minimatch

A minimal matching utility.

This is the matching library used internally by npm.

It works by converting glob expressions into JavaScript `RegExp` objects.

Usage

```
// hybrid module, load with require() or
        import
import { minimatch } from 'minimatch'
// or:
const { minimatch } =
        require('minimatch')

minimatch('bar.foo', '*.foo') // true!
minimatch('bar.foo', '*.bar') // false!
minimatch('bar.foo', '*.+(bar|foo)', {
        debug: true }) // true, and
        noisy!
```

Features

Supports these glob features:

- Brace Expansion
- Extended glob matching
- "Globstar" `**` matching
- Posix character classes, like `[[:alpha:]]`, supporting the full range of Unicode characters. For example, `[[:alpha:]]` will match against `'é'`, though `[a-zA-Z]` will not. Collating symbol and set matching is not supported, so `[[=e=]]` will *not* match `'é'` and `[[.ch.]]` will not match `'ch'` in locales where `ch` is considered a single character.

See:

- `man sh`
- `man bash` Pattern Matching
- `man 3 fnmatch`
- `man 5 gitignore`

Windows

Please only use forward-slashes in glob expressions.

Though windows uses either `/` or `\` as its path separator, only `/` characters are used by this glob implementation. You must use forward-slashes **only** in glob expressions. Back-slashes in patterns will always be interpreted as escape characters, not path separators.

Note that `\` or `/` *will* be interpreted as path separators in paths on Windows, and will match against `/` in glob expressions.

So just always use `/` in patterns.

UNC Paths

On Windows, UNC paths like `//?/c:/...` or `//ComputerName/Share/...` are handled specially.

- Patterns starting with a double-slash followed by some non-slash characters will preserve their double-slash. As a result, a pattern like `//*` will match `//x`, but not `/x`.
- Patterns staring with `//?/<drive letter>:` will *not* treat the `?` as a wildcard character. Instead, it will be treated as a normal string.
- Patterns starting with `//?/<drive letter>:/...` will match file paths starting with `<drive letter>:/...`, and vice versa, as if the `//?/` was not present. This behavior only is present when the drive letters are a case-insensitive match to one another. The remaining portions of the path/pattern are compared case sensitively, unless `nocase:true` is set.

Note that specifying a UNC path using `\` characters as path separators is always allowed in the file path argument, but only allowed in the pattern argument when `windowsPathsNoEscape: true` is set in the options.

Minimatch Class

Create a minimatch object by instantiating the `minimatch.Minimatch` class.

```
var Minimatch =
        require('minimatch').Minimatch
var mm = new Minimatch(pattern, options)
```

Properties

- `pattern` The original pattern the minimatch object represents.
- `options` The options supplied to the constructor.
- `set` A 2-dimensional array of regexp or string expressions. Each row in the array corresponds to a brace-expanded pattern. Each item in the row corresponds to a single path-part. For example, the pattern `{a,b/c}/d` would expand to a set of patterns like:

```
    [ [ a, d ]
  , [ b, c, d ] ]
```

If a portion of the pattern doesn't have any "magic" in it (that is, it's something like `"foo"` rather than `fo*o?`), then it will be left as a string rather than converted to a regular expression.

- `regexp` Created by the `makeRe` method. A single regular expression expressing the entire pattern. This is useful in cases

start with `<start><pattern>`. However, `<start>!(<pattern>*)*` *will* match paths starting with `<start><pattern>`, because the empty string can match against the negated portion. In this library, `<start>!(<pattern>*|)*` will *not* match any pattern starting with `<start>`, due to a difference in precisely which patterns are considered "greedy" in Regular Expressions vs bash path expansion. This may be fixable, but not without incurring some complexity and performance costs, and the trade-off seems to not be worth pursuing.

Note that `fnmatch(3)` in libc is an extremely naive string comparison matcher, which does not do anything special for slashes. This library is designed to be used in glob searching and file walkers, and so it does do special things with `/`. Thus, `foo*` will not match `foo/bar` in this library, even though it would in `fnmatch(3)`.

`!` characters at the start of a pattern will negate the pattern multiple times.

If a pattern starts with #, then it is treated as a comment, and will not match anything. Use `\#` to match a literal # at the start of a line, or set the `nocomment` flag to suppress this behavior.

The double-star character `**` is supported by default, unless the `noglobstar` flag is set. This is supported in the manner of bsdglob and bash 4.1, where `**` only has special significance if it is the only thing in a path part. That is, `a/**/b` will match `a/x/y/b`, but `a/**b` will not.

If an escaped pattern has no matches, and the `nonull` flag is set, then minimatch.match returns the pattern as-provided, rather than interpreting the character escapes. For example, `minimatch.match([], "\\*a\\?")` will return `"\\*a\\?"` rather than `"*a?"`. This is akin to setting the `nullglob` option in bash, except that it does not resolve escaped pattern characters.

