Package ‘zarr’

February 11, 2026

Title Native and Extensible R Driver for 'Zarr'

Version 0.2.0

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 7.3.3

Imports blosc, jsonlite, methods, R6

Suggests bit64, curl, digest, qs2, testthat (>= 3.0.0), zlib

Config/testthat/edition 3

Config/Needs/website rmarkdown

Depends R (>= 4.1)

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

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

NeedsCompilation no

Author Patrick Van Laake [aut, cre, cph]

Maintainer Patrick Van Laake <patrick@vanlaake.net>

Repository CRAN

Date/Publication 2026-02-11 12:20:02 UTC

Contents

`array-indexing` *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[..., drop = TRUE]
```

Arguments

`x`	A `zarr_array` object of which to extract or replace the data.
`...`	Indices specifying elements to extract or replace. Indices are numeric, empty (missing) or `NULL`. Numeric values are coerced to integer or whole numbers. 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

When extracting data, a vector, matrix or array, having dimensions as specified in the indices. When replacing part of the Zarr array, returns `x` invisibly.

Examples

```
x <- array(1:100, c(10, 10))
z <- as_zarr(x)
arr <- z[["/"]]
arr[3:5, 7:9]
```

`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 attributes in a Zarr array metadata 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.

`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.

`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()`

Method `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.

Method `print()`**:** Print the array metadata to the console.

Usage:

```
array_builder$print()
```

Method `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.

Method `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.

Method `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.

Method `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 = NULL, location = NULL)
```

Arguments

`x`	The R object to convert. Must be a vector, matrix or array of a numeric or logical type.
`name`	Optional. The name of the Zarr array to be created.
`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. It manages reading from and writing to Zarr stores, using the codecs for data transformation.

Super class

`zarr::zarr_extension` -> `chunk_grid_regular`

Active bindings

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

`chunk_grid` (read-only) The chunk grid of the associated array, i.e. the number of chunks in each dimension.

`chunk_encoding` Set or retrieve the chunk key encoding to be used for creating store keys for chunks.

`data_type` The data type of the array using the chunking scheme. This is set by the array when starting to use chunking for file I/O.

`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.

`store` The store of the array using the chunking scheme. This is set by the array when starting to use chunking for file I/O.

`array_prefix` The prefix of the array using the chunking scheme. This is set by the array when starting to use chunking for file I/O.

Methods

Public methods:

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

Method `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.

`chunk_shape` Integer vector of the dimensions of each chunk.

Returns: An instance of `chunk_grid_regular`.

Method `print()`**:** Print a short description of this chunking scheme to the console.

Usage:

```
chunk_grid_regular$print()
```

Returns: Self, invisibly.

Method `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 codec.

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

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.

Returns: A vector, matrix or array of data.

Method `write()`**:** Write data to the array.

Usage:

```
chunk_grid_regular$write(data, start, stop)
```

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.

Returns: Self, invisibly.

`create_zarr` *Create a Zarr store*

Description

This function creates a Zarr v.3 instance, with a store located on the local file system. 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`	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.

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
```

---

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 protocol. The Zarr object can be either v.2 or v.3.

Usage

```
open_zarr(location, read_only = FALSE)
```

Arguments

location	Character string that indicates a location on a file system or a HTTP 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 `FALSE` for a local file system store, `TRUE` otherwise.

Value

A zarr object.

Examples

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

`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.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`.

Value

A `zarr` object.

Active bindings

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

`root` (read-only) The root node of the Zarr object, usually a zarr_group instance but it could also be a zarr_array instance.

`store` (read-only) The store of the 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$clone()`

Method `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.

Usage:

```
zarr$new(store)
```

Arguments:

`store` An instance of a zarr_store descendant class where the Zarr objects are located.

Method `print()`**:** Print a summary of the Zarr object to the console.

Usage:

```
zarr$print()
```

Method `hierarchy()`**:** Print the Zarr hierarchy to the console.

Usage:

```
zarr$hierarchy()
```

Method `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.

Method `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.

Method `add_array()`**:** Add an array in a group with a given path.

Usage:

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

Arguments:

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

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

`metadata` A `list` with the metadata for the new array.

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

Method `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.

Method `delete_array()`**:** Delete an array from the Zarr object. If the array is the root of the Zarr object, it will be converted into a regular Zarr object with a root group. **Warning:** this operation is irreversible for many stores!

Usage:

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

Arguments:

`path` The path to the array.

Returns: Self, invisible.

Method `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 class

`zarr::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).

Methods

Public methods:

- `zarr_array$new()`
- `zarr_array$print()`
- `zarr_array$hierarchy()`
- `zarr_array$read()`
- `zarr_array$write()`

Method `new()`**:** Initialize a new array in a Zarr hierarchy. The array must already exist in the store

Usage:

```
zarr_array$new(name, metadata, parent, store)
```

Arguments:

`name` The name of the array.

`metadata` List with the metadata of the array.

`parent` The parent `zarr_group` instance of this new array, can be missing or `NULL` if the Zarr object should have just this array.

`store` The zarr_store instance to persist data in.

Returns: An instance of `zarr_array`.

Method `print()`**:** Print a summary of the array to the console.

Usage:

```
zarr_array$print()
```

Method `hierarchy()`**:** Prints the hierarchy of the groups and arrays to the console. Usually called from the Zarr object or its root group to display the full group hierarchy.

Usage:

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

Arguments:

`idx, total` Arguments to control indentation.

Method `read()`**:** Read some or all of the array data for the array.

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 write. If missing, the entire array will be read.

Returns: A vector, matrix or array of data.

Method `write()`**:** Write data for the array. The data will be chunked, encoded and persisted in the store that the array is using.

Usage:

```
zarr_array$write(data, selection)
```

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 of the array.

`selection` 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 entire `data` object will be written.

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::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()`

Method `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.

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

Usage:

```
zarr_codec$copy()
```

Returns: This method always throws an error.

Method `print()`**:** Print a summary of the codec to the console.

Usage:

```
zarr_codec$print()
```

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

Usage:

```
zarr_codec$metadata_fragment()
```

Returns: A list with the metadata of this codec.

Method `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.

Method `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 uncompressing when reading.

Super classes

`zarr::zarr_extension` -> `zarr::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()`

Method `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.

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

Usage:

```
zarr_codec_blosc$copy()
```

Returns: An instance of `zarr_codec_blosc`.

Method `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.

Method `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::zarr_extension` -> `zarr::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()`

Method `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.

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

Usage:

```
zarr_codec_bytes$copy()
```

Returns: An instance of `zarr_codec_bytes`.

Method `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.

Method `encode()`**:** This method writes an R object to a raw vector in the data type of the Zarr array. Prior to writing, any `NA` values are assigned the `fill_value` of the `data_type` of the Zarr array. Note that the logical type cannot encode `NA` in Zarr and any `NA` values are set to `FALSE`.

Usage:

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

Arguments:

`data` The data to be encoded.

Returns: A raw vector with the encoded data object.

Method `decode()`**:** This method takes a raw vector and converts it to an R object of an appropriate type. For all types other than logical, any data elements with the `fill_value` of the Zarr data type are set to `NA`.

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::zarr_extension` -> `zarr::zarr_codec` -> `zarr_codec_crc32c`

Methods

Public methods:

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

Method `new()`**:** Create a new "crc32c" codec object.

Usage:

```
zarr_codec_crc32c$new()
```

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.

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

Usage:

```
zarr_codec_crc32c$copy()
```

Returns: An instance of `zarr_codec_crc32c`.

Method `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.

Method `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::zarr_extension` -> `zarr::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()`

Method `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.

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

Usage:

`zarr_codec_gzip$copy()`

Returns: An instance of `zarr_codec_gzip`.

Method `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.

Method `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_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::zarr_extension` -> `zarr::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()`

Method `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.

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

Usage:

```
zarr_codec_transpose$copy()
```

Returns: An instance of `zarr_codec_transpose`.

Method `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.

Method `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_zstd` *Zarr "zstd" codec*

Description

This class provides the codec for "zstd" compression.

Super classes

`zarr::zarr_extension` -> `zarr::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()`

Method `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.

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

Usage:

```
zarr_codec_zstd$copy()
```

Returns: An instance of `zarr_codec_zstd`.

Method `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.

Method `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_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::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()`

Method `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.

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

Usage:

```
zarr_data_type$print()
```

Method `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_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()`

Method `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.

Method `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_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 class

`zarr::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$new()`
- `zarr_group$print()`
- `zarr_group$hierarchy()`
- `zarr_group$build_hierarchy()`
- `zarr_group$get_node()`
- `zarr_group$count_arrays()`
- `zarr_group$add_group()`
- `zarr_group$add_array()`
- `zarr_group$delete()`
- `zarr_group$delete_all()`

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

Usage:

```
zarr_group$new(name, metadata, parent, store)
```

Arguments:

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

`metadata` List with the metadata of the group.

`parent` The parent `zarr_group` instance of this new group, can be missing or `NULL` for the root group.

`store` The zarr_store instance to persist data in.

Returns: An instance of `zarr_group`.

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

Usage:

```
zarr_group$print()
```

Method `hierarchy()`**:** Prints the hierarchy of the group and its subgroups and arrays to the console. Usually called from the Zarr object or its root group to display the full group hierarchy.

Usage:

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

Arguments:

`idx, total` 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.

Method `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.

Method `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.

Method `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`.

Method `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, or `NULL` if the group could not be created.

Method `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.

Method `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.

Method `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::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$new()`
- `zarr_httpstore$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$get_metadata()`
- `zarr_httpstore$set_metadata()`
- `zarr_httpstore$is_group()`
- `zarr_httpstore$create_group()`
- `zarr_httpstore$create_array()`

Method `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.

Method `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.

Method `clear()`**:** Clearing the store is not supported.

Usage:

```
zarr_httpstore$clear()
```

Returns: `FALSE`.

Method `erase()`**:** Removing a key from the store is not supported.

Usage:

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

Arguments:

`key` Ignored.

Returns: `FALSE`.

Method `erase_prefix()`**:** Removing keys from the store is not supported.

Usage:

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

Arguments:

`prefix` Ignored.

Returns: `FALSE`.

Method `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.

Method `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.

Method `set()`**:** Storing a `(key, value)` pair is not supported.

Usage:

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

Arguments:

`key` Ignored.

`value` Ignored.

Returns: Self, invisibly.

Method `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.

Method `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` Ignored. The full data value is always returned.

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

Method `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.

Method `set_metadata()`**:** Setting metadata is not supported.

Usage:

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

Arguments:

`prefix` Ignored.

`metadata` Ignored.

Returns: Self, invisible

Method `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.

Method `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.

Method `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::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 '.'.

Methods

Public methods:

- `zarr_localstore$new()`
- `zarr_localstore$exists()`
- `zarr_localstore$clear()`
- `zarr_localstore$erase()`
- `zarr_localstore$erase_prefix()`
- `zarr_localstore$list_dir()`
- `zarr_localstore$list_prefix()`
- `zarr_localstore$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()`

Method `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.

Method `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.

Method `clear()`**:** Clear the store. Remove all keys and values from the store. Invoking this method deletes affected files on the file system and this action can not be undone. The only file that will remain is "zarr.json" or ".zgroup" (version 2) in the root of this store.

Usage:

```
zarr_localstore$clear()
```

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

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

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

Method `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.

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

Usage:

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

Arguments:

`path` 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.

Method `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::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$list_prefix()`
- `zarr_memorystore$set()`
- `zarr_memorystore$set_if_not_exists()`
- `zarr_memorystore$get()`
- `zarr_memorystore$get_metadata()`
- `zarr_memorystore$create_group()`
- `zarr_memorystore$create_array()`

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

Usage:

```
zarr_memorystore$new()
```

Returns: An instance of this class.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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`.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

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

Usage:

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

Arguments:

`path` 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.

Method `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.

Active bindings

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

`parent` (read-only) The parent of the node. For a root node this returns `NULL`, otherwise this `zarr_group` or `zarr_array` instance.

`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` (read-only) The metadata document of this node, a list.

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

Methods

Public methods:

- `zarr_node$new()`
- `zarr_node$print_attributes()`
- `zarr_node$set_attribute()`
- `zarr_node$delete_attributes()`
- `zarr_node$save()`

Method `new()`**:** Initialize a new node in a Zarr hierarchy.

Usage:

```
zarr_node$new(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. Must be omitted when initializing a root node.

`store` The store to persist data in. Ignored if a `parent` is specified.

Method `print_attributes()`**:** Print the metadata "attributes" to the console. Usually called by the zarr_group and zarr_array `print()` methods.

Usage:

```
zarr_node$print_attributes(...)
```

Arguments:

`...` Arguments passed to embedded functions. Of particular interest is `width = .` to specify the maximum width of the columns.

Method `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 must begin with a letter and be composed of letters, digits, and underscores, with a maximum length of 255 characters.

`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.

Method `delete_attributes()`**:** Delete attributes. If an attribute in `name` is not present this method simply returns.

Usage:

```
zarr_node$delete_attributes(name)
```

Arguments:

`name` Vector of names of the attributes to delete.

Returns: Self, invisibly.

Method `save()`**:** Persist any edits to the group or array to the store.

Usage:

```
zarr_node$save()
```

`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.

`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$new()`
- `zarr_store$clear()`
- `zarr_store$erase()`
- `zarr_store$erase_prefix()`
- `zarr_store$exists()`
- `zarr_store$get()`
- `zarr_store$getsize()`
- `zarr_store$getsize_prefix()`
- `zarr_store$is_empty()`

- `zarr_store$list()`
- `zarr_store$list_dir()`
- `zarr_store$list_prefix()`
- `zarr_store$set()`
- `zarr_store$set_if_not_exists()`
- `zarr_store$get_metadata()`
- `zarr_store$set_metadata()`
- `zarr_store$create_group()`
- `zarr_store$create_array()`

Method `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`.

`version` The version of the Zarr store. By default this is 3.

Returns: An instance of this class.

Method `clear()`**:** Clear the store. Remove all keys and values from the store.

Usage:

```
zarr_store$clear()
```

Returns: Self, invisibly.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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.

Method `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` *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()
z$add_group("/", "tst")
z$add_group("/tst", "subtst")
tst <- z[["/tst"]]
tst[["subtst"]]
```

Index