Zlib

Stability: 2 - Stable

Source Code: lib/zlib.js

The `node:zlib` module provides compression functionality implemented using Gzip, Deflate/Inflate, and Brotli.

To access it:

```
const zlib = require('node:zlib');
```

COPY

Compression and decompression are built around the Node.js Streams API .

Compressing or decompressing a stream (such as a file) can be accomplished by piping the source stream through a `zlib` `Transform` stream into a destination stream:

```
const { createGzip } = require('node:zlib');
const { pipeline } = require('node:stream');
const {
  createReadStream,
  createWriteStream,
} = require('node:fs');

const gzip = createGzip();
const source = createReadStream('input.txt');
const destination = createWriteStream('input.txt.gz');

pipeline(source, gzip, destination, (err) => {
  if (err) {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  }
});

// Or, Promisified

const { promisify } = require('node:util');
const pipe = promisify(pipeline);

async function do_gzip(input, output) {
  const gzip = createGzip();
  const source = createReadStream(input);
  const destination = createWriteStream(output);
  await pipe(source, gzip, destination);
}
```

```
do_gzip('input.txt', 'input.txt.gz')
  .catch((err) => {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  });
```

COPY

It is also possible to compress or decompress data in a single step:

```
const { deflate, unzip } = require('node:zlib');

const input = '.................................';
deflate(input, (err, buffer) => {
  if (err) {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  }
  console.log(buffer.toString('base64'));
});

const buffer = Buffer.from('eJzT0yMAAGTvBe8=', 'base64');
unzip(buffer, (err, buffer) => {
  if (err) {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  }
  console.log(buffer.toString());
});

// Or, Promisified

const { promisify } = require('node:util');
const do_unzip = promisify(unzip);

do_unzip(buffer)
  .then((buf) => console.log(buf.toString()))
  .catch((err) => {
    console.error('An error occurred:', err);
    process.exitCode = 1;
  });
```

COPY

Threadpool usage and performance considerations

All `zlib` APIs, except those that are explicitly synchronous, use the Node.js internal threadpool. This can lead to surprising effects and performance limitations in some applications.

Creating and using a large number of zlib objects simultaneously can cause significant memory fragmentation.

```
const zlib = require('node:zlib');

const payload = Buffer.from('This is some data');
```

```
// WARNING: DO NOT DO THIS!
for (let i = 0; i < 30000; ++i) {
  zlib.deflate(payload, (err, buffer) => {});
}
```

COPY

In the preceding example, 30,000 deflate instances are created concurrently. Because of how some operating systems handle memory allocation and deallocation, this may lead to significant memory fragmentation.

It is strongly recommended that the results of compression operations be cached to avoid duplication of effort.

Compressing HTTP requests and responses

The `node:zlib` module can be used to implement support for the `gzip`, `deflate` and `br` content-encoding mechanisms defined by HTTP.

The HTTP `Accept-Encoding` header is used within an HTTP request to identify the compression encodings accepted by the client. The `Content-Encoding` header is used to identify the compression encodings actually applied to a message.

The examples given below are drastically simplified to show the basic concept. Using `zlib` encoding can be expensive, and the results ought to be cached. See Memory usage tuning for more information on the speed/memory/compression tradeoffs involved in `zlib` usage.

```
// Client request example
const zlib = require('node:zlib');
const http = require('node:http');
const fs = require('node:fs');
const { pipeline } = require('node:stream');

const request = http.get({ host: 'example.com',
                           path: '/',
                           port: 80,
                           headers: { 'Accept-Encoding': 'br,gzip,deflate' } });
request.on('response', (response) => {
  const output = fs.createWriteStream('example.com_index.html');

  const onError = (err) => {
    if (err) {
      console.error('An error occurred:', err);
      process.exitCode = 1;
    }
  };

  switch (response.headers['content-encoding']) {
    case 'br':
      pipeline(response, zlib.createBrotliDecompress(), output, onError);
      break;
    // Or, just use zlib.createUnzip() to handle both of the following cases:
    case 'gzip':
      pipeline(response, zlib.createGunzip(), output, onError);
      break;
    case 'deflate':
      pipeline(response, zlib.createInflate(), output, onError);
      break;
    default:
      pipeline(response, output, onError);
      break;
```

```
  }
});
```