If brace expansion is not disabled, then it is performed before any other interpretation of the glob pattern. Thus, a pattern like `+(a|{b),c)}`, which would not be valid in bash or zsh, is expanded **first** into the set of `+(a|b)` and `+(a|c)`, and those patterns are checked for validity. Since those two are valid, matching proceeds.

Negated extglob patterns are handled as closely as possible to Bash semantics, but there are some cases with negative extglobs which are exceedingly difficult to express in a JavaScript regular expression. In particular the negated pattern `<start>!(<pattern>*|)*` will in bash match anything that does not

where you wish to use the pattern somewhat like `fnmatch(3)` with `FNM_PATH` enabled.

- `negate` True if the pattern is negated.
- `comment` True if the pattern is a comment.
- `empty` True if the pattern is `""`.

Methods

- `makeRe()` Generate the `regexp` member if necessary, and return it. Will return `false` if the pattern is invalid.
- `match(fname)` Return true if the filename matches the pattern, or false otherwise.
- `matchOne(fileArray, patternArray, partial)` Take a `/`-split filename, and match it against a single row in the `regExpSet`. This method is mainly for internal use, but is exposed so that it can be used by a glob-walker that needs to avoid excessive filesystem calls.
- `hasMagic()` Returns true if the parsed pattern contains any magic characters. Returns false if all comparator parts are string literals. If the `magicalBraces` option is set on the constructor, then it will consider brace expansions which are not otherwise magical to be magic. If not set, then a pattern like `a{b,c}d` will return `false`, because neither `abd` nor `acd` contain any special glob characters.

  This does **not** mean that the pattern string can be used as a literal filename, as it may contain magic glob characters that are escaped. For example, the pattern `\\*` or `[*]` would not be

considered to have magic, as the matching portion parses to the literal string `'*'` and would match a path named `'*'`, not `'\\*'` or `'[*]'`. The `minimatch.unescape()` method may be used to remove escape characters.

All other methods are internal, and will be called as necessary.

minimatch(path, pattern, options)

Main export. Tests a path against the pattern using the options.

```
var isJS = minimatch(file, '*.js', {
        matchBase: true })
```

minimatch.filter(pattern, options)

Returns a function that tests its supplied argument, suitable for use with `Array.filter`. Example:

```
var javascripts =
        fileList.filter(minimatch.filter('*.js',
        { matchBase: true }))
```

minimatch.escape(pattern, options = {})

Escape all magic characters in a glob pattern, so that it will only ever match literal strings.

Specifically, while the `Minimatch.match()` method will optimize the file path string in the same ways, resulting in the same matches, it will fail when tested with the regular expression provided by `Minimatch.makeRe()`, unless the path string is first processed with `minimatch.levelTwoFileOptimize()` or similar.

platform

When set to `win32`, this will trigger all windows-specific behaviors (special handling for UNC paths, and treating `\` as separators in file paths for comparison.)

Defaults to the value of `process.platform`.

Comparisons to other fnmatch/glob implementations

While strict compliance with the existing standards is a worthwhile goal, some discrepancies exist between minimatch and other implementations. Some are intentional, and some are unavoidable.

If the pattern starts with a `!` character, then it is negated. Set the `nonegate` flag to suppress this behavior, and treat leading `!` characters normally. This is perhaps relevant if you wish to start the pattern with a negative extglob pattern like `!(a|B)`. Multiple

- 1 - (default) Remove cases where a double-dot `..` follows a pattern portion that is not `**`, `.`, `..`, or empty `''`. For example, the pattern `./a/b/../*` is converted to `./a/*`, and so it will match the path string `./a/c`, but not the path string `./a/b/../c`. Dots and empty path portions in the pattern are preserved.
- 2 (or higher) - Much more aggressive optimizations, suitable for use with file-walking cases:
  - Remove cases where a double-dot `..` follows a pattern portion that is not `**`, `.`, or empty `''`. Remove empty and `.` portions of the pattern, where safe to do so (ie, anywhere other than the last position, the first position, or the second position in a pattern starting with `/`, as this may indicate a UNC path on Windows).
  - Convert patterns containing `<pre>/**/../<p>/<rest>` into the equivalent `<pre>/{..,**}/<p>/<rest>`, where `<p>` is a a pattern portion other than `.`, `..`, `**`, or empty `''`.
  - Dedupe patterns where a `**` portion is present in one and omitted in another, and it is not the final path portion, and they are otherwise equivalent. So `{a/**/b,a/b}` becomes `a/**/b`, because `**` matches against an empty path portion.
  - Dedupe patterns where a `*` portion is present in one, and a non-dot pattern other than `**`, `.`, `..`, or `''` is in the same position in the other. So `a/{*,x}/b` becomes `a/*/b`, because `*` can match against `x`.

  While these optimizations improve the performance of file-walking use cases such as glob (ie, the reason this module exists), there are cases where it will fail to match a literal string that would have been matched in optimization level 1 or 0.

If the `windowsPathsNoEscape` option is used, then characters are escaped by wrapping in `[]`, because a magic character wrapped in a character class can only be satisfied by that exact character.

