path-scurry

Extremely high performant utility for building tools that read the file system, minimizing filesystem and path string munging operations to the greatest degree possible.

Ugh, yet another file traversal thing on npm?

Yes. None of the existing ones gave me exactly what I wanted.

Well what is it you wanted?

While working on glob, I found that I needed a module to very efficiently manage the traversal over a folder tree, such that:

1. No `readdir()` or `stat()` would ever be called on the same file or directory more than one time.
2. No `readdir()` calls would be made if we can be reasonably sure that the path is not a directory. (Ie, a previous `readdir()` or `stat()` covered the path, and `ent.isDirectory()` is false.)

3. `path.resolve()`, `dirname()`, `basename()`, and other string-parsing/munging operations are be minimized. This means it has to track “provisional” child nodes that may not exist (and if we find that they *don’t* exist, store that information as well, so we don’t have to ever check again).
4. The API is not limited to use as a stream/iterator/etc. There are many cases where an API like node’s `fs` is preferrable.
5. It’s more important to prevent excess syscalls than to be up to date, but it should be smart enough to know what it *doesn’t* know, and go get it seamlessly when requested.
6. Do not blow up the JS heap allocation if operating on a directory with a huge number of entries.
7. Handle all the weird aspects of Windows paths, like UNC paths and drive letters and wrongway slashes, so that the consumer can return canonical platform-specific paths without having to parse or join or do any error-prone string munging.

PERFORMANCE

JavaScript people throw around the word “blazing” a lot. I hope that this module doesn’t blaze anyone. But it does go very fast, in the cases it’s optimized for, if used properly.

PathScurry provides ample opportunities to get extremely good performance, as well as several options to trade performance for convenience.

Benchmarks can be run by executing `npm run bench`.

As is always the case, doing more means going slower, doing less means going faster, and there are trade offs between speed and memory usage.

PathScurry makes heavy use of LRUCache to efficiently cache whatever it can, and `Path` objects remain in the graph for the lifetime of the walker, so repeated calls with a single PathScurry object will be extremely fast. However, adding items to a cold cache means "doing more", so in those cases, we pay a price. Nothing is free, but every effort has been made to reduce costs wherever possible.

Also, note that a "cache as long as possible" approach means that changes to the filesystem may not be reflected in the results of repeated PathScurry operations.

For resolving string paths, `PathScurry` ranges from 5-50 times faster than `path.resolve` on repeated resolutions, but around 100 to 1000 times *slower* on the first resolution. If your program is spending a lot of time resolving the *same* paths repeatedly (like, thousands or millions of times), then this can be beneficial. But both implementations are pretty fast, and speeding up an infrequent operation from 4µs to 400ns is not going to move the needle on your app's performance.

For walking file system directory trees, a lot depends on how often a given PathScurry object will be used, and also on the walk method used.

With default settings on a folder tree of 100,000 items, consisting of around a 10-to-1 ratio of normal files to directories, PathScurry performs comparably to @nodelib/fs.walk, which is the fastest and most reliable file system walker I could find. As far as I can tell, it's almost impossible to go much faster in a Node.js program, just based on how fast you can push syscalls out to the fs thread pool.

On my machine, that is about 1000-1200 completed walks per second for async or stream walks, and around 500-600 walks per second synchronously.

In the warm cache state, PathScurry's performance increases around 4x for async `for await` iteration, 10-15x faster for streams and synchronous `for of` iteration, and anywhere from 30x to 80x faster for the rest.

```
# walk 100,000 fs entries, 10/1 file/dir
ratio
# operations / ms
 New PathScurry object  |  Reuse
PathScurry object
     stream:  1112.589  |  13974.917
sync stream:   492.718  |  15028.343
 async walk:  1095.648  |  32706.395
  sync walk:   527.632  |  46129.772
 async iter:  1288.821  |   5045.510
  sync iter:   498.496  |  17920.746
```

A hand-rolled walk calling `entry.readdir()` and recursing through the entries can benefit even more from caching, with greater flexibility and without the overhead of streams or generators.

The cold cache state is still limited by the costs of file system operations, but with a warm cache, the only bottleneck is CPU speed and VM optimizations. Of course, in that case, some care must be taken to ensure that you don't lose performance as a result

`async path.readlink()`

Return the `Path` object referenced by the `path` as a symbolic link.

If the `path` is not a symbolic link, or any error occurs, returns `undefined`.

`path.readlinkSync()`

Synchronous `path.readlink()`

`async path.lstat()`

Call `lstat` on the path object, and fill it in with details determined.

If path does not exist, or any other error occurs, returns `undefined`, and marks the path as "unknown" type.

`path.lstatSync()`

Synchronous `path.lstat()`

`async path.realpath()`

Call `realpath` on the path, and return a Path object corresponding to the result, or `undefined` if any error occurs.