COPY

```
// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
const zlib = require('node:zlib');
const http = require('node:http');
const fs = require('node:fs');
const { pipeline } = require('node:stream');

http.createServer((request, response) => {
  const raw = fs.createReadStream('index.html');
  // Store both a compressed and an uncompressed version of the resource.
  response.setHeader('Vary', 'Accept-Encoding');
  let acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  const onError = (err) => {
    if (err) {
      // If an error occurs, there's not much we can do because
      // the server has already sent the 200 response code and
      // some amount of data has already been sent to the client.
      // The best we can do is terminate the response immediately
      // and log the error.
      response.end();
      console.error('An error occurred:', err);
    }
  };

  // Note: This is not a conformant accept-encoding parser.
  // See https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
  if (/\bdeflate\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'deflate' });
    pipeline(raw, zlib.createDeflate(), response, onError);
  } else if (/\bgzip\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'gzip' });
    pipeline(raw, zlib.createGzip(), response, onError);
  } else if (/\bbr\b/.test(acceptEncoding)) {
    response.writeHead(200, { 'Content-Encoding': 'br' });
    pipeline(raw, zlib.createBrotliCompress(), response, onError);
  } else {
    response.writeHead(200, {});
    pipeline(raw, response, onError);
  }
}).listen(1337);
```

COPY

By default, the `zlib` methods will throw an error when decompressing truncated data. However, if it is known that the data is incomplete, or the desire is to inspect only the beginning of a compressed file, it is possible to suppress the default error handling by changing the flushing method that is used to decompress the last chunk of input data:

```
// This is a truncated version of the buffer from the above examples
const buffer = Buffer.from('eJzT0yMA', 'base64');

zlib.unzip(
  buffer,
  // For Brotli, the equivalent is zlib.constants.BROTLI_OPERATION_FLUSH.
  { finishFlush: zlib.constants.Z_SYNC_FLUSH },
  (err, buffer) => {
    if (err) {
      console.error('An error occurred:', err);
      process.exitCode = 1;
    }
    console.log(buffer.toString());
  });
```

COPY

This will not change the behavior in other error-throwing situations, e.g. when the input data has an invalid format. Using this method, it will not be possible to determine whether the input ended prematurely or lacks the integrity checks, making it necessary to manually check that the decompressed result is valid.

Memory usage tuning

For zlib-based streams

From `zlib/zconf.h`, modified for Node.js usage:

The memory requirements for deflate are (in bytes):

```
(1 << (windowBits + 2)) + (1 << (memLevel + 9))
```

COPY

That is: 128K for `windowBits` = 15 + 128K for `memLevel` = 8 (default values) plus a few kilobytes for small objects.

For example, to reduce the default memory requirements from 256K to 128K, the options should be set to:

```
const options = { windowBits: 14, memLevel: 7 };
```

COPY

This will, however, generally degrade compression.

The memory requirements for inflate are (in bytes) `1 << windowBits`. That is, 32K for `windowBits` = 15 (default value) plus a few kilobytes for small objects.

This is in addition to a single internal output slab buffer of size `chunkSize`, which defaults to 16K.

The speed of `zlib` compression is affected most dramatically by the `level` setting. A higher level will result in better compression, but will take longer to complete. A lower level will result in less compression, but will be much faster.

In general, greater memory usage options will mean that Node.js has to make fewer calls to `zlib` because it will be able to process more data on each `write` operation. So, this is another factor that affects the speed, at the cost of memory usage.

For Brotli-based streams

There are equivalents to the zlib options for Brotli-based streams, although these options have different ranges than the zlib ones:

- zlib's `level` option matches Brotli's `BROTLI_PARAM_QUALITY` option.

- zlib's `windowBits` option matches Brotli's `BROTLI_PARAM_LGWIN` option.

See below for more details on Brotli-specific options.

Flushing

Calling `.flush()` on a compression stream will make `zlib` return as much output as currently possible. This may come at the cost of degraded compression quality, but can be useful when data needs to be available as soon as possible.

