applications, all methods dealing with attributes take a `list` object with the attributes as the first argument - the class has no private or public field to hold the attributes.

This class should never have to be instantiated or accessed directly. Instead, use instances of `zarr_group` or `zarr_array`. Function arguments are largely not checked, the group and array instances should do so prior to calling methods here. The big exception is checking the validity of node names.

Active bindings

`name` Set or retrieve the name of the Zarr object.

Methods

Public methods:

- `zarr_object$new()`
- `zarr_object$print_attributes()`
- `zarr_object$attribute()`
- `zarr_object$set_attribute()`
- `zarr_object$append_array_attribute()`
- `zarr_object$delete_attribute()`
- `zarr_object$clone()`

`zarr_object$new()`: Initialize a new Zarr object.

Usage:

```
zarr_object$new(name)
```

Arguments:

`name` The name of the node.

`zarr_object$print_attributes()`: Print attributes to the console. Usually called by the zarr_group and zarr_array `print()` methods.

Usage:

```
zarr_object$print_attributes(dirty = FALSE, ...)
```

Arguments:

`dirty` Optional. Logical flag that indicates if the attributes have unsaved edits. Default is `FALSE`.

`...` Arguments passed to embedded functions. Of particular interest is `width = .` to specify the maximum width of the columns.

Returns: A (multi-line) character string with the attributes of the Zarr object.

`zarr_object$attribute()`: Retrieve a specific attribute by path.

Usage:

```
zarr_object$attribute(atts, name)
```

Arguments:

`atts` The `list` object with the attributes.

`name` The name (path) of the attribute to retrieve, using `/` as separator for nested attributes. Numeric path segments index into array attributes (1-based), e.g. `"zarr_conventions/2/name"` retrieves the `name` field of the second convention object.

Returns: The attribute value, or `NULL` if not found.

`zarr_object$set_attribute()`: Add an attribute to the metadata of the object. If an attribute `name` already exists, it will be overwritten.

Usage:

```
zarr_object$set_attribute(atts, name, value)
```

Arguments:

`atts` The `list` object with the attributes.

`name` The name of the attribute. The name may be a compound path, relative to the "attributes" entry in the metadata, using a slash "/" as path separator. Each of the elements in the path (between slashes) must begin with a letter and be composed of letters, digits, and underscores and can be at most 255 characters long. Missing path elements will be created.

`value` The value of the attribute. This can be of any supported type, including a vector or list of values. In general, an attribute should be a character value, a numeric value, a logical value, or a short vector or list of any of these.

Returns: A `list` with the updated attributes.

`zarr_object$append_array_attribute()`: Append an attribute to an array in the metadata of the object. If an attribute `name` already exists, it will be overwritten.

Usage:

```
zarr_object$append_array_attribute(atts, name, value, after = NULL)
```

Arguments:

`atts` The `list` object with the attributes.

`name` The name of the attribute. The name may be a compound path, relative to the "attributes" entry in the metadata, using a slash "/" as path separator. Each of the elements in the path (between slashes) must begin with a letter and be composed of letters, digits, and underscores and can be at most 255 characters long. Missing path elements will be created.

`value` The value of the attribute. This can be of any supported type, including a vector or list of values. In general, an attribute should be a character value, a numeric value, a logical value, or a short vector or list of any of these.

`after` A subscript, after which `value` is to be appended. The default is `NULL`, meaning that `value` will be placed after the existing values. Specifying `after = 0L` will place `value` before the existing values.

Returns: A `list` with the updated attributes.

`zarr_object$delete_attribute()`: Delete an attribute or array element. If the attribute is not present, this method simply returns.

Usage:

```
zarr_object$delete_attribute(atts, name)
```

Arguments:

`atts` The `list` object with the attributes.

`name` Character. The name (path) of the attribute to delete, using `/` as separator for nested attributes, e.g. `"first/second/my_att"`. The `name` is relative to the `attributes` entry in the metadata of the node. To target an element of a JSON array attribute, append the 1-based index as the path segment, e.g. `"first/second/my_arr/2"` to delete the second element in the array, or `"first/second/my_arr/2/description"` to delete only the `description` field inside it. This nesting can be arbitrarily deep, including over multiple JSON arrays.

