Path

Stability: 2 - Stable

Source Code: lib/path.js

The `node:path` module provides utilities for working with file and directory paths. It can be accessed using:

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

COPY

Windows vs. POSIX

The default operation of the `node:path` module varies based on the operating system on which a Node.js application is running. Specifically, when running on a Windows operating system, the `node:path` module will assume that Windows-style paths are being used.

So using `path.basename()` might yield different results on POSIX and Windows:

On POSIX:

```
path.basename('C:\\temp\\myfile.html');
// Returns: 'C:\\temp\\myfile.html'
```

COPY

On Windows:

```
path.basename('C:\\temp\\myfile.html');
// Returns: 'myfile.html'
```

COPY

To achieve consistent results when working with Windows file paths on any operating system, use `path.win32` :

On POSIX and Windows:

```
path.win32.basename('C:\\temp\\myfile.html');
// Returns: 'myfile.html'
```

COPY

To achieve consistent results when working with POSIX file paths on any operating system, use `path.posix` :

On POSIX and Windows:

```
path.posix.basename('/tmp/myfile.html');
// Returns: 'myfile.html'
```

COPY

On Windows Node.js follows the concept of per-drive working directory. This behavior can be observed when using a drive path without a backslash. For example, `path.resolve('C:\\')` can potentially return a different result than `path.resolve('C:')` . For more information, see this MSDN page .

path.basename(path[, suffix])

- `path` <string>
- `suffix` <string> An optional suffix to remove
- Returns: <string>

The `path.basename()` method returns the last portion of a `path`, similar to the Unix `basename` command. Trailing directory separators are ignored.

```
path.basename('/foo/bar/baz/asdf/quux.html');
// Returns: 'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html');
// Returns: 'quux'
```

COPY

Although Windows usually treats file names, including file extensions, in a case-insensitive manner, this function does not. For example, `C:\\foo.html` and `C:\\foo.HTML` refer to the same file, but `basename` treats the extension as a case-sensitive string:

```
path.win32.basename('C:\\foo.html', '.html');
// Returns: 'foo'

path.win32.basename('C:\\foo.HTML', '.html');
// Returns: 'foo.HTML'
```

COPY

A TypeError is thrown if `path` is not a string or if `suffix` is given and is not a string.

path.delimiter

- <string>

Provides the platform-specific path delimiter:

- `;` for Windows
- `:` for POSIX

For example, on POSIX:

```
console.log(process.env.PATH);
// Prints: '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env.PATH.split(path.delimiter);
// Returns: ['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

COPY

On Windows:

```
console.log(process.env.PATH);
// Prints: 'C:\Windows\system32;C:\Windows;C:\Program Files\node\'

process.env.PATH.split(path.delimiter);
// Returns ['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\node\\']
```

COPY

path.dirname(path)

- `path` `<string>`
- Returns: `<string>`

The `path.dirname()` method returns the directory name of a `path`, similar to the Unix `dirname` command. Trailing directory separators are ignored, see `path.sep`.

```
path.dirname('/foo/bar/baz/asdf/quux');
// Returns: '/foo/bar/baz/asdf'
```

COPY

A `TypeError` is thrown if `path` is not a string.

path.extname(path)

- `path` `<string>`
- Returns: `<string>`

The `path.extname()` method returns the extension of the `path`, from the last occurrence of the `.` (period) character to end of string in the last portion of the `path`. If there is no `.` in the last portion of the `path`, or if there are no `.` characters other than the first character of the basename of `path` (see `path.basename()`), an empty string is returned.

```
path.extname('index.html');
// Returns: '.html'

path.extname('index.coffee.md');
// Returns: '.md'

path.extname('index.');
// Returns: '.'

path.extname('index');
// Returns: ''

path.extname('.index');
// Returns: ''

path.extname('.index.md');
// Returns: '.md'
```

COPY

A `TypeError` is thrown if `path` is not a string.

path.format(pathObject)

- `pathObject` `<Object>` Any JavaScript object having the following properties:
  - `dir` `<string>`
  - `root` `<string>`
  - `base` `<string>`
  - `name` `<string>`
  - `ext` `<string>`
- Returns: `<string>`

The `path.format()` method returns a path string from an object. This is the opposite of `path.parse()`.

When providing properties to the `pathObject` remember that there are combinations where one property has priority over another:

- `pathObject.root` is ignored if `pathObject.dir` is provided
- `pathObject.ext` and `pathObject.name` are ignored if `pathObject.base` exists

For example, on POSIX:

```
// If `dir`, `root` and `base` are provided,
// `${dir}${path.sep}${base}`
// will be returned. `root` is ignored.
path.format({
  root: '/ignored',
  dir: '/home/user/dir',
  base: 'file.txt',
});
// Returns: '/home/user/dir/file.txt'

// `root` will be used if `dir` is not specified.
// If only `root` is provided or `dir` is equal to `root` then the
// platform separator will not be included. `ext` will be ignored.
path.format({
  root: '/',
  base: 'file.txt',
  ext: 'ignored',
});
// Returns: '/file.txt'

// `name` + `ext` will be used if `base` is not specified.
path.format({
  root: '/',
  name: 'file',
  ext: '.txt',
});
// Returns: '/file.txt'

// The dot will be added if it is not specified in `ext`.
path.format({
  root: '/',
  name: 'file',
  ext: 'txt',
});
// Returns: '/file.txt'
```

COPY

On Windows:

```
path.format({
  dir: 'C:\\path\\dir',
  base: 'file.txt',
});
// Returns: 'C:\\path\\dir\\file.txt'
```

COPY

path.matchesGlob(path, pattern)

Stability: 1 - Experimental

- `path` <string> The path to glob-match against.
- `pattern` <string> The glob to check the path against.
- Returns: <boolean> Whether or not the `path` matched the `pattern`.

The `path.matchesGlob()` method determines if `path` matches the `pattern`.

For example:

```
path.matchesGlob('/foo/bar', '/foo/*'); // true
path.matchesGlob('/foo/bar*', 'foo/bird'); // false
```

COPY

A TypeError is thrown if `path` or `pattern` are not strings.

path.isAbsolute(path)

- `path` <string>
- Returns: <boolean>

The `path.isAbsolute()` method determines if `path` is an absolute path.

If the given `path` is a zero-length string, `false` will be returned.

For example, on POSIX:

```
path.isAbsolute('/foo/bar'); // true
path.isAbsolute('/baz/..');  // true
path.isAbsolute('qux/');     // false
path.isAbsolute('.');        // false
```

COPY

On Windows:

```
path.isAbsolute('//server');    // true
path.isAbsolute('\\\\server');  // true
path.isAbsolute('C:/foo/..');   // true
path.isAbsolute('C:\\foo\\..'); // true
path.isAbsolute('bar\\baz');    // false
path.isAbsolute('bar/baz');     // false
path.isAbsolute('.');           // false
```

COPY

A TypeError is thrown if `path` is not a string.

path.join([...paths])

- `...paths` <string> A sequence of path segments
- Returns: <string>

The `path.join()` method joins all given `path` segments together using the platform-specific separator as a delimiter, then normalizes the resulting path.

Zero-length `path` segments are ignored. If the joined path string is a zero-length string then `'.'` will be returned, representing the current working directory.

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');
// Returns: '/foo/bar/baz/asdf'

path.join('foo', {}, 'bar');
// Throws 'TypeError: Path must be a string. Received {}'
```

COPY

A `TypeError` is thrown if any of the path segments is not a string.

path.normalize(path)

- `path` `<string>`
- Returns: `<string>`

The `path.normalize()` method normalizes the given `path`, resolving `'..'` and `'.'` segments.

When multiple, sequential path segment separation characters are found (e.g. `/` on POSIX and either `\` or `/` on Windows), they are replaced by a single instance of the platform-specific path segment separator (`/` on POSIX and `\` on Windows). Trailing separators are preserved.

If the `path` is a zero-length string, `'.'` is returned, representing the current working directory.

On POSIX, the types of normalization applied by this function do not strictly adhere to the POSIX specification. For example, this function will replace two leading forward slashes with a single slash as if it was a regular absolute path, whereas a few POSIX systems assign special meaning to paths beginning with exactly two forward slashes. Similarly, other substitutions performed by this function, such as removing `..` segments, may change how the underlying system resolves the path.

For example, on POSIX:

```
path.normalize('/foo/bar//baz/asdf/quux/..');
// Returns: '/foo/bar/baz/asdf'
```

COPY

On Windows:

```
path.normalize('C:\\temp\\\\foo\\bar\\..\\');
// Returns: 'C:\\temp\\foo\\'
```

COPY

Since Windows recognizes multiple path separators, both separators will be replaced by instances of the Windows preferred separator (`\`):

```
path.win32.normalize('C:////temp\\\\/\\/\\/foo/bar');
// Returns: 'C:\\temp\\foo\\bar'
```

COPY

A `TypeError` is thrown if `path` is not a string.

path.parse(path)

- `path` `<string>`
- Returns: `<Object>`

The `path.parse()` method returns an object whose properties represent significant elements of the `path`. Trailing directory separators are ignored, see `path.sep`.