In the following example, `flush()` is used to write a compressed partial HTTP response to the client:

```
const zlib = require('node:zlib');
const http = require('node:http');
const { pipeline } = require('node:stream');

http.createServer((request, response) => {
  // For the sake of simplicity, the Accept-Encoding checks are omitted.
  response.writeHead(200, { 'content-encoding': 'gzip' });
  const output = zlib.createGzip();
  let i;

  pipeline(output, response, (err) => {
    if (err) {
      // If an error occurs, there's not much we can do because
      // the server has already sent the 200 response code and
      // some amount of data has already been sent to the client.
      // The best we can do is terminate the response immediately
      // and log the error.
      clearInterval(i);
      response.end();
      console.error('An error occurred:', err);
    }
  });

  i = setInterval(() => {
    output.write(`The current time is ${Date()}\n`, () => {
      // The data has been passed to zlib, but the compression algorithm may
      // have decided to buffer the data for more efficient compression.
      // Calling .flush() will make the data available as soon as the client
      // is ready to receive it.
      output.flush();
    });
  }, 1000);
}).listen(1337);
```

COPY

Constants

zlib constants

All of the constants defined in `zlib.h` are also defined on `require('node:zlib').constants`. In the normal course of operations, it will not be necessary to use these constants. They are documented so that their presence is not surprising. This section is taken almost directly from the zlib documentation.

Previously, the constants were available directly from `require('node:zlib')`, for instance `zlib.Z_NO_FLUSH`. Accessing the constants directly from the module is currently still possible but is deprecated.

Allowed flush values.

- `zlib.constants.Z_NO_FLUSH`
- `zlib.constants.Z_PARTIAL_FLUSH`
- `zlib.constants.Z_SYNC_FLUSH`
- `zlib.constants.Z_FULL_FLUSH`
- `zlib.constants.Z_FINISH`
- `zlib.constants.Z_BLOCK`
- `zlib.constants.Z_TREES`

Return codes for the compression/decompression functions. Negative values are errors, positive values are used for special but normal events.

- `zlib.constants.Z_OK`
- `zlib.constants.Z_STREAM_END`
- `zlib.constants.Z_NEED_DICT`
- `zlib.constants.Z_ERRNO`
- `zlib.constants.Z_STREAM_ERROR`
- `zlib.constants.Z_DATA_ERROR`
- `zlib.constants.Z_MEM_ERROR`
- `zlib.constants.Z_BUF_ERROR`
- `zlib.constants.Z_VERSION_ERROR`

Compression levels.

- `zlib.constants.Z_NO_COMPRESSION`
- `zlib.constants.Z_BEST_SPEED`
- `zlib.constants.Z_BEST_COMPRESSION`
- `zlib.constants.Z_DEFAULT_COMPRESSION`

Compression strategy.

- `zlib.constants.Z_FILTERED`
- `zlib.constants.Z_HUFFMAN_ONLY`
- `zlib.constants.Z_RLE`
- `zlib.constants.Z_FIXED`
- `zlib.constants.Z_DEFAULT_STRATEGY`

Brotli constants

There are several options and other constants available for Brotli-based streams:

Flush operations

The following values are valid flush operations for Brotli-based streams:

- `zlib.constants.BROTLI_OPERATION_PROCESS` (default for all operations)
- `zlib.constants.BROTLI_OPERATION_FLUSH` (default when calling `.flush()`)

- `zlib.constants.BROTLI_OPERATION_FINISH` (default for the last chunk)
- `zlib.constants.BROTLI_OPERATION_EMIT_METADATA`
  - This particular operation may be hard to use in a Node.js context, as the streaming layer makes it hard to know which data will end up in this frame. Also, there is currently no way to consume this data through the Node.js API.

Compressor options

There are several options that can be set on Brotli encoders, affecting compression efficiency and speed. Both the keys and the values can be accessed as properties of the `zlib.constants` object.

The most important options are:

- `BROTLI_PARAM_MODE`
  - `BROTLI_MODE_GENERIC` (default)
  - `BROTLI_MODE_TEXT`, adjusted for UTF-8 text
  - `BROTLI_MODE_FONT`, adjusted for WOFF 2.0 fonts
- `BROTLI_PARAM_QUALITY`
  - Ranges from `BROTLI_MIN_QUALITY` to `BROTLI_MAX_QUALITY`, with a default of `BROTLI_DEFAULT_QUALITY`.
- `BROTLI_PARAM_SIZE_HINT`
  - Integer value representing the expected input size; defaults to `0` for an unknown input size.

