NAME

File::Find - Traverse a directory tree.

SYNOPSIS

```
use File::Find;
find(\&wanted, @directories_to_search);
sub wanted { ... }

use File::Find;
finddepth(\&wanted, @directories_to_search);
sub wanted { ... }

use File::Find;
find({ wanted => \&process, follow => 1 }, '.');
```

DESCRIPTION

These are functions for searching through directory trees doing work on each file found similar to the Unix *find* command. File::Find exports two functions, `find` and `finddepth`. They work similarly but have subtle differences.

find

```
find(\&wanted,  @directories);
find(\%options, @directories);
```

find() does a depth-first search over the given @directories in the order they are given. For each file or directory found, it calls the &wanted subroutine. (See below for details on how to use the &wanted function). Additionally, for each directory found, it will chdir() into that directory and continue the search, invoking the &wanted function on each file or subdirectory in the directory.

finddepth

```
finddepth(\&wanted,  @directories);
finddepth(\%options, @directories);
```

finddepth() works just like find() except that it invokes the &wanted function for a directory *after* invoking it for the directory's contents. It does a postorder traversal instead of a preorder traversal, working from the bottom of the directory tree up where find() works from the top of the tree down.

%options

The first argument to find() is either a code reference to your &wanted function, or a hash reference describing the operations to be performed for each file. The code reference is described in *The wanted function* below.

Here are the possible keys for the hash:

wanted

The value should be a code reference. This code reference is described in *The wanted function* below. The &wanted subroutine is mandatory.

bydepth

Reports the name of a directory only AFTER all its entries have been reported. Entry point finddepth() is a shortcut for specifying { bydepth => 1 } in the first argument of find() .

---

preprocess

> The value should be a code reference. This code reference is used to preprocess the current directory. The name of the currently processed directory is in $File::Find::dir. Your preprocessing function is called after readdir(), but before the loop that calls the wanted() function. It is called with a list of strings (actually file/directory names) and is expected to return a list of strings. The code can be used to sort the file/directory names alphabetically, numerically, or to filter out directory entries based on their name alone. When *follow* or *follow_fast* are in effect, preprocess is a no-op.

postprocess

> The value should be a code reference. It is invoked just before leaving the currently processed directory. It is called in void context with no arguments. The name of the current directory is in $File::Find::dir. This hook is handy for summarizing a directory, such as calculating its disk usage. When *follow* or *follow_fast* are in effect, postprocess is a no-op.

follow

> Causes symbolic links to be followed. Since directory trees with symbolic links (followed) may contain files more than once and may even have cycles, a hash has to be built up with an entry for each file. This might be expensive both in space and time for a large directory tree. See *follow_fast* and *follow_skip* below. If either *follow* or *follow_fast* is in effect:
>
> - It is guaranteed that an *lstat* has been called before the user's wanted() function is called. This enables fast file checks involving _. Note that this guarantee no longer holds if *follow* or *follow_fast* are not set.
>
> - There is a variable $File::Find::fullname which holds the absolute pathname of the file with all symbolic links resolved. If the link is a dangling symbolic link, then fullname will be set to undef.
>
> This is a no-op on Win32.

follow_fast

> This is similar to *follow* except that it may report some files more than once. It does detect cycles, however. Since only symbolic links have to be hashed, this is much cheaper both in space and time. If processing a file more than once (by the user's wanted() function) is worse than just taking time, the option *follow* should be used.
>
> This is also a no-op on Win32.

follow_skip

> follow_skip==1, which is the default, causes all files which are neither directories nor symbolic links to be ignored if they are about to be processed a second time. If a directory or a symbolic link are about to be processed a second time, File::Find dies.
>
> follow_skip==0 causes File::Find to die if any file is about to be processed a second time.
>
> follow_skip==2 causes File::Find to ignore any duplicate files and directories but to proceed normally otherwise.

dangling_symlinks

> If true and a code reference, will be called with the symbolic link name and the directory it lives in as arguments. Otherwise, if true and warnings are on, warning "symbolic_link_name is a dangling symbolic link\n" will be issued. If false, the dangling symbolic link will be silently ignored.

no_chdir

> Does not chdir() to each directory as it recurses. The wanted() function will need to be aware of this, of course. In this case, $_ will be the same as $File::Find::name.

---

untaint

> If find is used in taint-mode (-T command line switch or if EUID != UID or if EGID != GID) then internally directory names have to be untainted before they can be chdir'ed to. Therefore they are checked against a regular expression *untaint_pattern*. Note that all names passed to the user's *wanted()* function are still tainted. If this option is used while not in taint-mode, untaint is a no-op.

untaint_pattern

> See above. This should be set using the qr quoting operator. The default is set to qr|^([-+@\w./]+)$|. Note that the parentheses are vital.

untaint_skip

> If set, a directory which fails the *untaint_pattern* is skipped, including all its sub-directories. The default is to 'die' in such a case.

