bmatcuk / **doublestar** Public

Implements support for double star (**) matches in golang's path.Match and filepath.Glob.

MIT license

373 stars **50** forks

Star Watch

Code Issues 2 Pull requests 1 Actions Projects Wiki Security Ins

master

bmatcuk ··· on Jan 5

View code

doublestar

Path pattern matching and globbing supporting `doublestar` (`**`) patterns.

About

Upgrading?

doublestar is a golang implementation of path pattern matching and globbing with support for "doublestar" (aka globstar: `**`) patterns.

doublestar patterns match files and directories recursively. For example, if you had the following directory structure:

```
grandparent
`-- parent
    |-- child1
    `-- child2
```

You could find the children with patterns such as: `**/child*` , `grandparent/**/child?` , `**/parent/*` , or even just `**` by itself (which will return all files and directories recursively).

Bash's globstar is doublestar's inspiration and, as such, works similarly. Note that the doublestar must appear as a path component by itself. A pattern such as `/path**` is invalid and will be treated the same as `/path*` , but `/path*/**` should achieve the desired result. Additionally, `/path/**` will match all directories and files under the path directory, but `/path/**/` will only match directories.

v4 is a complete rewrite with a focus on performance. Additionally, doublestar has been updated to use the new io/fs package for filesystem access. As a result, it is only supported by golang v1.16+.

Installation

doublestar can be installed via `go get`:

```
go get github.com/bmatcuk/doublestar/v4
```

To use it in your code, you must import it:

```
import "github.com/bmatcuk/doublestar/v4"
```

Usage

ErrBadPattern

```
doublestar.ErrBadPattern
```

Returned by various functions to report that the pattern is malformed. At the moment, this value is equal to `path.ErrBadPattern`, but, for portability, this equivalence should probably not be relied upon.

Match

```
func Match(pattern, name string) (bool, error)
```

Match returns true if `name` matches the file name `pattern` (see "patterns"). `name` and `pattern` are split on forward slash (`/`) characters and may be relative or absolute.

Match requires pattern to match all of name, not just a substring. The only possible returned error is `ErrBadPattern`, when pattern is malformed.

Note: this is meant as a drop-in replacement for `path.Match()` which always uses `'/'` as the path separator. If you want to support systems which use a different path separator (such as Windows), what you want is `PathMatch()`. Alternatively, you can run `filepath.ToSlash()` on both pattern and name and then use this function.

Note: users should *not* count on the returned error, `doublestar.ErrBadPattern`, being equal to `path.ErrBadPattern`.

PathMatch

```
func PathMatch(pattern, name string) (bool, error)
```

PathMatch returns true if `name` matches the file name `pattern` (see "patterns"). The difference between Match and PathMatch is that PathMatch will automatically use your system's path separator to split `name` and `pattern`. On systems where the path separator is `'\'`, escaping will be disabled.

Note: this is meant as a drop-in replacement for `filepath.Match()`. It assumes that both `pattern` and `name` are using the system's path separator. If you can't be sure of that, use `filepath.ToSlash()` on both `pattern` and `name`, and then use the `Match()` function instead.

GlobOption

Options that may be passed to `Glob`, `GlobWalk`, or `FilepathGlob`. Any number of options may be passed to these functions, and in any order, as the last argument(s).

```
WithFailOnIOErrors()
```

If passed, doublestar will abort and return IO errors when encountered. Note that if the glob pattern references a path that does not exist (such as `nonexistent/path/*`), this is *not* considered an IO error: it is considered a pattern with no matches.

```
WithFailOnPatternNotExist()
```

If passed, doublestar will abort and return `doublestar.ErrPatternNotExist` if the pattern references a path that does not exist before any meta characters such as `nonexistent/path/*`. Note that alts (ie, `{...}`) are expanded before this check. In other words, a pattern such as `{a,b}/*` may fail if either `a` or `b` do not exist but `*/{a,b}` will never fail because the star may match nothing.

```
WithFilesOnly()
```

If passed, doublestar will only return "files" from `Glob`, `GlobWalk`, or `FilepathGlob`. In this context, "files" are anything that is not a directory or a symlink to a directory.

Note: if combined with the WithNoFollow option, symlinks to directories *will* be included in the result since no attempt is made to follow the symlink.

```
WithNoFollow()
```

If passed, doublestar will not follow symlinks while traversing the filesystem. However, due to io/fs's *very* poor support for querying the filesystem about symlinks, there's a caveat here: if part of the pattern before any meta characters contains a reference to a symlink, it will be followed. For example, a pattern such as `path/to/symlink/*` will be followed assuming it is a valid symlink to a directory. However, from this same example, a pattern such as `path/to/**` will not traverse the `symlink`, nor would `path/*/symlink/*`

Note: if combined with the WithFilesOnly option, symlinks to directories *will* be included in the result since no attempt is made to follow the symlink.