The following flags can be set for advanced control over the compression algorithm and memory usage tuning:

- `BROTLI_PARAM_LGWIN`
  - Ranges from `BROTLI_MIN_WINDOW_BITS` to `BROTLI_MAX_WINDOW_BITS`, with a default of `BROTLI_DEFAULT_WINDOW`, or up to `BROTLI_LARGE_MAX_WINDOW_BITS` if the `BROTLI_PARAM_LARGE_WINDOW` flag is set.
- `BROTLI_PARAM_LGBLOCK`
  - Ranges from `BROTLI_MIN_INPUT_BLOCK_BITS` to `BROTLI_MAX_INPUT_BLOCK_BITS`.
- `BROTLI_PARAM_DISABLE_LITERAL_CONTEXT_MODELING`
  - Boolean flag that decreases compression ratio in favour of decompression speed.
- `BROTLI_PARAM_LARGE_WINDOW`
  - Boolean flag enabling "Large Window Brotli" mode (not compatible with the Brotli format as standardized in RFC 7932).
- `BROTLI_PARAM_NPOSTFIX`
  - Ranges from `0` to `BROTLI_MAX_NPOSTFIX`.
- `BROTLI_PARAM_NDIRECT`
  - Ranges from `0` to `15 << NPOSTFIX` in steps of `1 << NPOSTFIX`.

Decompressor options

These advanced options are available for controlling decompression:

- `BROTLI_DECODER_PARAM_DISABLE_RING_BUFFER_REALLOCATION`
  - Boolean flag that affects internal memory allocation patterns.
- `BROTLI_DECODER_PARAM_LARGE_WINDOW`
  - Boolean flag enabling "Large Window Brotli" mode (not compatible with the Brotli format as standardized in RFC 7932).

Class: Options

Each zlib-based class takes an `options` object. No options are required.

Some options are only relevant when compressing and are ignored by the decompression classes.

- `flush` `<integer>` **Default:** `zlib.constants.Z_NO_FLUSH`
- `finishFlush` `<integer>` **Default:** `zlib.constants.Z_FINISH`

- `chunkSize` `<integer>` **Default:** `16 * 1024`
- `windowBits` `<integer>`
- `level` `<integer>` (compression only)
- `memLevel` `<integer>` (compression only)
- `strategy` `<integer>` (compression only)
- `dictionary` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` (deflate/inflate only, empty dictionary by default)
- `info` `<boolean>` (If `true`, returns an object with `buffer` and `engine`.)
- `maxOutputLength` `<integer>` Limits output size when using convenience methods. **Default:** `buffer.kMaxLength`

See the `deflateInit2` and `inflateInit2` documentation for more information.

Class: BrotliOptions

Each Brotli-based class takes an `options` object. All options are optional.

- `flush` `<integer>` **Default:** `zlib.constants.BROTLI_OPERATION_PROCESS`
- `finishFlush` `<integer>` **Default:** `zlib.constants.BROTLI_OPERATION_FINISH`
- `chunkSize` `<integer>` **Default:** `16 * 1024`
- `params` `<Object>` Key-value object containing indexed Brotli parameters.
- `maxOutputLength` `<integer>` Limits output size when using convenience methods. **Default:** `buffer.kMaxLength`

For example:

```
const stream = zlib.createBrotliCompress({
  chunkSize: 32 * 1024,
  params: {
    [zlib.constants.BROTLI_PARAM_MODE]: zlib.constants.BROTLI_MODE_TEXT,
    [zlib.constants.BROTLI_PARAM_QUALITY]: 4,
    [zlib.constants.BROTLI_PARAM_SIZE_HINT]: fs.statSync(inputFile).size,
  },
});
```

COPY

Class: zlib.BrotliCompress

Compress data using the Brotli algorithm.

Class: zlib.BrotliDecompress

Decompress data using the Brotli algorithm.

Class: zlib.Deflate

Compress data using deflate.