The wanted function

The wanted() function does whatever verifications you want on each file and directory. Note that despite its name, the wanted() function is a generic callback function, and does **not** tell File::Find if a file is "wanted" or not. In fact, its return value is ignored.

The wanted function takes no arguments but rather does its work through a collection of variables.

$File::Find::dir is the current directory name,

$_ is the current filename within that directory

$File::Find::name is the complete pathname to the file.

The above variables have all been localized and may be changed without affecting data outside of the wanted function.

For example, when examining the file */some/path/foo.ext* you will have:

```
$File::Find::dir  = /some/path/
$_                = foo.ext
$File::Find::name = /some/path/foo.ext
```

You are chdir()'d to $File::Find::dir when the function is called, unless no_chdir was specified. Note that when changing to directories is in effect the root directory (/) is a somewhat special case inasmuch as the concatenation of $File::Find::dir, '/' and $_ is not literally equal to $File::Find::name. The table below summarizes all variants:

```
            $File::Find::name  $File::Find::dir  $_
 default     /                  /                 .
 no_chdir=>0 /etc               /                 etc
             /etc/x             /etc              x

 no_chdir=>1 /                  /                 /
             /etc               /                 /etc
             /etc/x             /etc              /etc/x
```

When follow or follow_fast are in effect, there is also a $File::Find::fullname. The function may set $File::Find::prune to prune the tree unless bydepth was specified. Unless follow or follow_fast is specified, for compatibility reasons (find.pl, find2perl) there are in addition the following globals available: $File::Find::topdir, $File::Find::topdev, $File::Find::topino, $File::Find::topmode and $File::Find::topnlink.

This library is useful for the find2perl tool, which when fed,

```
find2perl / -name .nfs\* -mtime +7 \
    -exec rm -f {} \; -o -fstype nfs -prune
```

produces something like:

```
sub wanted {
    /^\.nfs.*\z/s &&
    (($dev, $ino, $mode, $nlink, $uid, $gid) = lstat($_)) &&
    int(-M _) > 7 &&
    unlink($_)
    ||
    ($nlink || (($dev, $ino, $mode, $nlink, $uid, $gid) = lstat($_)))
&&
    $dev < 0 &&
    ($File::Find::prune = 1);
}
```

Notice the _ in the above int(-M _): the _ is a magical filehandle that caches the information from the preceding stat(), lstat(), or filetest.

Here's another interesting wanted function. It will find all symbolic links that don't resolve:

```
sub wanted {
    -l && !-e && print "bogus link: $File::Find::name\n";
}
```

Note that you may mix directories and (non-directory) files in the list of directories to be searched by the wanted() function.

```
find(\&wanted, "./foo", "./bar", "./baz/epsilon");
```

In the example above, no file in *./baz/* other than *./baz/epsilon* will be evaluated by wanted().

See also the script pfind on CPAN for a nice application of this module.

WARNINGS

If you run your program with the -w switch, or if you use the warnings pragma, File::Find will report warnings for several weird situations. You can disable these warnings by putting the statement

```
no warnings 'File::Find';
```

in the appropriate scope. See *perllexwarn* for more info about lexical warnings.

CAVEAT

$dont_use_nlink

You can set the variable $File::Find::dont_use_nlink to 1, if you want to force File::Find to always stat directories. This was used for file systems that do not have an nlink count matching the number of sub-directories. Examples are ISO-9660 (CD-ROM), AFS, HPFS (OS/2 file system), FAT (DOS file system) and a couple of others.

You shouldn't need to set this variable, since File::Find should now detect such file systems on-the-fly and switch itself to using stat. This works even for parts of your file system, like a mounted CD-ROM.

If you do set $File::Find::dont_use_nlink to 1, you will notice slow-downs.

symlinks

Be aware that the option to follow symbolic links can be dangerous. Depending on the structure of the directory tree (including symbolic links to directories) you might traverse a given (physical) directory more than once (only if `follow_fast` is in effect). Furthermore, deleting or changing files in a symbolically linked directory might cause very unpleasant surprises, since you delete or change files in an unknown directory.

BUGS AND CAVEATS

Despite the name of the `finddepth()` function, both `find()` and `finddepth()` perform a depth-first search of the directory hierarchy.

HISTORY

File::Find used to produce incorrect results if called recursively. During the development of perl 5.8 this bug was fixed. The first fixed version of File::Find was 1.01.

SEE ALSO

find, find2perl.