Glob

```go
func Glob(fsys fs.FS, pattern string, opts ...GlobOption) ([]string, error)
```

Glob returns the names of all files matching pattern or nil if there is no matching file. The syntax of patterns is the same as in `Match()`. The pattern may describe hierarchical names such as `usr/*/bin/ed`.

Glob ignores file system errors such as I/O errors reading directories by default. The only possible returned error is `ErrBadPattern`, reporting that the pattern is malformed.

To enable aborting on I/O errors, the `WithFailOnIOErrors` option can be passed.

Note: this is meant as a drop-in replacement for `io/fs.Glob()`. Like `io/fs.Glob()`, this function assumes that your pattern uses `/` as the path separator even if that's not correct for your OS (like Windows). If you aren't sure if that's the case, you can use `filepath.ToSlash()` on your pattern before calling `Glob()`.

Like `io/fs.Glob()`, patterns containing `/./`, `/../`, or starting with `/` will return no results and no errors. This seems to be a conscious decision, even if counter-intuitive. You can use SplitPattern to divide a pattern into a base path (to initialize an `FS` object) and pattern.

Note: users should *not* count on the returned error, `doublestar.ErrBadPattern`, being equal to `path.ErrBadPattern`.

GlobWalk

```go
type GlobWalkFunc func(path string, d fs.DirEntry) error

func GlobWalk(fsys fs.FS, pattern string, fn GlobWalkFunc, opts ...GlobOption) error
```

GlobWalk calls the callback function `fn` for every file matching pattern. The syntax of pattern is the same as in Match() and the behavior is the same as Glob(), with regard to limitations (such as patterns containing `/./`, `/../`, or starting with `/`). The pattern may describe hierarchical names such as usr/*/bin/ed.

GlobWalk may have a small performance benefit over Glob if you do not need a slice of matches because it can avoid allocating memory for the matches. Additionally, GlobWalk gives you access to the `fs.DirEntry` objects for each match, and lets you quit early by returning a non-nil error from your callback function. Like `io/fs.WalkDir`, if your callback returns `SkipDir`, GlobWalk will skip the current directory. This means that if the current path *is* a directory, GlobWalk will not recurse into it. If the current path is not a directory, the rest of the parent directory will be skipped.

GlobWalk ignores file system errors such as I/O errors reading directories by default. GlobWalk may return `ErrBadPattern`, reporting that the pattern is malformed.

To enable aborting on I/O errors, the `WithFailOnIOErrors` option can be passed.

Additionally, if the callback function `fn` returns an error, GlobWalk will exit immediately and return that error.

Like Glob(), this function assumes that your pattern uses `/` as the path separator even if that's not correct for your OS (like Windows). If you aren't sure if that's the case, you can use filepath.ToSlash() on your pattern before calling GlobWalk().

Note: users should *not* count on the returned error, `doublestar.ErrBadPattern`, being equal to `path.ErrBadPattern`.

FilepathGlob

```
func FilepathGlob(pattern string, opts ...GlobOption) (matches []string, err error)
```

FilepathGlob returns the names of all files matching pattern or nil if there is no matching file. The syntax of pattern is the same as in Match(). The pattern may describe hierarchical names such as usr/*/bin/ed.

FilepathGlob ignores file system errors such as I/O errors reading directories by default. The only possible returned error is `ErrBadPattern`, reporting that the pattern is malformed.

To enable aborting on I/O errors, the `WithFailOnIOErrors` option can be passed.

Note: FilepathGlob is a convenience function that is meant as a drop-in replacement for `path/filepath.Glob()` for users who don't need the complication of io/fs. Basically, it:

- Runs `filepath.Clean()` and `ToSlash()` on the pattern
- Runs `SplitPattern()` to get a base path and a pattern to Glob
- Creates an FS object from the base path and `Glob()s` on the pattern
- Joins the base path with all of the matches from `Glob()`

Returned paths will use the system's path separator, just like `filepath.Glob()`.

Note: the returned error `doublestar.ErrBadPattern` is not equal to `filepath.ErrBadPattern`.