Class: zlib.DeflateRaw

Compress data using deflate, and do not append a `zlib` header.

Class: zlib.Gunzip

Decompress a gzip stream.

Class: zlib.Gzip

Compress data using gzip.

Class: zlib.Inflate

Decompress a deflate stream.

Class: zlib.InflateRaw

Decompress a raw deflate stream.

Class: zlib.Unzip

Decompress either a Gzip- or Deflate-compressed stream by auto-detecting the header.

Class: zlib.ZlibBase

Not exported by the `node:zlib` module. It is documented here because it is the base class of the compressor/decompressor classes.

This class inherits from `stream.Transform`, allowing `node:zlib` objects to be used in pipes and similar stream operations.

zlib.bytesRead

Stability: 0 - Deprecated: Use `zlib.bytesWritten` instead.

- `<number>`

Deprecated alias for `zlib.bytesWritten`. This original name was chosen because it also made sense to interpret the value as the number of bytes read by the engine, but is inconsistent with other streams in Node.js that expose values under these names.

zlib.bytesWritten

- `<number>`

The `zlib.bytesWritten` property specifies the number of bytes written to the engine, before the bytes are processed (compressed or decompressed, as appropriate for the derived class).

zlib.crc32(data[, value])

- `data` `<string>` | `<Buffer>` | `<TypedArray>` | `<DataView>` When `data` is a string, it will be encoded as UTF-8 before being used for computation.
- `value` `<integer>` An optional starting value. It must be a 32-bit unsigned integer. **Default:** `0`
- Returns: `<integer>` A 32-bit unsigned integer containing the checksum.

Computes a 32-bit Cyclic Redundancy Check checksum of `data`. If `value` is specified, it is used as the starting value of the checksum, otherwise, 0 is used as the starting value.

The CRC algorithm is designed to compute checksums and to detect error in data transmission. It's not suitable for cryptographic authentication.

To be consistent with other APIs, if the `data` is a string, it will be encoded with UTF-8 before being used for computation. If users only use Node.js to compute and match the checksums, this works well with other APIs that uses the UTF-8 encoding by default.

Some third-party JavaScript libraries compute the checksum on a string based on `str.charCodeAt()` so that it can be run in browsers. If users want to match the checksum computed with this kind of library in the browser, it's better to use the same library in Node.js if it also runs in Node.js. If users have to use `zlib.crc32()` to match the checksum produced by such a third-party library:

1. If the library accepts `Uint8Array` as input, use `TextEncoder` in the browser to encode the string into a `Uint8Array` with UTF-8 encoding, and compute the checksum based on the UTF-8 encoded string in the browser.
2. If the library only takes a string and compute the data based on `str.charCodeAt()`, on the Node.js side, convert the string into a buffer using `Buffer.from(str, 'utf16le')`.

```mjs
import zlib from 'node:zlib';
import { Buffer } from 'node:buffer';

let crc = zlib.crc32('hello');  // 907060870
crc = zlib.crc32('world', crc);  // 4192936109

crc = zlib.crc32(Buffer.from('hello', 'utf16le'));  // 1427272415
crc = zlib.crc32(Buffer.from('world', 'utf16le'), crc);  // 4150509955
```

```cjs
const zlib = require('node:zlib');
const { Buffer } = require('node:buffer');

let crc = zlib.crc32('hello');  // 907060870
crc = zlib.crc32('world', crc);  // 4192936109

crc = zlib.crc32(Buffer.from('hello', 'utf16le'));  // 1427272415
crc = zlib.crc32(Buffer.from('world', 'utf16le'), crc);  // 4150509955
```

COPY

zlib.close([callback])

- `callback` <Function>

Close the underlying handle.

zlib.flush([kind,]callback)

- `kind` **Default:** `zlib.constants.Z_FULL_FLUSH` for zlib-based streams, `zlib.constants.BROTLI_OPERATION_FLUSH` for Brotli-based streams.
- `callback` <Function>

Flush pending data. Don't call this frivolously, premature flushes negatively impact the effectiveness of the compression algorithm.