The returned object will have the following properties:

- `dir` <string>
- `root` <string>
- `base` <string>
- `name` <string>
- `ext` <string>

For example, on POSIX:

```
path.parse('/home/user/dir/file.txt');
// Returns:
// { root: '/',
//   dir: '/home/user/dir',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file' }
```

COPY

```
┌─────────────────────┬────────────┐
│          dir        │    base    │
├──────┐              ├──────┬─────┤
│ root │              │ name │ ext │
"  /    home/user/dir / file  .txt "
└──────┴──────────────┴──────┴─────┘
(All spaces in the "" line should be ignored. They are purely for formatting.)
```

COPY

On Windows:

```
path.parse('C:\\path\\dir\\file.txt');
// Returns:
// { root: 'C:\\',
//   dir: 'C:\\path\\dir',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file' }
```

COPY

```
┌─────────────────────┬────────────┐
│          dir        │    base    │
├──────┐              ├──────┬─────┤
│ root │              │ name │ ext │
" C:\      path\dir   \ file  .txt "
└──────┴──────────────┴──────┴─────┘
(All spaces in the "" line should be ignored. They are purely for formatting.)
```

COPY

A `TypeError` is thrown if `path` is not a string.

path.posix

- <Object>

The `path.posix` property provides access to POSIX specific implementations of the `path` methods.

The API is accessible via `require('node:path').posix` or `require('node:path/posix')`.

path.relative(from, to)

- `from` <string>
- `to` <string>
- Returns: <string>

The `path.relative()` method returns the relative path from `from` to `to` based on the current working directory. If `from` and `to` each resolve to the same path (after calling `path.resolve()` on each), a zero-length string is returned.

If a zero-length string is passed as `from` or `to`, the current working directory will be used instead of the zero-length strings.

For example, on POSIX:

```
path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');
// Returns: '../../impl/bbb'
```

COPY

On Windows:

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb');
// Returns: '..\\..\\impl\\bbb'
```

COPY

A TypeError is thrown if either `from` or `to` is not a string.

path.resolve([...paths])

- `...paths` <string> A sequence of paths or path segments
- Returns: <string>

The `path.resolve()` method resolves a sequence of paths or path segments into an absolute path.

The given sequence of paths is processed from right to left, with each subsequent `path` prepended until an absolute path is constructed. For instance, given the sequence of path segments: `/foo`, `/bar`, `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz` because `'baz'` is not an absolute path but `'/bar' + '/' + 'baz'` is.

If, after processing all given `path` segments, an absolute path has not yet been generated, the current working directory is used.

The resulting path is normalized and trailing slashes are removed unless the path is resolved to the root directory.

Zero-length `path` segments are ignored.

If no `path` segments are passed, `path.resolve()` will return the absolute path of the current working directory.

```
path.resolve('/foo/bar', './baz');
// Returns: '/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/');
// Returns: '/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
// If the current working directory is /home/myself/node,
// this returns '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

COPY

A `TypeError` is thrown if any of the arguments is not a string.

path.sep

- `<string>`

Provides the platform-specific path segment separator:

- `\` on Windows
- `/` on POSIX

For example, on POSIX:

```
'foo/bar/baz'.split(path.sep);
// Returns: ['foo', 'bar', 'baz']
```

COPY

On Windows:

```
'foo\\bar\\baz'.split(path.sep);
// Returns: ['foo', 'bar', 'baz']
```

COPY

On Windows, both the forward slash (`/`) and backward slash (`\`) are accepted as path segment separators; however, the `path` methods only add backward slashes (`\`).

path.toNamespacedPath(path)

- `path` `<string>`
- Returns: `<string>`

On Windows systems only, returns an equivalent namespace-prefixed path for the given `path` . If `path` is not a string, `path` will be returned without modifications.

This method is meaningful only on Windows systems. On POSIX systems, the method is non-operational and always returns `path` without modifications.

path.win32

- `<Object>`

The `path.win32` property provides access to Windows-specific implementations of the `path` methods.

The API is accessible via `require('node:path').win32` or `require('node:path/win32')` .