SplitPattern

```
func SplitPattern(p string) (base, pattern string)
```

SplitPattern is a utility function. Given a pattern, SplitPattern will return two strings: the first string is everything up to the last slash (`/`) that appears *before* any unescaped "meta" characters (ie, `*?[{`). The second string is everything after that slash. For example, given the pattern:

```
../../path/to/meta*/**
             ^----------- split here
```

SplitPattern returns "../../path/to" and "meta*/**". This is useful for initializing os.DirFS() to call Glob() because Glob() will silently fail if your pattern includes `/./` or `/../`. For example:

```
base, pattern := SplitPattern("../../path/to/meta*/**")
fsys := os.DirFS(base)
matches, err := Glob(fsys, pattern)
```

If SplitPattern cannot find somewhere to split the pattern (for example, `meta*/**`), it will return "." and the unaltered pattern (`meta*/**` in this example).

Of course, it is your responsibility to decide if the returned base path is "safe" in the context of your application. Perhaps you could use Match() to validate against a list of approved base directories?

ValidatePattern

```
func ValidatePattern(s string) bool
```

Validate a pattern. Patterns are validated while they run in Match(), PathMatch(), and Glob(), so, you normally wouldn't need to call this. However, there are cases where this might be useful: for example, if your program allows a user to enter a pattern that you'll run at a later time, you might want to validate it.

ValidatePattern assumes your pattern uses '/' as the path separator.

ValidatePathPattern

```
func ValidatePathPattern(s string) bool
```

Like ValidatePattern, only uses your OS path separator. In other words, use ValidatePattern if you would normally use Match() or Glob(). Use ValidatePathPattern if you would normally use PathMatch(). Keep in mind, Glob() requires '/' separators, even if your OS uses something else.

Patterns

doublestar supports the following special terms in the patterns:

Special Terms	Meaning
`*`	matches any sequence of non-path-separators
`/**/`	matches zero or more directories
`?`	matches any single non-path-separator character
`[class]`	matches any single non-path-separator character against a class of characters (see "character classes")
`{alt1,...}`	matches a sequence of characters if one of the comma-separated alternatives matches

Any character with a special meaning can be escaped with a backslash (`\`).

A doublestar (`**`) should appear surrounded by path separators such as `/**/` . A mid-pattern doublestar (`**`) behaves like bash's globstar option: a pattern such as `path/to/**.txt` would return the same results as `path/to/*.txt` . The pattern you're looking for is `path/to/**/*.txt` .

Character Classes

Character classes support the following:

Class	Meaning
`[abc]`	matches any single character within the set
`[a-z]`	matches any single character in the range
`[^class]`	matches any single character which does *not* match the class
`[!class]`	same as `^` : negates the class

Performance

```
goos: darwin
goarch: amd64
pkg: github.com/bmatcuk/doublestar/v4
cpu: Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz
BenchmarkMatch-8                  285639              3868 ns/op               0 B/op
0 allocs/op
BenchmarkGoMatch-8                286945              3726 ns/op               0 B/op
0 allocs/op
BenchmarkPathMatch-8              320511              3493 ns/op               0 B/op
0 allocs/op
BenchmarkGoPathMatch-8            304236              3434 ns/op               0 B/op
```

README.md

```
2849 allocs/op
BenchmarkGlobWalk-8                  476           2536293 ns/op          184017 B/op
2750 allocs/op
BenchmarkGoGlob-8                    463           2574836 ns/op          194249 B/op
2929 allocs/op
```

These benchmarks (in `doublestar_test.go`) compare Match() to path.Match(), PathMath() to filepath.Match(), and Glob() + GlobWalk() to io/fs.Glob(). They only run patterns that the standard go packages can understand as well (so, no `{alts}` or `**`) for a fair comparison. Of course, alts and doublestars will be less performant than the other pattern meta characters.

Alts are essentially like running multiple patterns, the number of which can get large if your pattern has alts nested inside alts. This affects both matching (ie, Match()) and globbing (Glob()).

`**` performance in matching is actually pretty similar to a regular `*` , but can cause a large number of reads when globbing as it will need to recursively traverse your filesystem.

Sponsors

I started this project in 2014 in my spare time and have been maintaining it ever since. In that time, it has grown into one of the most popular globbing libraries in the Go ecosystem. So, if **doublestar** is a useful library in your project, consider sponsoring my work! I'd really appreciate it!

Thanks for sponsoring me!

License

MIT License

Releases 44

Added WithNoFollow Latest
on Jan 5

+ 43 releases

Sponsor this project

bmatcuk Bob Matcuk

Sponsor

Learn more about GitHub Sponsors

Packages

No packages published

Used by 3.1k

+ 3,117

Contributors 10

Languages

- **Go** 100.0%