Calling this only flushes data from the internal `zlib` state, and does not perform flushing of any kind on the streams level. Rather, it behaves like a normal call to `.write()`, i.e. it will be queued up behind other pending writes and will only produce output when data is being read from the stream.

zlib.params(level, strategy, callback)

- `level` <integer>
- `strategy` <integer>
- `callback` <Function>

This function is only available for zlib-based streams, i.e. not Brotli.

Dynamically update the compression level and compression strategy. Only applicable to deflate algorithm.

zlib.reset()

Reset the compressor/decompressor to factory defaults. Only applicable to the inflate and deflate algorithms.

zlib.constants

Provides an object enumerating Zlib-related constants.

zlib.createBrotliCompress([options])

- `options` `<brotli options>`

Creates and returns a new `BrotliCompress` object.

zlib.createBrotliDecompress([options])

- `options` `<brotli options>`

Creates and returns a new `BrotliDecompress` object.

zlib.createDeflate([options])

- `options` `<zlib options>`

Creates and returns a new `Deflate` object.

zlib.createDeflateRaw([options])

- `options` `<zlib options>`

Creates and returns a new `DeflateRaw` object.

An upgrade of zlib from 1.2.8 to 1.2.11 changed behavior when `windowBits` is set to 8 for raw deflate streams. zlib would automatically set `windowBits` to 9 if was initially set to 8. Newer versions of zlib will throw an exception, so Node.js restored the original behavior of upgrading a value of 8 to 9, since passing `windowBits = 9` to zlib actually results in a compressed stream that effectively uses an 8-bit window only.

zlib.createGunzip([options])

- `options` `<zlib options>`

Creates and returns a new `Gunzip` object.

zlib.createGzip([options])

- `options` `<zlib options>`

Creates and returns a new `Gzip` object. See example .

zlib.createInflate([options])

- `options` `<zlib options>`

Creates and returns a new `Inflate` object.

zlib.createInflateRaw([options])

- `options` `<zlib options>`

Creates and returns a new `InflateRaw` object.

zlib.createUnzip([options])

- `options` `<zlib options>`

Creates and returns a new `Unzip` object.

Convenience methods

All of these take a `Buffer`, `TypedArray`, `DataView`, `ArrayBuffer` or string as the first argument, an optional second argument to supply options to the `zlib` classes and will call the supplied callback with `callback(error, result)`.

Every method has a `*Sync` counterpart, which accept the same arguments, but without a callback.

zlib.brotliCompress(buffer[, options], callback)

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`
- `callback` `<Function>`

zlib.brotliCompressSync(buffer[, options])

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`

Compress a chunk of data with `BrotliCompress`.

zlib.brotliDecompress(buffer[, options], callback)

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`
- `callback` `<Function>`

zlib.brotliDecompressSync(buffer[, options])

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<brotli options>`

Decompress a chunk of data with `BrotliDecompress`.

zlib.deflate(buffer[, options], callback)

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`
- `callback` `<Function>`

zlib.deflateSync(buffer[, options])

- `buffer` `<Buffer>` | `<TypedArray>` | `<DataView>` | `<ArrayBuffer>` | `<string>`
- `options` `<zlib options>`

Compress a chunk of data with `Deflate`.

zlib.deflateRaw(buffer[, options], callback)

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>
- callback <Function>

zlib.deflateRawSync(buffer[, options])

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

Compress a chunk of data with DeflateRaw.

zlib.gunzip(buffer[, options], callback)

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>
- callback <Function>

zlib.gunzipSync(buffer[, options])

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

Decompress a chunk of data with Gunzip.

zlib.gzip(buffer[, options], callback)

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>
- callback <Function>

zlib.gzipSync(buffer[, options])

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

Compress a chunk of data with Gzip.

zlib.inflate(buffer[, options], callback)

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>
- callback <Function>

zlib.inflateSync(buffer[, options])

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

Decompress a chunk of data with Inflate.

zlib.inflateRaw(buffer[, options], callback)

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

- callback <Function>

zlib.inflateRawSync(buffer[, options])

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

Decompress a chunk of data with InflateRaw.

zlib.unzip(buffer[, options], callback)

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>
- callback <Function>

zlib.unzipSync(buffer[, options])

- buffer <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <string>
- options <zlib options>

Decompress a chunk of data with Unzip.