godirwalk

`godirwalk` is a library for traversing a directory tree on a file system.

In short, why do I use this library?

1. It's faster than `filepath.Walk`.
2. It's more correct on Windows than `filepath.Walk`.
3. It's more easy to use than `filepath.Walk`.
4. It's more flexible than `filepath.Walk`.

Usage Example

Additional examples are provided in the `examples/` subdirectory.

This library will normalize the provided top level directory name based on the os-specific path separator by calling `filepath.Clean` on its first argument. However it always provides the pathname created by using the correct os-specific path separator when invoking the provided callback function.

```
    dirname := "some/directory/root"
    err := godirwalk.Walk(dirname, &godirwalk.Options{
        Callback: func(osPathname string, de *godirwalk.Dirent) error {
            // Following string operation is not most performant way
            // of doing this, but common enough to warrant a simple
            // example here:
            if strings.Contains(osPathname, ".git") {
                return godirwalk.SkipThis
            }
            fmt.Printf("%s %s\n", de.ModeType(), osPathname)
            return nil
        },
        Unsorted: true, // (optional) set true for faster yet non-deterministic
enumeration (see godoc)
    })
```

This library not only provides functions for traversing a file system directory tree, but also for obtaining a list of immediate descendants of a particular directory, typically much more quickly than using `os.ReadDir` or `os.ReadDirnames`.

Description

Here's why I use `godirwalk` in preference to `filepath.Walk`, `os.ReadDir`, and `os.ReadDirnames`.

It's faster than `filepath.Walk`

When compared against `filepath.Walk` in benchmarks, it has been observed to run between five and ten times the speed on darwin, at speeds comparable to the that of the unix `find` utility; and about twice the speed on linux;

and about four times the speed on Windows.

How does it obtain this performance boost? It does less work to give you nearly the same output. This library calls the same `syscall` functions to do the work, but it makes fewer calls, does not throw away information that it might need, and creates less memory churn along the way by reusing the same scratch buffer for reading from a directory rather than reallocating a new buffer every time it reads file system entry data from the operating system.

While traversing a file system directory tree, `filepath.Walk` obtains the list of immediate descendants of a directory, and throws away the node type information for the file system entry that is provided by the operating system that comes with the node's name. Then, immediately prior to invoking the callback function, `filepath.Walk` invokes `os.Stat` for each node, and passes the returned `os.FileInfo` information to the callback.

While the `os.FileInfo` information provided by `os.Stat` is extremely helpful--and even includes the `os.FileMode` data--providing it requires an additional system call for each node.

Because most callbacks only care about what the node type is, this library does not throw the type information away, but rather provides that information to the callback function in the form of a `os.FileMode` value. Note that the provided `os.FileMode` value that this library provides only has the node type information, and does not have the permission bits, sticky bits, or other information from the file's mode. If the callback does care about a particular node's entire `os.FileInfo` data structure, the callback can easiy invoke `os.Stat` when needed, and only when needed.

Benchmarks

macOS

```
$ go test -bench=. -benchmem
goos: darwin
goarch: amd64
pkg: github.com/karrick/godirwalk
BenchmarkReadDirnamesStandardLibrary-12   50000       26250  ns/op       10360  B/op
16  allocs/op
BenchmarkReadDirnamesThisLibrary-12       50000       24372  ns/op        5064  B/op
20  allocs/op
BenchmarkFilepathWalk-12                      1  1099524875  ns/op   228415912  B/op
416952  allocs/op
BenchmarkGodirwalk-12                         2   526754589  ns/op   103110464  B/op
451442  allocs/op
BenchmarkGodirwalkUnsorted-12                 3   509219296  ns/op   100751400  B/op
378800  allocs/op
BenchmarkFlameGraphFilepathWalk-12            1  7478618820  ns/op  2284138176  B/op
4169453  allocs/op
BenchmarkFlameGraphGodirwalk-12               1  4977264058  ns/op  1031105328  B/op
4514423  allocs/op
PASS
ok      github.com/karrick/godirwalk    21.219s
```