`path.realpathSync()`

Synchornous `path.realpath()`

If the nearest common ancestor is the root, then an absolute path is returned.

`path.relativePosix(): string`

Return the relative path from the PathWalker cwd to the supplied path string or entry, using `/` path separators.

If the nearest common ancestor is the root, then an absolute path is returned.

On posix platforms (ie, all platforms except Windows), this is identical to `pw.relative(path)`.

On Windows systems, it returns the resulting string as a `/`-delimited path. If an absolute path is returned (because the target does not share a common ancestor with `pw.cwd`), then a full absolute UNC path will be returned. Ie, instead of `'C:\\foo\\bar`, it would return `//?/C:/foo/bar`.

`async path.readdir()`

Return an array of `Path` objects found by reading the associated path entry.

If path is not a directory, or if any error occurs, returns `[]`, and marks all children as provisional and non-existent.

`path.readdirSync()`

Synchronous `path.readdir()`

of silly mistakes, like calling `readdir()` on entries that you know are not directories.

```
# manual recursive iteration functions
      cold cache   |  warm cache
async:  1164.901   |  17923.320
   cb:  1101.127   |  40999.344
zalgo:  1082.240   |  66689.936
 sync:   526.935   |  87097.591
```

In this case, the speed improves by around 10-20x in the async case, 40x in the case of using `entry.readdirCB` with protections against synchronous callbacks, and 50-100x with callback deferrals disabled, and *several hundred times faster* for synchronous iteration.

If you can think of a case that is not covered in these benchmarks, or an implementation that performs significantly better than PathScurry, please let me know.

USAGE

```
// hybrid module, load with either
        method
import { PathScurry, Path } from 'path-
        scurry'
// or:
const { PathScurry, Path } =
        require('path-scurry')
```

```
// very simple example, say we want to
        find and
// delete all the .DS_Store files in a
        given path
// note that the API is very similar to
        just a
// naive walk with fs.readdir()
import { unlink } from 'fs/promises'

// easy way, iterate over the directory
        and do the thing
const pw = new PathScurry(process.cwd())
for await (const entry of pw) {
  if (entry.isFile() && entry.name ===
        '.DS_Store') {
    unlink(entry.fullpath())
  }
}

// here it is as a manual recursive
        method
const walk = async (entry: Path) => {
  const promises: Promise<any> = []
  // readdir doesn't throw on non-
        directories, it just doesn't
  // return any entries, to save stack
        trace costs.
  // Items are returned in arbitrary
        unsorted order
  for (const child of await
        pw.readdir(entry)) {
    // each child is a Path object
    if (child.name === '.DS_Store' &&
        child.isFile()) {
```

`path.fullpath()`

The fully resolved path to the entry.

`path.fullpathPosix()`

The fully resolved path to the entry, using / separators.

On posix systems, this is identical to `path.fullpath()`. On windows, this will return a fully resolved absolute UNC path using / separators. Eg, instead of `'C:\\foo\\bar'`, it will return `'//?/C:/foo/bar'`.

`path.isFile()`, `path.isDirectory()`, **etc.**

Same as the identical `fs.Dirent.isX()` methods.

`path.isUnknown()`

Returns true if the path's type is unknown. Always returns true when the path is known to not exist.

`path.resolve(p: string)`

Return a `Path` object associated with the provided path string as resolved from the current Path object.

`path.relative(): string`

Return the relative path from the PathWalker cwd to the supplied path string or entry.

`path.isNamed(name: string): boolean`

Return true if the path is a match for the given path name. This handles case sensitivity and unicode normalization.

Note: even on case-sensitive systems, it is **not** safe to test the equality of the `.name` property to determine whether a given pathname matches, due to unicode normalization mismatches.

Always use this method instead of testing the `path.name` property directly.

`path.isCWD`

Set to true if this `Path` object is the current working directory of the `PathScurry` collection that contains it.

`path.getType()`

Returns the type of the Path object, `'File'`, `'Directory'`, etc.

`path.isType(t: type)`

Returns true if `is{t}()` returns true.

For example, `path.isType('Directory')` is equivalent to `path.isDirectory()`.

`path.depth()`

Return the depth of the Path entry within the directory tree. Root paths have a depth of `0`.

```
      // could also do pw.resolve(entry,
        child.name),
      // just like fs.readdir walking,
        but .fullpath is
      // a *slightly* more efficient
        shorthand.
      promises.push(unlink(child.fullpath()))
    } else if (child.isDirectory()) {
      promises.push(walk(child))
    }
  }
  return Promise.all(promises)
}

walk(pw.cwd).then(() => {
  console.log('all .DS_Store files
        removed')
})

const pw2 = new PathScurry('/a/b/c') //
        pw2.cwd is the Path for /a/b/c
const relativeDir = pw2.cwd.resolve('../
        x') // Path entry for '/a/b/x'
const relative2 = pw2.cwd.resolve('/a/b/
        d/../x') // same path, same
        entry
assert.equal(relativeDir, relative2)
```

API

Full TypeDoc API

There are platform-specific classes exported, but for the most part, the default `PathScurry` and `Path` exports are what you most likely need, unless you are testing behavior for other platforms.