Returns: A `list` with the updated attributes.

`zarr_object$clone()`: The objects of this class are cloneable with this method.

Usage:

```
zarr_object$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`zarr_options` *Zarr package options*

Description

Use this function to read or modify package options. On modifying an option, if the new value is not valid then it is silently discarded and the original setting remains.

Usage

```
zarr_options(key, value)
```

Arguments

`key`	Character. A key whose value to retrieve or modify. If missing, all options are returned.
`value`	Optional. The new value for the option.

Value

Nothing if argument `value` is provided. The value of argument `key` if it is provided, or a `list` with all options otherwise.

Examples

```
zarr_options()
```

zarr_register_domain *Register a Zarr domain for this session*

Description

Register a Zarr domain for this session

Usage

```
zarr_register_domain(domain)
```

Arguments

domain	An instance of a class descending from zarr_domain.

Value

Nothing.

zarr_s3store *Zarr Store for AWS S3 (and S3-compatible) access*

Description

This class implements a Zarr S3 store, read/write capable for authenticated users. It uses the paws.storage package for all S3 operations, including credential resolution, so this class never handles secret keys directly unless the caller explicitly passes them through.

Like zarr_httpstore, this class will look for consolidated metadata (.zmetadata) or a zarr.json / .zarray / .zgroup document at the configured prefix to identify the store version and root node. Unlike the HTTP store, this class does not *need* consolidated metadata to enumerate nodes: S3's ListObjectsV2 API is used directly for list_dir() and list_prefix(), so the store reflects the live state of the bucket.

Credentials are resolved by paws.storage using its standard provider chain (environment variables, shared credentials file / profile, IAM role or instance/task metadata, SSO). Pass profile to select a named profile, or explicit access_key/secret_key/session_token for credentials threaded through from elsewhere. If none of those are given, the store defaults to anonymous (unsigned) requests, since paws.storage does not fall back to anonymous access on its own and the common case for a freshly-opened store is a public bucket; pass anonymous = FALSE explicitly to force normal credential resolution with none of the above supplied (e.g. to pick up IAM role / instance metadata credentials with no explicit profile or key).

NOTE: this class performs no sanity checks on any of the arguments passed to the methods, for performance reasons, consistent with zarr_httpstore. It should be accessed through group and array objects.

Super class

zarr_store -> zarr_s3store

Active bindings

friendlyClassName (read-only) Name of the class for printing.

service (read-only) The S3 connection object.

root (read-only) The bucket and prefix of the store, as s3://.

uri (read-only) The URI of the store location, identical to root.

separator (read-only) The default chunk separator of the store, usually a slash '/'.

Methods

Public methods:

- zarr_s3store$new()
- zarr_s3store$brief()
- zarr_s3store$exists()
- zarr_s3store$clear()
- zarr_s3store$erase()
- zarr_s3store$erase_prefix()
- zarr_s3store$list_dir()
- zarr_s3store$list_prefix()
- zarr_s3store$list()
- zarr_s3store$list_chunks()
- zarr_s3store$rename()
- zarr_s3store$getsize()
- zarr_s3store$getsize_prefix()
- zarr_s3store$is_empty()
- zarr_s3store$set()
- zarr_s3store$set_if_not_exists()
- zarr_s3store$get()
- zarr_s3store$get_many()
- zarr_s3store$get_metadata()
- zarr_s3store$set_metadata()
- zarr_s3store$is_group()
- zarr_s3store$create_group()
- zarr_s3store$create_array()

zarr_s3store$new(): Create an instance of this class.

Usage:

```
zarr_s3store$new(
  bucket,
  prefix = "",
  region = NULL,
  profile = NULL,
  access_key = NULL,
  secret_key = NULL,
  session_token = NULL,
  endpoint = NULL,
  path_style = NULL,
  anonymous = NULL,
  read_only = FALSE
)
```

Arguments:

`bucket` Character string. The S3 bucket name.

`prefix` Character string. The key prefix within the bucket that acts as the root of this store (e.g. `"datasets/mycube/"`). Use `""` for the bucket root. A trailing `/` is added if missing and the prefix is non-empty.

`region` Character string, AWS region, e.g. `"eu-west-1"`. If `NULL`, resolved by `paws.storage` from the environment/config.

`profile` Character string, a named profile from the shared AWS credentials file. Ignored if `access_key` is supplied.

`access_key, secret_key, session_token` Optional explicit credentials, for cases where they must be threaded through from elsewhere rather than resolved by `paws.storage` itself. Prefer `profile` or the default provider chain over these.

`endpoint` Character string. Optional custom endpoint URL, for S3-compatible backends (MinIO, Ceph RGW, EMBASSY Cloud, etc.) rather than AWS S3.

`path_style` Logical. Force path-style addressing (`endpoint/bucket/key`) instead of virtual-hosted-style (`bucket.endpoint/key`). Non-AWS S3-compatible backends generally only support path-style, so this defaults to `TRUE` whenever `endpoint` is supplied, and `FALSE` (AWS default) otherwise. Set explicitly to override.

`anonymous` Logical. If `TRUE`, make unsigned requests, for public buckets. If `FALSE`, use `paws.storage`'s normal credential resolution (explicit `profile`/`access_key`, then its default provider chain). If left `NULL` (the default), this is decided automatically: `TRUE` when neither `profile` nor `access_key` was given, `FALSE` otherwise. `paws.storage` does not fall back to unsigned requests on its own — a call against a public bucket with no credentials configured fails outright rather than trying anonymously, which is what this default is for.

`read_only` Logical. If `TRUE`, disable all write methods (`set()`, `erase()`, etc. become no-ops), regardless of the permissions of the underlying credentials. Default `FALSE`.

Returns: An instance of this class.

`zarr_s3store$brief()`**:** Print a brief summary of the S3 store to the console. This will use `cli` if available.

Usage:

```
zarr_s3store$brief()
```

`zarr_s3store$exists()`**:** Check if a key exists in the store.

Usage:

```
zarr_s3store$exists(key)
```

Arguments:

`key` Character string. The key that the store will be searched for.

Returns: `TRUE` if argument `key` is found, `FALSE` otherwise.

`zarr_s3store$clear()`: Remove all objects below this store's prefix. No-op if the store was opened with `read_only = TRUE`.

Usage:

```
zarr_s3store$clear()
```

Returns: `TRUE` on success, `FALSE` otherwise.

`zarr_s3store$erase()`: Remove a single key from the store. No-op if the store was opened with `read_only = TRUE`.

Usage:

```
zarr_s3store$erase(key)
```

Arguments:

`key` Character string. The key to remove.

Returns: `TRUE` on success, `FALSE` otherwise.

`zarr_s3store$erase_prefix()`: Remove all keys below a prefix. Batches deletes in groups of up to 1000, the S3 `DeleteObjects` limit. No-op if the store was opened with `read_only = TRUE`.

Usage:

```
zarr_s3store$erase_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix whose keys to remove.

Returns: `TRUE` on success, `FALSE` otherwise.

`zarr_s3store$list_dir()`: Retrieve the immediate child nodes below a prefix (single path component only), using S3's native `Delimiter` support.

Usage:

```
zarr_s3store$list_dir(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character vector of node names immediately below `prefix`.

`zarr_s3store$list_prefix()`: Retrieve all keys below a prefix (recursive).

Usage:

```
zarr_s3store$list_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix whose nodes to list.

Returns: A character vector with all paths found below `prefix`, prefixed with `/` for consistency with zarr_httpstore.

`zarr_s3store$list()`**:** Retrieve all keys in the store.

Usage:

```
zarr_s3store$list()
```

Returns: A character vector with all keys found in the store.

`zarr_s3store$list_chunks()`**:** Retrieve all chunk (and shard) keys stored for the array at the given prefix. Used internally to support resizing and promoting arrays.

Usage:

```
zarr_s3store$list_chunks(prefix)
```

Arguments:

`prefix` The prefix of the array whose chunk keys to retrieve.

Returns: A character vector of full store keys under `prefix`. This may include keys that are not chunks, such as the metadata document of the array; the chunking scheme of the array selects the chunk keys.

`zarr_s3store$rename()`**:** Rename a key in the store, moving its value without necessarily reading or re-encoding it. Used internally to support resizing and promoting arrays. The default implementation is a plain copy, for stores without a cheaper native operation.