Linux

```
$ go test -bench=. -benchmem
goos: linux
```

```
goarch: amd64
pkg: github.com/karrick/godirwalk
BenchmarkReadDirnamesStandardLibrary-12  100000       15458  ns/op       10360  B/op
16  allocs/op
BenchmarkReadDirnamesThisLibrary-12      100000       14646  ns/op        5064  B/op
20  allocs/op
BenchmarkFilepathWalk-12                      2   631034745  ns/op   228210216  B/op
416939  allocs/op
BenchmarkGodirwalk-12                         3   358714883  ns/op   102988664  B/op
451437  allocs/op
BenchmarkGodirwalkUnsorted-12                 3   355363915  ns/op   100629234  B/op
378796  allocs/op
BenchmarkFlameGraphFilepathWalk-12            1  6086913991  ns/op  2282104720  B/op
4169417  allocs/op
BenchmarkFlameGraphGodirwalk-12               1  3456398824  ns/op  1029886400  B/op
4514373  allocs/op
PASS
ok      github.com/karrick/godirwalk    19.179s
```

It's more correct on Windows than `filepath.Walk`

I did not previously care about this either, but humor me. We all love how we can write once and run everywhere. It is essential for the language's adoption, growth, and success, that the software we create can run unmodified on all architectures and operating systems supported by Go.

When the traversed file system has a logical loop caused by symbolic links to directories, on unix `filepath.Walk` ignores symbolic links and traverses the entire directory tree without error. On Windows however, `filepath.Walk` will continue following directory symbolic links, even though it is not supposed to, eventually causing `filepath.Walk` to terminate early and return an error when the pathname gets too long from concatenating endless loops of symbolic links onto the pathname. This error comes from Windows, passes through `filepath.Walk`, and to the upstream client running `filepath.Walk`.

The takeaway is that behavior is different based on which platform `filepath.Walk` is running. While this is clearly not intentional, until it is fixed in the standard library, it presents a compatibility problem.

This library fixes the above problem such that it will never follow logical file sytem loops on either unix or Windows. Furthermore, it will only follow symbolic links when `FollowSymbolicLinks` is set to true. Behavior on Windows and other operating systems is identical.

It's more easy to use than `filepath.Walk`

While this library strives to mimic the behavior of the incredibly well-written `filepath.Walk` standard library, there are places where it deviates a bit in order to provide a more easy or intuitive caller interface.

Callback interface does not send you an error to check

Since this library does not invoke `os.Stat` on every file system node it encounters, there is no possible error event for the callback function to filter on. The third argument in the `filepath.WalkFunc` function signature to pass the error from `os.Stat` to the callback function is no longer necessary, and thus eliminated from signature of the callback function from this library.

Furthermore, this slight interface difference between `filepath.WalkFunc` and this library's `WalkFunc` eliminates the boilerplate code that callback handlers must write when they use `filepath.Walk`. Rather than every callback function needing to check the error value passed into it and branch accordingly, users of this library do not even have an error value to check immediately upon entry into the callback function. This is an improvement both in runtime performance and code clarity.

Callback function is invoked with OS specific file system path separator

On every OS platform `filepath.Walk` invokes the callback function with a solidus (`/`) delimited pathname. By contrast this library invokes the callback with the os-specific pathname separator, obviating a call to `filepath.Clean` in the callback function for each node prior to actually using the provided pathname.

In other words, even on Windows, `filepath.Walk` will invoke the callback with `some/path/to/foo.txt`, requiring well written clients to perform pathname normalization for every file prior to working with the specified file. This is a hidden boilerplate requirement to create truly os agnostic callback functions. In truth, many clients developed on unix and not tested on Windows neglect this subtlety, and will result in software bugs when someone tries to run that software on Windows.

This library invokes the callback function with `some\path\to\foo.txt` for the same file when running on Windows, eliminating the need to normalize the pathname by the client, and lessen the likelyhood that a client will work on unix but not on Windows.

This enhancement eliminates necessity for some more boilerplate code in callback functions while improving the runtime performance of this library.

