Node.js (1)

Node.js v5.11.1 Documentation

Index | View on single page | View as JSON

Table of Contents

File System#

Stability: 2 - Stable

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Here is an example of the asynchronous version:

```
const fs = require('fs');

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Here is the synchronous version:

```
const fs = require('fs');

fs.unlinkSync('/tmp/hello');
console.log('successfully deleted /tmp/hello');
```

With the asynchronous methods there is no guaranteed ordering. So the following is prone to error:

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', (err, stats) => {
  if (err) throw err;
  console.log(`stats: ${JSON.stringify(stats)}`);
});
```

It could be that `fs.stat` is executed before `fs.rename`. The correct way to do this is to chain the callbacks.

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  fs.stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
  });
});
```

In busy processes, the programmer is *strongly encouraged* to use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete--halting all connections.

The relative path to a filename can be used. Remember, however, that this path will be relative to `process.cwd()`.

Most fs functions let you omit the callback argument. If you do, a default callback is used that rethrows errors. To get a trace to the original call site, set the `NODE_DEBUG` environment variable:

```
$ cat script.js
function bad() {
  require('fs').readFile('/');
}
bad();

$ env NODE_DEBUG=fs node script.js
fs.js:66
        throw err;
              ^
Error: EISDIR, read
    at rethrow (fs.js:61:21)
    at maybeCallback (fs.js:79:42)
    at Object.fs.readFile (fs.js:153:18)
    at bad (/path/to/script.js:2:17)
    at Object.<anonymous> (/path/to/script.js:5:1)
    <etc.>
```

Buffer API#

`fs` functions support passing and receiving paths as both strings and Buffers. The latter is intended to make it possible to work with filesystems that allow for non-UTF-8 filenames. For most typical uses, working with paths as Buffers will be unnecessary, as the string API converts to and from UTF-8 automatically.

Note that on certain file systems (such as NTFS and HFS+) filenames will always be encoded as UTF-8. On such file systems, passing non-UTF-8 encoded Buffers to `fs`

functions will not work as expected.

Class: fs.FSWatcher#

Objects returned from `fs.watch()` are of this type.

Event: 'change'#

- `event` <String> The type of fs change
- `filename` <String> The filename that changed (if relevant/available)

Emitted when something changes in a watched directory or file. See more details in `fs.watch()`.

Event: 'error'#

- `error` <Error>

Emitted when an error occurs.

watcher.close()#

Stop watching for changes on the given `fs.FSWatcher`.

Class: fs.ReadStream#

`ReadStream` is a Readable Stream.

Event: 'open'#

- `fd` <Number> Integer file descriptor used by the ReadStream.

Emitted when the ReadStream's file is opened.

readStream.path#

The path to the file the stream is reading from.

Class: fs.Stats#

Objects returned from `fs.stat()`, `fs.lstat()` and `fs.fstat()` and their synchronous counterparts are of this type.

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()` (only valid with `fs.lstat()`)
- `stats.isFIFO()`
- `stats.isSocket()`

For a regular file `util.inspect(stats)` would return a string very similar to this:

```
{
  dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT
}
```

Please note that `atime`, `mtime`, `birthtime`, and `ctime` are instances of `Date` object and to compare the values of these objects you should use appropriate methods. For most general uses `getTime()` will return the number of milliseconds elapsed since *1 January 1970 00:00:00 UTC* and this integer should be sufficient for any comparison, however there are additional methods which can be used for displaying fuzzy information. More details can be found in the MDN JavaScript Reference page.

Stat Time Values#

The times in the stat object have the following semantics:

- `atime` "Access Time" - Time when file data last accessed. Changed by the `mknod(2)`, `utimes(2)`, and `read(2)` system calls.
- `mtime` "Modified Time" - Time when file data last modified. Changed by the `mknod(2)`, `utimes(2)`, and `write(2)` system calls.
- `ctime` "Change Time" - Time when file status was last changed (inode data modification). Changed by the `chmod(2)`, `chown(2)`, `link(2)`, `mknod(2)`, `rename(2)`, `unlink(2)`, `utimes(2)`, `read(2)`, and `write(2)` system calls.
- `birthtime` "Birth Time" - Time of file creation. Set once when the file is created. On

filesystems where birthtime is not available, this field may instead hold either the `ctime` or `1970-01-01T00:00Z` (ie, unix epoch timestamp `0`). Note that this value may be greater than `atime` or `mtime` in this case. On Darwin and other FreeBSD variants, also set if the `atime` is explicitly set to an earlier value than the current `birthtime` using the `utimes(2)` system call.

Prior to Node v0.12, the `ctime` held the `birthtime` on Windows systems. Note that as of v0.12, `ctime` is not "creation time", and on Unix systems, it never was.

Class: fs.WriteStream#

`WriteStream` is a Writable Stream.

Event: 'open'#

- `fd` <Number> Integer file descriptor used by the WriteStream.

Emitted when the WriteStream's file is opened.

writeStream.bytesWritten#

The number of bytes written so far. Does not include data that is still queued for writing.

writeStream.path#

The path to the file the stream is writing to.

fs.access(path[, mode], callback)#

Tests a user's permissions for the file specified by `path`. `mode` is an optional integer that specifies the accessibility checks to be performed. The following constants define the possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values.

- `fs.F_OK` - File is visible to the calling process. This is useful for determining if a file exists, but says nothing about `rwx` permissions. Default if no `mode` is specified.
- `fs.R_OK` - File can be read by the calling process.
- `fs.W_OK` - File can be written by the calling process.
- `fs.X_OK` - File can be executed by the calling process. This has no effect on Windows (will behave like `fs.F_OK`).

The final argument, `callback`, is a callback function that is invoked with a possible error argument. If any of the accessibility checks fail, the error argument will be populated. The following example checks if the file `/etc/passwd` can be read and written by the current

process.

```
fs.access('/etc/passwd', fs.R_OK | fs.W_OK, (err) => {
  console.log(err ? 'no access!' : 'can read/write');
});
```

fs.accessSync(path[, mode])#

Synchronous version of fs.access(). This throws if any accessibility checks fail, and does nothing otherwise.

fs.appendFile(file, data[, options], callback)#

- file <String> | <Number> filename or file descriptor
- data <String> | <Buffer>
- options <Object> | <String>
    - encoding <String> | <Null> default = 'utf8'
    - mode <Number> default = 0o666
    - flag <String> default = 'a'
- callback <Function>

Asynchronously append data to a file, creating the file if it does not yet exist. data can be a string or a buffer.

Example:

```
fs.appendFile('message.txt', 'data to append', (err) => {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

If options is a string, then it specifies the encoding. Example:

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback);
```

Any specified file descriptor has to have been opened for appending.

Note: Specified file descriptors will not be closed automatically.

fs.appendFileSync(file, data[, options])#

The synchronous version of fs.appendFile(). Returns undefined.

fs.chmod(path, mode, callback)#

Asynchronous chmod(2). No arguments other than a possible exception are given to the completion callback.

fs.chmodSync(path, mode)#

Synchronous chmod(2). Returns `undefined`.

fs.chown(path, uid, gid, callback)#

Asynchronous chown(2). No arguments other than a possible exception are given to the completion callback.

fs.chownSync(path, uid, gid)#

Synchronous chown(2). Returns `undefined`.

fs.close(fd, callback)#

Asynchronous close(2). No arguments other than a possible exception are given to the completion callback.

fs.closeSync(fd)#

Synchronous close(2). Returns `undefined`.

fs.createReadStream(path[, options])#

Returns a new `ReadStream` object. (See Readable Stream).

Be aware that, unlike the default value set for `highWaterMark` on a readable stream (16 kb), the stream returned by this method has a default value of 64 kb for the same parameter.

`options` is an object or string with the following defaults:

```
{
  flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

options can include start and end values to read a range of bytes from the file instead of the entire file. Both start and end are inclusive and start at 0. The encoding can be any one of those accepted by Buffer.

If fd is specified, ReadStream will ignore the path argument and will use the specified file descriptor. This means that no 'open' event will be emitted. Note that fd should be blocking; non-blocking fds should be passed to net.Socket.

If autoClose is false, then the file descriptor won't be closed, even if there's an error. It is your responsibility to close it and make sure there's no file descriptor leak. If autoClose is set to true (default behavior), on error or end the file descriptor will be closed automatically.

mode sets the file mode (permission and sticky bits), but only if the file was created.