Intended public API is documented here, but the full documentation does include internal types, which should not be accessed directly.

Interface `PathScurryOpts`

The type of the `options` argument passed to the `PathScurry` constructor.

- `nocase`: Boolean indicating that file names should be compared case-insensitively. Defaults to `true` on darwin and win32 implementations, `false` elsewhere.

  Warning Performing case-insensitive matching on a case-sensitive filesystem will result in occasionally very bizarre behavior. Performing case-sensitive matching on a case-insensitive filesystem may negatively impact performance.
- `childrenCacheSize`: Number of child entries to cache, in order to speed up `resolve()` and `readdir()` calls. Defaults to `16 * 1024` (ie, `16384`).

  Setting it to a higher value will run the risk of JS heap allocation errors on large directory trees. Setting it to `256` or smaller will significantly reduce the construction time and data consumption overhead, but with the downside of operations being slower on large directory trees. Setting it to `0` will mean that

Class `Path` implements fs.Dirent

Object representing a given path on the filesystem, which may or may not exist.

Note that the actual class in use will be either `PathWin32` or `PathPosix`, depending on the implementation of `PathScurry` in use. They differ in the separators used to split and join path strings, and the handling of root paths.

In `PathPosix` implementations, paths are split and joined using the `'/'` character, and `'/'` is the only root path ever in use.

In `PathWin32` implementations, paths are split using either `'/'` or `'\\'` and joined using `'\\'`, and multiple roots may be in use based on the drives and UNC paths encountered. UNC paths such as `//?/C:/` that identify a drive letter, will be treated as an alias for the same root entry as their associated drive letter (in this case `'C:\\'`).

`path.name`

Name of this file system entry.

Important: *always* test the path name against any test string using the `isNamed` method, and not by directly comparing this string. Otherwise, unicode path strings that the system sees as identical will not be properly treated as the same path, leading to incorrect behavior and possible security issues.

`async pw.lstat(entry = pw.cwd)`

Call `fs.lstat` on the supplied string or Path object, and fill in as much information as possible, returning the updated `Path` object.

Returns `undefined` if the entry does not exist, or if any error is encountered.

Note that some `Stats` data (such as `ino`, `dev`, and `mode`) will not be supplied. For those things, you'll need to call `fs.lstat` yourself.

`pw.lstatSync(entry = pw.cwd)`

Synchronous `pw.lstat()`

`pw.realpath(entry = pw.cwd, opts = { withFileTypes: false })`

Call `fs.realpath` on the supplied string or Path object, and return the realpath if available.

Returns `undefined` if any error occurs.

May be called as `pw.realpath({ withFileTypes: boolean })` to run on `pw.cwd`.

`pw.realpathSync(entry = pw.cwd, opts = { withFileTypes: false })`

Synchronous `pw.realpath()`

effectively no operations are cached, and this module will be roughly the same speed as `fs` for file system operations, and *much* slower than `path.resolve()` for repeated path resolution.

- `fs` An object that will be used to override the default `fs` methods. Any methods that are not overridden will use Node's built-in implementations.
  - lstatSync
  - readdir (callback `withFileTypes` Dirent variant, used for readdirCB and most walks)
  - readdirSync
  - readlinkSync
  - realpathSync
  - promises: Object containing the following async methods:
    - lstat
    - readdir (Dirent variant only)
    - readlink
    - realpath

Interface `WalkOptions`

The options object that may be passed to all walk methods.

- `withFileTypes`: Boolean, default true. Indicates that `Path` objects should be returned. Set to `false` to get string paths instead.
- `follow`: Boolean, default false. Attempt to read directory entries from symbolic links. Otherwise, only actual directories are

traversed. Regardless of this setting, a given target path will only ever be walked once, meaning that a symbolic link to a previously traversed directory will never be followed.

Setting this imposes a slight performance penalty, because `readlink` must be called on all symbolic links encountered, in order to avoid infinite cycles.

- `filter`: Function `(entry: Path) => boolean`. If provided, will prevent the inclusion of any entry for which it returns a falsey value. This will not prevent directories from being traversed if they do not pass the filter, though it will prevent the directories themselves from being included in the results. By default, if no filter is provided, then all entries are included in the results.
- `walkFilter`: Function `(entry: Path) => boolean`. If provided, will prevent the traversal of any directory (or in the case of `follow:true` symbolic links to directories) for which the function returns false. This will not prevent the directories themselves from being included in the result set. Use `filter` for that.