Slashes (and backslashes in `windowsPathsNoEscape` mode) cannot be escaped or unescaped.

minimatch.unescape(pattern, options = {})

Un-escape a glob string that may contain some escaped characters.

If the `windowsPathsNoEscape` option is used, then square-brace escapes are removed, but not backslash escapes. For example, it will turn the string `'[*]'` into `*`, but it will not turn `'\\*'` into `'*'`, because `\` is a path separator in `windowsPathsNoEscape` mode.

When `windowsPathsNoEscape` is not set, then both brace escapes and backslash escapes are removed.

Slashes (and backslashes in `windowsPathsNoEscape` mode) cannot be escaped or unescaped.

minimatch.match(list, pattern, options)

Match against the list of files, in the style of fnmatch or glob. If nothing is matched, and options.nonull is set, then return a list containing the pattern itself.

```
var javascripts =
        minimatch.match(fileList,
        '*.js', { matchBase: true })
```

minimatch.makeRe(pattern, options)

Make a regular expression object from the pattern.

Options

All options are `false` by default.

debug

Dump a ton of stuff to stderr.

nobrace

Do not expand `{a,b}` and `{1..3}` brace sets.

noglobstar

Disable `**` matching against multiple folder names.

pattern into a case-insensitive regular expression, and instead leave them as strings.

This is the default when the platform is `win32` and `nocase:true` is set.

preserveMultipleSlashes

By default, multiple `/` characters (other than the leading `//` in a UNC path, see "UNC Paths" above) are treated as a single `/`.

That is, a pattern like `a///b` will match the file path `a/b`.

Set `preserveMultipleSlashes: true` to suppress this behavior.

optimizationLevel

A number indicating the level of optimization that should be done to the pattern prior to parsing and using it for matches.

Globstar parts `**` are always converted to `*` when `noglobstar` is set, and multiple adjacent `**` parts are converted into a single `**` (ie, `a/**/**/b` will be treated as `a/**/b`, as this is equivalent in all cases).

- `0` - Make no further changes. In this mode, `.` and `..` are maintained in the pattern, meaning that they must also appear in the same position in the test path string. Eg, a pattern like `a/*/../c` will match the string `a/b/../c` but not the string `a/c`.

```
minimatch('/a/b', '/a/*/c/d', {
        partial: true }) // true, might
        be /a/b/c/d
minimatch('/a/b', '/**/d', { partial:
        true }) // true, might be /a/
        b/.../d
minimatch('/x/y/z', '/a/**/z', {
        partial: true }) // false,
        because x !== a
```

windowsPathsNoEscape

Use `\\` as a path separator *only*, and *never* as an escape character. If set, all `\\` characters are replaced with `/` in the pattern. Note that this makes it **impossible** to match against paths containing literal glob pattern characters, but allows matching with patterns constructed using `path.join()` and `path.resolve()` on Windows platforms, mimicking the (buggy!) behavior of earlier versions on Windows. Please use with caution, and be mindful of the caveat about Windows paths.

For legacy reasons, this is also set if `options.allowWindowsEscape` is set to the exact value `false`.

windowsNoMagicRoot

When a pattern starts with a UNC path or drive letter, and in `nocase:true` mode, do not convert the root portions of the

dot

Allow patterns to match filenames starting with a period, even if the pattern does not explicitly have a period in that spot.

Note that by default, `a/**/b` will **not** match `a/.d/b`, unless `dot` is set.

noext

Disable “extglob” style patterns like `+(a|b)`.

nocase

Perform a case-insensitive match.

nocaseMagicOnly

When used with `{nocase: true}`, create regular expressions that are case-insensitive, but leave string match portions untouched. Has no effect when used without `{nocase: true}`.

Useful when some other form of case-insensitive matching is used, or if the original string representation is useful in some other way.

nonull

When a match is not found by `minimatch.match`, return a list containing the pattern itself if this option is set. When not set, an empty list is returned if there are no matches.

magicalBraces

This only affects the results of the `Minimatch.hasMagic` method.

If the pattern contains brace expansions, such as `a{b,c}d`, but no other magic characters, then the `Minimatch.hasMagic()` method will return `false` by default. When this option set, it will return `true` for brace expansion as well as other magic glob characters.

matchBase

If set, then patterns without slashes will be matched against the basename of the path if it contains slashes. For example, `a?b` would match the path `/xyz/123/acb`, but not `/xyz/acb/123`.

nocomment

Suppress the behavior of treating # at the start of a pattern as a comment.

nonegate

Suppress the behavior of treating a leading `!` character as negation.

flipNegate

Returns from negate expressions the same as if they were not negated. (Ie, true on a hit, false on a miss.)

partial

Compare a partial path to a pattern. As long as the parts of the path that are present are not contradicted by the pattern, it will be treated as a match. This is useful in applications where you're walking through a folder structure, and don't yet have the full path, but want to ensure that you do not walk down paths that can never be a match.

For example,