Usage:

```
zarr_s3store$rename(old_key, new_key)
```

Arguments:

`old_key`, `new_key` Character strings, the source and destination keys. `old_key` must exist; `new_key` is overwritten if it exists.

Returns: Self, invisibly.

`zarr_s3store$getsize()`**:** Return the size, in bytes, of a value in the store, via a `HeadObject` request (S3-native, no need to download the object).

Usage:

```
zarr_s3store$getsize(key)
```

Arguments:

`key` Character string. The key whose length will be returned.

Returns: The size, in bytes, of the object.

`zarr_s3store$getsize_prefix()`**:** Return the total size, in bytes, of all objects found under the group indicated by `prefix`, summed from a single (paginated) `ListObjectsV2` scan rather than per-key `HeadObject` calls.

Usage:

```
zarr_s3store$getsize_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix to groups to scan.

Returns: The total size, in bytes, as a single numeric value.

`zarr_s3store$is_empty()`: Is the group empty? Uses a single `MaxKeys = 1` `ListObjectsV2` request rather than a full listing.

Usage:

```
zarr_s3store$is_empty(prefix)
```

Arguments:

`prefix` Character string. The prefix to the group to scan.

Returns: `TRUE` if the group indicated by argument `prefix` has no sub-groups or arrays, `FALSE` otherwise.

`zarr_s3store$set()`: Store a `(key, value)` pair. No-op if the store was opened with `read_only = TRUE`.

Single-request upload only (S3's ~5 GB `PutObject` limit); this is not expected to matter for typical Zarr chunk sizes. If you start writing very large chunks (or use very large shards), this needs a multipart-upload path instead — not implemented here.

Usage:

```
zarr_s3store$set(key, value)
```

Arguments:

`key` Character string. The key to write to.

`value` Raw vector. The data to store.

Returns: Self, invisibly.

`zarr_s3store$set_if_not_exists()`: Store a `(key, value)` pair only if the key does not already exist. No-op if the store was opened with `read_only = TRUE`.

Uses a conditional `PutObject` with `IfNoneMatch = "*"` for atomic create-if-absent semantics — no separate existence check, no race window. This is an AWS S3 feature added in August 2024; it requires a `paws.storage` version whose S3 client exposes `IfNoneMatch` on `put_object()`, and a backend that honors it. Most S3-compatible services (older MinIO releases, some others) reject or ignore the header — if you're targeting one of those, this will need a fallback to a check-then-put instead, which reintroduces the race.

Usage:

```
zarr_s3store$set_if_not_exists(key, value)
```

Arguments:

`key` Character string. The key to write to.

`value` Raw vector. The data to store.

Returns: Self, invisibly.

`zarr_s3store$get()`: Retrieve the value associated with a given key.

Usage:

```
zarr_s3store$get(key, prototype = NULL, byte_range = NULL)
```

Arguments:

`key` Character string. The key for which to get data.

`prototype` Ignored. The only buffer type that is supported maps directly to an R raw vector.

`byte_range` As in zarr_httpstore: `NULL` for the whole object, a single positive integer for an open-ended range from that offset, a single negative integer for the final N bytes, or a length-2 vector `c(start, end)` with an exclusive end.

Returns: A raw vector with the data pointed at by the key, or `NULL` if the key does not exist.

`zarr_s3store$get_many()`**:** Retrieve the values of several keys concurrently.

Usage:

```
zarr_s3store$get_many(keys)
```

Arguments:

`keys` Character vector of keys.

Returns: A list as long as `keys` with a raw vector for each key, or `NULL` for a key that is not present in the store.

`zarr_s3store$get_metadata()`**:** Retrieve the metadata document of the node at `prefix`, always presented in Zarr v.3 format.

Usage:

```
zarr_s3store$get_metadata(prefix)
```

Arguments:

`prefix` The prefix of the node whose metadata document to retrieve.

Returns: A list with the metadata, or `NULL` if the prefix does not point to a Zarr group or array.

`zarr_s3store$set_metadata()`**:** Write the metadata document for the node at `prefix`. No-op if the store was opened with `read_only = TRUE`.

Usage:

```
zarr_s3store$set_metadata(prefix, metadata)
```

Arguments:

`prefix` The prefix of the node whose metadata document to write.

`metadata` A list with the Zarr v.3-format metadata to write.

Returns: Self, invisibly.

`zarr_s3store$is_group()`**:** Test if `path` is pointing to a Zarr group.

Usage:

```
zarr_s3store$is_group(path)
```

Arguments:

`path` The path to test.

Returns: `TRUE` if the `path` points to a Zarr group, `FALSE` otherwise.

`zarr_s3store$create_group()`**:** Create a new group in the store under `parent`. Errors if the store was opened with `read_only = TRUE`.

Usage:

```
zarr_s3store$create_group(parent, name)
```

Arguments:

`parent` Character string. Path of the parent node.

`name` Character string. Name of the new group.

Returns: A list with the metadata of the newly-created group.

`zarr_s3store$create_array()`: Create a new array in the store under `parent`. As with zarr_httpstore, the abstract interface's `create_array(parent, name)` is extended here with an explicit `metadata` argument, since array metadata (shape, dtype, chunking, codecs) cannot be inferred and must be supplied by the caller. Errors if the store was opened with `read_only = TRUE`.

Usage:

```
zarr_s3store$create_array(parent, name, metadata)
```

Arguments:

`parent` Character string. Path of the parent node.

`name` Character string. Name of the new array.

`metadata` A list with the Zarr v.3-format array metadata.

Returns: A list with the metadata of the newly-created array, as written.

`zarr_store` *Zarr Abstract Store*

Description

This class implements a Zarr abstract store. It provides the basic plumbing for specific implementations of a Zarr store. It implements the Zarr abstract store interface, with some extensions from the Python `zarr.abc.store.Store` abstract class. Functions `set_partial_values()` and `get_partial_values()` are not implemented.

Active bindings

`friendlyClassName` (read-only) Name of the class for printing.

`read_only` (read-only) Flag to indicate if the store is read-only.

`supports_consolidated_metadata` Flag to indicate if the store can consolidate metadata.

`supports_deletes` Flag to indicate if keys and arrays can be deleted.

`supports_listing` Flag to indicate if the store can list its keys.

`supports_partial_writes` Deprecated, always `FALSE`.

`supports_writes` Flag to indicate if the store can write data.

`supports_prefix_rename` (read-only) Flag to indicate whether the store can rename an entire prefix (e.g. a directory) as a single operation, moving everything below it regardless of how many chunk keys that represents.

`version` (read-only) The Zarr version of the store.

`separator` (read-only) The default separator between elements of chunks of arrays in the store. Every store typically has a default which is used when creating arrays. The actual chunk separator being used is determined by looking at the "chunk_key_encoding" attribute of each array.

Methods

Public methods:

- zarr_store$new()
- zarr_store$print()
- zarr_store$brief()
- zarr_store$clear()
- zarr_store$erase()
- zarr_store$erase_prefix()
- zarr_store$exists()
- zarr_store$get()
- zarr_store$get_many()
- zarr_store$getsize()
- zarr_store$getsize_prefix()
- zarr_store$is_empty()
- zarr_store$list()
- zarr_store$list_dir()
- zarr_store$list_prefix()
- zarr_store$list_chunks()
- zarr_store$rename()
- zarr_store$rename_prefix()
- zarr_store$set()
- zarr_store$set_if_not_exists()
- zarr_store$get_metadata()
- zarr_store$set_metadata()
- zarr_store$create_group()
- zarr_store$create_array()

`zarr_store$new()`: Create an instance of this class. Since this class is "abstract", it should not be instantiated directly - it is intended to be called by descendant classes, exclusively.

Usage:

```
zarr_store$new(read_only = FALSE, version = 3L)
```

Arguments:

`read_only` Flag to indicate if the store is read-only. Default `FALSE`. If argument `version` is smaller than 3, the store is always opened in read-only mode.

`version` The version of the Zarr store. By default this is 3.

Returns: An instance of this class.

`zarr_store$print()`: Print a summary of the store to the console. This will use `cli` if available.

Usage:

```
zarr_store$print()
```

Returns: Self, invisibly.

`zarr_store$brief()`: Print a brief summary of the store to the console. This will use `cli` if available. Specialised store classes can override this method to provide more details.

Usage:

```
zarr_store$brief()
```

`zarr_store$clear()`: Clear the store. Remove all keys and values from the store.

Usage:

```
zarr_store$clear()
```

Returns: Self, invisibly.

`zarr_store$erase()`: Remove a key from the store. This method is part of the abstract store interface in ZEP0001.

Usage:

```
zarr_store$erase(key)
```

Arguments:

`key` Character string. The key to remove from the store.

Returns: Self, invisibly.

`zarr_store$erase_prefix()`: Remove all keys and prefixes in the store that begin with a given prefix. This method is part of the abstract store interface in ZEP0001.

Usage:

```
zarr_store$erase_prefix(prefix)
```

Arguments:

`prefix` Character string. The prefix to groups or arrays to remove from the store, including in child groups.

Returns: Self, invisibly.

`zarr_store$exists()`: Check if a key exists in the store.

Usage:

```
zarr_store$exists(key)
```

Arguments:

`key` Character string. The key that the store will be searched for.

Returns: `TRUE` if argument `key` is found, `FALSE` otherwise.

`zarr_store$get()`: Retrieve the value associated with a given key. This method is part of the abstract store interface in ZEP0001.

Usage:

```
zarr_store$get(key, prototype, byte_range)
```

Arguments:

`key` Character string. The key for which to get data.

`prototype` Ignored. The only buffer type that is supported maps directly to an R raw vector.

byte_range If NULL, all data associated with the key is retrieved. If a single positive integer, all bytes starting from a given byte offset to the end of the object are returned. If a single negative integer, the final bytes are returned. If an integer vector of length 2, request a specific range of bytes where the end is exclusive. If the range ends after the end of the object, the entire remainder of the object will be returned. If the given range is zero-length or starts after the end of the object, an error will be returned.

Returns: An raw vector of data, or NULL if no data was found.

`zarr_store$get_many()`: Retrieve the values of several keys. This implementation retrieves them one by one; stores that can retrieve values concurrently override it.

Usage:

```
zarr_store$get_many(keys)
```

Arguments:

keys Character vector of keys.

Returns: A list as long as keys with a raw vector for each key, or NULL for a key that is not present in the store.

`zarr_store$getsize()`: Return the size, in bytes, of a value in a Store.

Usage:

```
zarr_store$getsize(key)
```

Arguments:

key Character string. The key whose length will be returned.

Returns: The size, in bytes, of the object.

`zarr_store$getsize_prefix()`: Return the size, in bytes, of all objects found under the group indicated by the prefix.

Usage:

```
zarr_store$getsize_prefix(prefix)
```

Arguments:

prefix Character string. The prefix to groups to scan.

Returns: The size, in bytes, of all the objects under a group, as a single integer value.

`zarr_store$is_empty()`: Is the group empty?

Usage:

```
zarr_store$is_empty(prefix)
```

Arguments:

prefix Character string. The prefix to the group to scan.

Returns: TRUE is the group indicated by argument prefix has no sub-groups or arrays, FALSE otherwise.

`zarr_store$list()`: Retrieve all keys in the store. This method is part of the abstract store interface in ZEP0001.

Usage:

`zarr_store$list()`

Returns: A character vector with all keys found in the store, both for groups and arrays.

`zarr_store$list_dir()`**:** Retrieve all keys and prefixes with a given prefix and which do not contain the character "/" after the given prefix. This method is part of the abstract store interface in ZEP0001.

Usage:

`zarr_store$list_dir(prefix)`

Arguments:

`prefix` Character string. The prefix to groups to list.

Returns: A list with all keys found in the store immediately below the `prefix`, both for groups and arrays.

`zarr_store$list_prefix()`**:** Retrieve all keys and prefixes with a given prefix. This method is part of the abstract store interface in ZEP0001.

Usage:

`zarr_store$list_prefix(prefix)`

Arguments:

`prefix` Character string. The prefix to groups to list.

Returns: A character vector with all fully-qualified keys found in the store, both for groups and arrays.

`zarr_store$list_chunks()`**:** Retrieve all chunk (and shard) keys stored for the array at the given prefix. Used internally to support resizing and promoting arrays.

Usage:

`zarr_store$list_chunks(prefix)`

Arguments:

`prefix` The prefix of the array whose chunk keys to retrieve.

Returns: A character vector of full store keys under `prefix`. This may include keys that are not chunks, such as the metadata document of the array; the chunking scheme of the array selects the chunk keys.

`zarr_store$rename()`**:** Rename a key in the store, moving its value without necessarily reading or re-encoding it. Used internally to support resizing and promoting arrays. The default implementation is a plain copy, for stores without a cheaper native operation.

Usage:

`zarr_store$rename(old_key, new_key)`

Arguments:

`old_key`, `new_key` Character strings, the source and destination keys. `old_key` must exist; `new_key` is overwritten if it exists.

Returns: Self, invisibly.

`zarr_store$rename_prefix()`**:** Rename an entire prefix in one operation. Only meaningful for stores where `supports_prefix_rename` is `TRUE`; callers must check that first.

Usage:

```
zarr_store$rename_prefix(old_prefix, new_prefix)
```

Arguments:

`old_prefix, new_prefix` Character strings, relative to the store root.

Returns: Self, invisibly.

`zarr_store$set()`: Store a (key, value) pair.

Usage:

```
zarr_store$set(key, value)
```

Arguments:

`key` The key whose value to set.

`value` The value to set, typically a chunk of data.

Returns: Self, invisibly.

`zarr_store$set_if_not_exists()`: Store a key to argument `value` if the key is not already present. This method is part of the abstract store interface in ZEP0001.

Usage:

```
zarr_store$set_if_not_exists(key, value)
```

Arguments:

`key` The key whose value to set.

`value` The value to set, typically an R array.

Returns: Self, invisibly.

`zarr_store$get_metadata()`: Retrieve the metadata document of the node at the location indicated by the `prefix` argument.

Usage:

```
zarr_store$get_metadata(prefix)
```

Arguments:

`prefix` The prefix of the node whose metadata document to retrieve.

`zarr_store$set_metadata()`: Set the metadata document of the node at the location indicated by the `prefix` argument. This is a no-op for stores that have no writing capability. Other stores must override this method.

Usage:

```
zarr_store$set_metadata(prefix, metadata)
```

Arguments:

`prefix` The prefix of the node whose metadata document to set.

`metadata` The metadata to persist, either a `list` or an instance of array_builder.

Returns: Self, invisible.

`zarr_store$create_group()`: Create a new group in the store under the specified path to the `parent` argument. The `parent` path must point to a Zarr group.

Usage:

```
zarr_store$create_group(parent, name)
```

Arguments:

`parent` The path to the parent group of the new group.

`name` The name of the new group.

Returns: A list with the metadata of the group, or an error if the group could not be created.

`zarr_store$create_array()`: Create a new array in the store under the specified path to the `parent` argument. The `parent` path must point to a Zarr group.

Usage:

```
zarr_store$create_array(parent, name)
```

Arguments:

`parent` The path to the parent group of the new array.

`name` The name of the new array.

Returns: A list with the metadata of the array, or an error if the array could not be created.

References

https://zarr-specs.readthedocs.io/en/latest/v3/core/index.html#abstract-store-interface

---

`zarr_unregister_domain` *Unregister a Zarr domain from this session*

Description

Unregister a Zarr domain from this session

Usage

```
zarr_unregister_domain(domain)
```

Arguments

`domain`	The name of the `zarr_domain` to unregister.

Value

Nothing.

`[.zarr_array` *Extract or replace parts of a Zarr array*

Description

These operators can be used to extract or replace data from an array by indices. Normal R array selection rules apply. The only limitation is that the indices have to be consecutive.

Usage

```
## S3 method for class 'zarr_array'
x[i, j, ..., drop = TRUE]

## S3 replacement method for class 'zarr_array'
x[...] <- value
```

Arguments

`x`	A `zarr_array` object of which to extract or replace the data.
`i, j, ...`	Indices specifying elements to extract or replace. Indices are numeric, empty (missing) or `NULL`. Numeric values are coerced to integer. The number of indices has to agree with the dimensionality of the array.
`drop`	If `TRUE` (the default), degenerate dimensions are dropped, if `FALSE` they are retained in the result.
`value`	The value to assign to the Zarr array. This must have the same shape as the slab being implied by the indices, with standard R recycling rules applying.

Value

When extracting data, a vector, matrix or array, having dimensions as specified in the indices. When replacing data, returns `x` invisibly.

Examples