Note that TypeScript return types will only be inferred properly from static analysis if the `withFileTypes` option is omitted, or a constant `true` or `false` value.

Class `PathScurry`

The main interface. Defaults to an appropriate class based on the current platform.

Can be called as `pw.readdir({ withFileTypes: boolean })` as well.

Returns `[]` if no entries are found, or if any error occurs.

Note that TypeScript return types will only be inferred properly from static analysis if the `withFileTypes` option is omitted, or a constant `true` or `false` value.

`pw.readdirSync(dir = pw.cwd, opts = { withFileTypes: true })`

Synchronous `pw.readdir()`

`async pw.readlink(link = pw.cwd, opts = { withFileTypes: false })`

Call `fs.readlink` on the supplied string or Path object, and return the result.

Can be called as `pw.readlink({ withFileTypes: boolean })` as well.

Returns `undefined` if any error occurs (for example, if the argument is not a symbolic link), or a `Path` object if `withFileTypes` is explicitly set to `true`, or a string otherwise.

Note that TypeScript return types will only be inferred properly from static analysis if the `withFileTypes` option is omitted, or a constant `true` or `false` value.

`pw.readlinkSync(link = pw.cwd, opts = { withFileTypes: false })`

Synchronous `pw.readlink()`

pw.relativePosix(path: string | Path): string

Return the relative path from the PathWalker cwd to the supplied path string or entry, using `/` path separators.

If the nearest common ancestor is the root, then an absolute path is returned.

On posix platforms (ie, all platforms except Windows), this is identical to `pw.relative(path)`.

On Windows systems, it returns the resulting string as a `/`-delimited path. If an absolute path is returned (because the target does not share a common ancestor with `pw.cwd`), then a full absolute UNC path will be returned. Ie, instead of `'C:\\foo\\bar`, it would return `//?/C:/foo/bar`.

pw.basename(path: string | Path): string

Return the basename of the provided string or Path.

pw.dirname(path: string | Path): string

Return the parent directory of the supplied string or Path.

async pw.readdir(dir = pw.cwd, opts = { withFileTypes: true })

Read the directory and resolve to an array of strings if `withFileTypes` is explicitly set to `false` or Path objects otherwise.

Use `PathScurryWin32`, `PathScurryDarwin`, or `PathScurryPosix` if implementation-specific behavior is desired.

All walk methods may be called with a `WalkOptions` argument to walk over the object’s current working directory with the supplied options.

async pw.walk(entry?: string | Path | WalkOptions, opts?: WalkOptions)

Walk the directory tree according to the options provided, resolving to an array of all entries found.

pw.walkSync(entry?: string | Path | WalkOptions, opts?: WalkOptions)

Walk the directory tree according to the options provided, returning an array of all entries found.

pw.iterate(entry?: string | Path | WalkOptions, opts?: WalkOptions)

Iterate over the directory asynchronously, for use with `for await of`. This is also the default async iterator method.

pw.iterateSync(entry?: string | Path | WalkOptions, opts?: WalkOptions)

Iterate over the directory synchronously, for use with `for of`. This is also the default sync iterator method.

`pw.stream(entry?: string | Path | WalkOptions, opts?: WalkOptions)`

Return a Minipass stream that emits each entry or path string in the walk. Results are made available asynchronously.

`pw.streamSync(entry?: string | Path | WalkOptions, opts?: WalkOptions)`

Return a Minipass stream that emits each entry or path string in the walk. Results are made available synchronously, meaning that the walk will complete in a single tick if the stream is fully consumed.

`pw.cwd`

Path object representing the current working directory for the PathScurry.

`pw.chdir(path: string)`

Set the new effective current working directory for the scurry object, so that `path.relative()` and `path.relativePosix()` return values relative to the new cwd path.

`pw.depth(path?: Path | string): number`

Return the depth of the specified path (or the PathScurry cwd) within the directory tree.

Root entries have a depth of `0`.

`pw.resolve(...paths: string[])`

Caching `path.resolve()`.

Significantly faster than `path.resolve()` if called repeatedly with the same paths. Significantly slower otherwise, as it builds out the cached Path entries.

To get a `Path` object resolved from the `PathScurry`, use `pw.cwd.resolve(path)`. Note that `Path.resolve` only takes a single string argument, not multiple.

`pw.resolvePosix(...paths: string[])`

Caching `path.resolve()`, but always using posix style paths.

This is identical to `pw.resolve(...paths)` on posix systems (ie, everywhere except Windows).

On Windows, it returns the full absolute UNC path using `/` separators. Ie, instead of `'C:\\foo\\bar`, it would return `//?/C:/foo/bar`.

`pw.relative(path: string | Path): string`

Return the relative path from the PathWalker cwd to the supplied path string or entry.

If the nearest common ancestor is the root, then an absolute path is returned.