An example to read the last 10 bytes of a file which is 100 bytes long:

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

If options is a string, then it specifies the encoding.

fs.createWriteStream(path[, options])#

Returns a new WriteStream object. (See Writable Stream).

options is an object or string with the following defaults:

```
{
  flags: 'w',
  defaultEncoding: 'utf8',
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

options may also include a start option to allow writing data at some position past the beginning of the file. Modifying a file rather than replacing it may require a flags mode of r+ rather than the default mode w. The defaultEncoding can be any one of those accepted by Buffer.

If autoClose is set to true (default behavior) on error or end the file descriptor will be closed automatically. If autoClose is false, then the file descriptor won't be closed, even if there's an error. It is your responsibility to close it and make sure there's no file descriptor leak.

Like ReadStream, if fd is specified, WriteStream will ignore the path argument and will

use the specified file descriptor. This means that no `'open'` event will be emitted. Note that `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

If `options` is a string, then it specifies the encoding.

fs.exists(path, callback)#

`Stability: 0 - Deprecated: Use fs.stat() or fs.access() instead.`

Test whether or not the given path exists by checking with the file system. Then call the `callback` argument with either true or false. Example:

```
fs.exists('/etc/passwd', (exists) => {
  console.log(exists ? 'it\'s there' : 'no passwd!');
});
```

`fs.exists()` should not be used to check if a file exists before calling `fs.open()`. Doing so introduces a race condition since other processes may change the file's state between the two calls. Instead, user code should call `fs.open()` directly and handle the error raised if the file is non-existent.

fs.existsSync(path)#

`Stability: 0 - Deprecated: Use fs.statSync() or fs.accessSync() instead.`

Synchronous version of `fs.exists()`. Returns `true` if the file exists, `false` otherwise.

fs.fchmod(fd, mode, callback)#

Asynchronous fchmod(2). No arguments other than a possible exception are given to the completion callback.

fs.fchmodSync(fd, mode)#

Synchronous fchmod(2). Returns `undefined`.

fs.fchown(fd, uid, gid, callback)#

Asynchronous fchown(2). No arguments other than a possible exception are given to the completion callback.

fs.fchownSync(fd, uid, gid)#

Synchronous fchown(2). Returns `undefined`.

fs.fdatasync(fd, callback)#

Asynchronous fdatasync(2). No arguments other than a possible exception are given to the completion callback.

fs.fdatasyncSync(fd)#

Synchronous fdatasync(2). Returns `undefined`.

fs.fstat(fd, callback)#

Asynchronous fstat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `fstat()` is identical to `stat()`, except that the file to be stat-ed is specified by the file descriptor `fd`.

fs.fstatSync(fd)#

Synchronous fstat(2). Returns an instance of `fs.Stats`.

fs.fsync(fd, callback)#

Asynchronous fsync(2). No arguments other than a possible exception are given to the completion callback.

fs.fsyncSync(fd)#

Synchronous fsync(2). Returns `undefined`.

fs.ftruncate(fd, len, callback)#

Asynchronous ftruncate(2). No arguments other than a possible exception are given to the completion callback.

fs.ftruncateSync(fd, len)#

Synchronous ftruncate(2). Returns `undefined`.

fs.futimes(fd, atime, mtime, callback)#

Change the file timestamps of a file referenced by the supplied file descriptor.

fs.futimesSync(fd, atime, mtime)#

Synchronous version of `fs.futimes()`. Returns `undefined`.

fs.lchmod(path, mode, callback)#

Asynchronous lchmod(2). No arguments other than a possible exception are given to the completion callback.

Only available on Mac OS X.

fs.lchmodSync(path, mode)#

Synchronous lchmod(2). Returns `undefined`.

fs.lchown(path, uid, gid, callback)#

Asynchronous lchown(2). No arguments other than a possible exception are given to the completion callback.

fs.lchownSync(path, uid, gid)#

Synchronous lchown(2). Returns `undefined`.

fs.link(srcpath, dstpath, callback)#

Asynchronous link(2). No arguments other than a possible exception are given to the completion callback.

fs.linkSync(srcpath, dstpath)#

Synchronous link(2). Returns `undefined`.

fs.lstat(path, callback)#

Asynchronous lstat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `lstat()` is identical to `stat()`, except that if `path` is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fs.lstatSync(path)#

Synchronous lstat(2). Returns an instance of `fs.Stats`.

fs.mkdir(path[, mode], callback)#

Asynchronous mkdir(2). No arguments other than a possible exception are given to the completion callback. `mode` defaults to `0o777`.

fs.mkdirSync(path[, mode])#

Synchronous mkdir(2). Returns `undefined`.

fs.mkdtemp(prefix, callback)#

Creates a unique temporary directory.

Generates six random characters to be appended behind a required `prefix` to create a unique temporary directory.

The created folder path is passed as a string to the callback's second parameter.

Example:

```
fs.mkdtemp('/tmp/foo-', (err, folder) => {
  console.log(folder);
    // Prints: /tmp/foo-itXde2
});
```

fs.mkdtempSync(template)#

The synchronous version of [`fs.mkdtemp()`][]. Returns the created folder path.

fs.open(path, flags[, mode], callback)#

Asynchronous file open. See open(2). `flags` can be:

- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs'` - Open file for reading in synchronous mode. Instructs the operating system to bypass the local file system cache.

This is primarily useful for opening files on NFS mounts as it allows you to skip the potentially stale local cache. It has a very real impact on I/O performance so don't use this flag unless you need it.

Note that this doesn't turn `fs.open()` into a synchronous blocking call. If that's what you want then you should be using `fs.openSync()`

- `'rs+'` - Open file for reading and writing, telling the OS to open it synchronously. See notes for `'rs'` about using this with caution.
- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` - Like `'w'` but fails if `path` exists.
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` - Like `'w+'` but fails if `path` exists.
- `'a'` - Open file for appending. The file is created if it does not exist.
- `'ax'` - Like `'a'` but fails if `path` exists.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` - Like `'a+'` but fails if `path` exists.

`mode` sets the file mode (permission and sticky bits), but only if the file was created. It defaults to `0666`, readable and writable.

The callback gets two arguments `(err, fd)`.

The exclusive flag `'x'` (`O_EXCL` flag in open(2)) ensures that `path` is newly created. On POSIX systems, `path` is considered to exist even if it is a symlink to a non-existent file. The exclusive flag may or may not work with network file systems.

`flags` can also be a number as documented by open(2); commonly used constants are available from `require('constants')`. On Windows, flags are translated to their equivalent ones where applicable, e.g. `O_WRONLY` to `FILE_GENERIC_WRITE`, or `O_EXCL|O_CREAT` to `CREATE_NEW`, as accepted by CreateFileW.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fs.openSync(path, flags[, mode])#

Synchronous version of `fs.open()`. Returns an integer representing the file descriptor.

fs.read(fd, buffer, offset, length, position, callback)#

Read data from the file specified by `fd`.

`buffer` is the buffer that the data will be written to.

`offset` is the offset in the buffer to start writing at.

`length` is an integer specifying the number of bytes to read.

`position` is an integer specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position.

The callback is given the three arguments, `(err, bytesRead, buffer)`.

fs.readdir(path, callback)#

Asynchronous readdir(3). Reads the contents of a directory. The callback gets two arguments `(err, files)` where `files` is an array of the names of the files in the directory excluding `'.'` and `'..'`.

fs.readdirSync(path)#

Synchronous readdir(3). Returns an array of filenames excluding `'.'` and `'..'`.

fs.readFile(file[, options], callback)#

- `file` <String> | <Integer> filename or file descriptor
- `options` <Object> | <String>
    - `encoding` <String> | <Null> default = `null`
    - `flag` <String> default = `'r'`
- `callback` <Function>

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If `options` is a string, then it specifies the encoding. Example:

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

Any specified file descriptor has to support reading.

Note: Specified file descriptors will not be closed automatically.

fs.readFileSync(file[, options])#

Synchronous version of `fs.readFile`. Returns the contents of the `file`.

If the `encoding` option is specified then this function returns a string. Otherwise it returns a buffer.

fs.readlink(path, callback)#

Asynchronous readlink(2). The callback gets two arguments `(err, linkString)`.

fs.readlinkSync(path)#

Synchronous readlink(2). Returns the symbolic link's string value.

fs.realpath(path[, cache], callback)#

Asynchronous realpath(2). The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd` to resolve relative paths. `cache` is an object literal of mapped paths that can be used to force a specific path resolution or avoid additional `fs.stat` calls for known real paths.

Example:

```
var cache = {'/etc':'/private/etc'};
fs.realpath('/etc/passwd', cache, (err, resolvedPath) => {
  if (err) throw err;
  console.log(resolvedPath);
});
```

fs.readSync(fd, buffer, offset, length, position)#

Synchronous version of `fs.read()`. Returns the number of `bytesRead`.

fs.realpathSync(path[, cache])#

Synchronous realpath(2). Returns the resolved path. `cache` is an object literal of mapped paths that can be used to force a specific path resolution or avoid additional `fs.stat` calls for known real paths.

fs.rename(oldPath, newPath, callback)#

Asynchronous rename(2). No arguments other than a possible exception are given to the completion callback.

fs.renameSync(oldPath, newPath)#

Synchronous rename(2). Returns `undefined`.

fs.rmdir(path, callback)#

Asynchronous rmdir(2). No arguments other than a possible exception are given to the completion callback.

fs.rmdirSync(path)#

Synchronous rmdir(2). Returns `undefined`.

fs.stat(path, callback)#

Asynchronous stat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. See the `fs.Stats` section for more information.

fs.statSync(path)#

Synchronous stat(2). Returns an instance of `fs.Stats`.

fs.symlink(target, path[, type], callback)#

Asynchronous symlink(2). No arguments other than a possible exception are given to the completion callback. The `type` argument can be set to `'dir'`, `'file'`, or `'junction'` (default is `'file'`) and is only available on Windows (ignored on other platforms). Note that Windows junction points require the destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

Here is an example below:

```
fs.symlink('./foo', './new-port');
```

It creates a symbolic link named "new-port" that points to "foo".

fs.symlinkSync(target, path[, type])#

Synchronous symlink(2). Returns `undefined`.

fs.truncate(path, len, callback)#

Asynchronous truncate(2). No arguments other than a possible exception are given to the completion callback. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncate()` is called.

fs.truncateSync(path, len)#

Synchronous truncate(2). Returns `undefined`.

fs.unlink(path, callback)#

Asynchronous unlink(2). No arguments other than a possible exception are given to the completion callback.

fs.unlinkSync(path)#

Synchronous unlink(2). Returns `undefined`.

fs.unwatchFile(filename[, listener])#

Stop watching for changes on `filename`. If `listener` is specified, only that particular listener is removed. Otherwise, *all* listeners are removed and you have effectively stopped watching `filename`.

Calling `fs.unwatchFile()` with a filename that is not being watched is a no-op, not an error.

Note: `fs.watch()` is more efficient than `fs.watchFile()` and `fs.unwatchFile()`. `fs.watch()` should be used instead of `fs.watchFile()` and `fs.unwatchFile()` when possible.

fs.utimes(path, atime, mtime, callback)#

Change file timestamps of the file referenced by the supplied path.

Note: the arguments `atime` and `mtime` of the following related functions does follow the below rules:

- If the value is a numberable string like `'123456789'`, the value would get converted to corresponding number.
- If the value is `NaN` or `Infinity`, the value would get converted to `Date.now()`.

fs.utimesSync(path, atime, mtime)#

Synchronous version of `fs.utimes()`. Returns `undefined`.

fs.watch(filename[, options][, listener])#

Watch for changes on `filename`, where `filename` is either a file or a directory. The returned object is a `fs.FSWatcher`.

The second argument is optional. The `options` if provided should be an object. The supported boolean members are `persistent` and `recursive`. `persistent` indicates whether the process should continue to run as long as files are being watched. `recursive` indicates whether all subdirectories should be watched, or only the current directory. This applies when a directory is specified, and only on supported platforms (See Caveats).

The default is `{ persistent: true, recursive: false }`.

The listener callback gets two arguments `(event, filename)`. `event` is either `'rename'` or `'change'`, and `filename` is the name of the file which triggered the event.

Caveats#

The `fs.watch` API is not 100% consistent across platforms, and is unavailable in some situations.

The recursive option is only supported on OS X and Windows.

Availability#

This feature depends on the underlying operating system providing a way to be notified of filesystem changes.

- On Linux systems, this uses `inotify`.

- On BSD systems, this uses `kqueue`.
- On OS X, this uses `kqueue` for files and 'FSEvents' for directories.
- On SunOS systems (including Solaris and SmartOS), this uses `event ports`.
- On Windows systems, this feature depends on `ReadDirectoryChangesW`.

If the underlying functionality is not available for some reason, then `fs.watch` will not be able to function. For example, watching files or directories on network file systems (NFS, SMB, etc.) often doesn't work reliably or at all.

You can still use `fs.watchFile`, which uses stat polling, but it is slower and less reliable.

Inodes#

On Linux and OS X systems, `fs.watch()` resolves the path to an inode and watches the inode. If the watched path is deleted and recreated, it is assigned a new inode. The watch will emit an event for the delete but will continue watching the *original* inode. Events for the new inode will not be emitted. This is expected behavior.

Filename Argument#

Providing `filename` argument in the callback is only supported on Linux and Windows. Even on supported platforms, `filename` is not always guaranteed to be provided. Therefore, don't assume that `filename` argument is always provided in the callback, and have some fallback logic if it is null.