```
x <- array(1:100, c(10, 10))
z <- as_zarr(x)
arr <- z[["/"]]
arr[3:5, 7:9]
arr[3, 7:9] <- 1L
arr[3:5, 7:9]
```

[[.zarr — *Get a group or array from a Zarr object*

Description

This method can be used to retrieve a group or array from the Zarr object by its path.

Usage

```
## S3 method for class 'zarr'
x[[i]]
```

Arguments

x	A zarr object to extract a group or array from.
i	The path to a group or array in x.

Value

An instance of zarr_group or zarr_array, or NULL if the path is not found.

Examples

```
z <- create_zarr()
z[["/"]]
```

[[.zarr_group — *Get a group or array from a Zarr group*

Description

This method can be used to retrieve a group or array from the Zarr group by a relative path to the desired group or array.

Usage

```
## S3 method for class 'zarr_group'
x[[i]]
```

Arguments

x	A zarr_group object to extract a group or array from.
i	The path to a group or array in x. The path is relative to the group, it must not start with a slash "/". The path may start with any number of double dots ".." separated by slashes "/" to denote groups higher up in the hierarchy.

Value

An instance of `zarr_group` or `zarr_array`, or `NULL` if the path is not found.

Examples

```
z <- create_zarr()
tst <- z$add_group("/", "tst")
z$add_group("/tst", "subtst")
tst[["subtst"]]
```

Index

[.zarr_array, 95
[<-.zarr_array *([.zarr_array)*, 95
[[,zarr-group-method *([[.zarr_group)*, 96
[[,zarr-method *([[.zarr)*, 96
[[.zarr, 96
[[.zarr_group, 96

array_builder, 3, *11*, *21*, *23*, *34*, *58*, *68, 69*, *73, 74*, *93*
as_zarr, 6

chunk_grid_regular, *4*, 7, *9*
chunk_grid_sharded, 9
chunking, *7*, *9*
create_zarr, 11

define_array, 11

is_valid_node_name, 12

open_zarr, 13
open_zarr(), *15*
optimal_chunking, 14

reset_options, 14

s3_list_dir, 15
save_options, 16
str.chunk_grid_regular, 16
str.chunk_grid_sharded, 17
str.zarr, 17
str.zarr_array, 18
str.zarr_group, 18

zarr, *11*, *13*, 19, *23*, *25*, *59*, *74, 75*
zarr_array, *19–21*, 22, *58*, *74*, *78*
zarr_codec, 26, *27*, *29*, *31–35*, *37, 38*, *48–55*
zarr_codec_blosc, 27
zarr_codec_bytes, 29
zarr_codec_crc32c, 30
zarr_codec_gzip, 32
zarr_codec_sharding, 33
zarr_codec_transpose, 34
zarr_codec_ucs4, 35
zarr_codec_vlenutf8, 37, *48*
zarr_codec_zstd, 38
zarr_convention, 39, *42, 43*
zarr_convention_ref, 41
zarr_convention_uom, 43
zarr_conventions, 41
zarr_data_type, *28, 29*, 44, *48*
zarr_domain, 45
zarr_domains, 46
zarr_extension, *7*, *9*, *26, 27*, *29*, *31–35*, *37, 38*, *44*, 47, *48–55*
zarr_filter, 48, *49–55*
zarr_filter_astype, 49
zarr_filter_categorize, 50
zarr_filter_checksum, 51
zarr_filter_delta, 52
zarr_filter_fixedscaleoffset, 53
zarr_filter_packbits, 54
zarr_filter_shuffle, 55
zarr_group, *6*, *19–21*, *23*, *25*, 56, *57, 58*, *74, 75*, *78*
zarr_httpstore, 59, *81*, *85*, *87, 88*
zarr_localstore, 64
zarr_memorystore, 69
zarr_node, *22*, *56*, 74, *78*
zarr_object, *22*, *56*, *74*, 77
zarr_options, 80
zarr_register_domain, 81
zarr_s3store, *13*, *15*, *20*, 81
zarr_store, *20*, *60*, *64*, *70*, *74*, *82*, 88
zarr_unregister_domain, 94