`godirwalk.SkipThis` is more intuitive to use than `filepath.SkipDir`

One arguably confusing aspect of the `filepath.WalkFunc` interface that this library must emulate is how a caller tells the `Walk` function to skip file system entries. With both `filepath.Walk` and this library's `Walk`, when a callback function wants to skip a directory and not descend into its children, it returns `filepath.SkipDir`. If the callback function returns `filepath.SkipDir` for a non-directory, `filepath.Walk` and this library will stop processing any more entries in the current directory. This is not necessarily what most developers want or expect. If you want to simply skip a particular non-directory entry but continue processing entries in the directory, the callback function must return nil.

The implications of this interface design is when you want to walk a file system hierarchy and skip an entry, you have to return a different value based on what type of file system entry that node is. To skip an entry, if the entry is a directory, you must return `filepath.SkipDir`, and if entry is not a directory, you must return `nil`. This is an unfortunate hurdle I have observed many developers struggling with, simply because it is not an intuitive interface.

Here is an example callback function that adheres to `filepath.WalkFunc` interface to have it skip any file system entry whose full pathname includes a particular substring, `optSkip`. Note that this library still supports identical behavior of `filepath.Walk` when the callback function returns `filepath.SkipDir`.

```
    func callback1(osPathname string, de *godirwalk.Dirent) error {
        if optSkip != "" && strings.Contains(osPathname, optSkip) {
            if b, err := de.IsDirOrSymlinkToDir(); b == true && err == nil {
                return filepath.SkipDir
            }
            return nil
        }
        // Process file like normal...
```

```
        return nil
    }
```

This library attempts to eliminate some of that logic boilerplate required in callback functions by providing a new token error value, `SkipThis`, which a callback function may return to skip the current file system entry regardless of what type of entry it is. If the current entry is a directory, its children will not be enumerated, exactly as if the callback had returned `filepath.SkipDir`. If the current entry is a non-directory, the next file system entry in the current directory will be enumerated, exactly as if the callback returned `nil`. The following example callback function has identical behavior as the previous, but has less boilerplate, and admittedly logic that I find more simple to follow.

```
    func callback2(osPathname string, de *godirwalk.Dirent) error {
        if optSkip != "" && strings.Contains(osPathname, optSkip) {
            return godirwalk.SkipThis
        }
        // Process file like normal...
        return nil
    }
```

It's more flexible than `filepath.Walk`

Configurable Handling of Symbolic Links

The default behavior of this library is to ignore symbolic links to directories when walking a directory tree, just like `filepath.Walk` does. However, it does invoke the callback function with each node it finds, including symbolic links. If a particular use case exists to follow symbolic links when traversing a directory tree, this library can be invoked in manner to do so, by setting the `FollowSymbolicLinks` config parameter to `true`.

Configurable Sorting of Directory Children

The default behavior of this library is to always sort the immediate descendants of a directory prior to visiting each node, just like `filepath.Walk` does. This is usually the desired behavior. However, this does come at slight performance and memory penalties required to sort the names when a directory node has many entries. Additionally if caller specifies `Unsorted` enumeration in the configuration parameter, reading directories is lazily performed as the caller consumes entries. If a particular use case exists that does not require sorting the directory's immediate descendants prior to visiting its nodes, this library will skip the sorting step when the `Unsorted` parameter is set to `true`.

Here's an interesting read of the potential hazzards of traversing a file system hierarchy in a non-deterministic order. If you know the problem you are solving is not affected by the order files are visited, then I encourage you to use `Unsorted`. Otherwise skip setting this option.

Researchers find bug in Python script may have affected hundreds of studies

Configurable Post Children Callback

This library provides upstream code with the ability to specify a callback function to be invoked for each directory after its children are processed. This has been used to recursively delete empty directories after traversing the file system in a more efficient manner. See the `examples/clean-empties` directory for an example of this usage.

Configurable Error Callback

This library provides upstream code with the ability to specify a callback to be invoked for errors that the operating system returns, allowing the upstream code to determine the next course of action to take, whether to halt walking the hierarchy, as it would do were no error callback provided, or skip the node that caused the error. See the `examples/walk-fast` directory for an example of this usage.