```
fs.watch('somedir', (event, filename) => {
  console.log(`event is: ${event}`);
  if (filename) {
    console.log(`filename provided: ${filename}`);
  } else {
    console.log('filename not provided');
  }
});
```

fs.watchFile(filename[, options], listener)#

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

The `options` argument may be omitted. If provided, it should be an object. The `options` object may contain a boolean named `persistent` that indicates whether the process should continue to run as long as files are being watched. The `options` object may specify an `interval` property indicating how often the target should be polled in milliseconds. The default is `{ persistent: true, interval: 5007 }`.

The `listener` gets two arguments the current stat object and the previous stat object:

```
fs.watchFile('message.text', (curr, prev) => {
  console.log(`the current mtime is: ${curr.mtime}`);
  console.log(`the previous mtime was: ${prev.mtime}`);
});
```

These stat objects are instances of `fs.Stat`.

If you want to be notified when the file was modified, not just accessed, you need to compare `curr.mtime` and `prev.mtime`.

Note: when an `fs.watchFile` operation results in an `ENOENT` error, it will invoke the listener once, with all the fields zeroed (or, for dates, the Unix Epoch). In Windows, `blksize` and `blocks` fields will be `undefined`, instead of zero. If the file is created later on, the listener will be called again, with the latest stat objects. This is a change in functionality since v0.10.

Note: `fs.watch()` is more efficient than `fs.watchFile` and `fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and `fs.unwatchFile` when possible.

fs.write(fd, buffer, offset, length[, position], callback)#

Write `buffer` to the file specified by `fd`.

`offset` and `length` determine the part of the buffer to be written.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'`, the data will be written at the current position. See pwrite(2).

The callback will be given three arguments `(err, written, buffer)` where `written` specifies how many *bytes* were written from `buffer`.

Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fs.write(fd, data[, position[, encoding]], callback)#

Write `data` to the file specified by `fd`. If `data` is not a Buffer instance then the value will be coerced to a string.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'` the data will be written at the current position. See

pwrite(2).

`encoding` is the expected string encoding.

The callback will receive the arguments `(err, written, string)` where `written` specifies how many *bytes* the passed string required to be written. Note that bytes written is not the same as string characters. See `Buffer.byteLength`.

Unlike when writing `buffer`, the entire string must be written. No substring may be specified. This is because the byte offset of the resulting data may not be the same as the string offset.

Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fs.writeFile(file, data[, options], callback)#

- `file` <String> | <Integer> filename or file descriptor
- `data` <String> | <Buffer>
- `options` <Object> | <String>
    - `encoding` <String> | <Null> default = `'utf8'`
    - `mode` <Number> default = `0o666`
    - `flag` <String> default = `'w'`
- `callback` <Function>

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer.

The `encoding` option is ignored if `data` is a buffer. It defaults to `'utf8'`.

Example:

```
fs.writeFile('message.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

If `options` is a string, then it specifies the encoding. Example:

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

Any specified file descriptor has to support writing.

Note that it is unsafe to use `fs.writeFile` multiple times on the same file without waiting

for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

Note: Specified file descriptors will not be closed automatically.

fs.writeFileSync(file, data[, options])#

The synchronous version of `fs.writeFile()`. Returns `undefined`.

fs.writeSync(fd, buffer, offset, length[, position])#

fs.writeSync(fd, data[, position[, encoding]])#

Synchronous versions of `fs.write()`. Returns the number of bytes written.