NAME

File::Path - Create or remove directory trees

VERSION

This document describes version 2.09 of File::Path, released 2013-01-17.

SYNOPSIS

```
use File::Path qw(make_path remove_tree);

make_path('foo/bar/baz', '/zug/zwang');
make_path('foo/bar/baz', '/zug/zwang', {
    verbose => 1,
    mode => 0711,
});

remove_tree('foo/bar/baz', '/zug/zwang');
remove_tree('foo/bar/baz', '/zug/zwang', {
    verbose => 1,
    error  => \my $err_list,
});

# legacy (interface promoted before v2.00)
mkpath('/foo/bar/baz');
mkpath('/foo/bar/baz', 1, 0711);
mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);
rmtree('foo/bar/baz', 1, 1);
rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);

# legacy (interface promoted before v2.06)
mkpath('foo/bar/baz', '/zug/zwang', { verbose => 1, mode => 0711 });
rmtree('foo/bar/baz', '/zug/zwang', { verbose => 1, mode => 0711 });
```

DESCRIPTION

This module provide a convenient way to create directories of arbitrary depth and to delete an entire directory subtree from the filesystem.

The following functions are provided:

make_path($dir1, $dir2, ...)

make_path($dir1, $dir2, ..., \%opts)

The `make_path` function creates the given directories if they don't exists before, much like the Unix command `mkdir -p`.

The function accepts a list of directories to be created. Its behaviour may be tuned by an optional hashref appearing as the last parameter on the call.

The function returns the list of directories actually created during the call; in scalar context the number of directories created.

The following keys are recognised in the option hash:

mode => $num

The numeric permissions mode to apply to each created directory (defaults to 0777), to be modified by the current `umask`. If the directory already exists (and thus does not need to be created), the permissions will not be modified.

`mask` is recognised as an alias for this parameter.

verbose => $bool

If present, will cause `make_path` to print the name of each directory as it is created. By default nothing is printed.

error => \$err

If present, it should be a reference to a scalar. This scalar will be made to reference an array, which will be used to store any errors that are encountered. See the *ERROR HANDLING* section for more information.

If this parameter is not used, certain error conditions may raise a fatal error that will cause the program will halt, unless trapped in an `eval` block.

owner => $owner

user => $owner

uid => $owner

If present, will cause any created directory to be owned by `$owner`. If the value is numeric, it will be interpreted as a uid, otherwise as username is assumed. An error will be issued if the username cannot be mapped to a uid, or the uid does not exist, or the process lacks the privileges to change ownership.

Ownwership of directories that already exist will not be changed.

`user` and `uid` are aliases of `owner`.

group => $group

If present, will cause any created directory to be owned by the group `$group`. If the value is numeric, it will be interpreted as a gid, otherwise as group name is assumed. An error will be issued if the group name cannot be mapped to a gid, or the gid does not exist, or the process lacks the privileges to change group ownership.

Group ownwership of directories that already exist will not be changed.

```
    make_path '/var/tmp/webcache', {owner=>'nobody',
group=>'nogroup'};
```

mkpath($dir)

mkpath($dir, $verbose, $mode)

mkpath([$dir1, $dir2,...], $verbose, $mode)

mkpath($dir1, $dir2,..., \%opt)

The mkpath() function provide the legacy interface of make_path() with a different interpretation of the arguments passed. The behaviour and return value of the function is otherwise identical to make_path().

remove_tree($dir1, $dir2, ...)

remove_tree($dir1, $dir2, ..., \%opts)

The `remove_tree` function deletes the given directories and any files and subdirectories they might contain, much like the Unix command `rm -r` or `del /s` on Windows.

The function accepts a list of directories to be removed. Its behaviour may be tuned by an optional hashref appearing as the last parameter on the call.

The functions returns the number of files successfully deleted.

The following keys are recognised in the option hash:

verbose => $bool

If present, will cause `remove_tree` to print the name of each file as it is unlinked. By default nothing is printed.

safe => $bool

When set to a true value, will cause `remove_tree` to skip the files for which the process lacks the required privileges needed to delete files, such as delete privileges on VMS. In other words, the code will make no attempt to alter file permissions. Thus, if the process is interrupted, no filesystem object will be left in a more permissive mode.

keep_root => $bool

When set to a true value, will cause all files and subdirectories to be removed, except the initially specified directories. This comes in handy when cleaning out an application's scratch directory.

```
remove_tree( '/tmp', {keep_root => 1} );
```

result => \$res

If present, it should be a reference to a scalar. This scalar will be made to reference an array, which will be used to store all files and directories unlinked during the call. If nothing is unlinked, the array will be empty.

```
remove_tree( '/tmp', {result => \my $list} );
print "unlinked $_\n" for @$list;
```

This is a useful alternative to the `verbose` key.

error => \$err

If present, it should be a reference to a scalar. This scalar will be made to reference an array, which will be used to store any errors that are encountered. See the *ERROR HANDLING* section for more information.

Removing things is a much more dangerous proposition than creating things. As such, there are certain conditions that `remove_tree` may encounter that are so dangerous that the only sane action left is to kill the program.

Use `error` to trap all that is reasonable (problems with permissions and the like), and let it die if things get out of hand. This is the safest course of action.

rmtree($dir)

rmtree($dir, $verbose, $safe)

rmtree([$dir1, $dir2,...], $verbose, $safe)

rmtree($dir1, $dir2,..., \%opt)

The rmtree() function provide the legacy interface of remove_tree() with a different interpretation of the arguments passed. The behaviour and return value of the function is otherwise identical to remove_tree().

ERROR HANDLING

NOTE:

The following error handling mechanism is considered experimental and is subject to change pending feedback from users.

If `make_path` or `remove_tree` encounter an error, a diagnostic message will be printed to `STDERR` via `carp` (for non-fatal errors), or via `croak` (for fatal errors).

If this behaviour is not desirable, the `error` attribute may be used to hold a reference to a variable, which will be used to store the diagnostics. The variable is made a reference to an array of hash references. Each hash contain a single key/value pair where the key is the name of the file, and the value is the error message (including the contents of `$!` when appropriate). If a general error is encountered the diagnostic key will be empty.

An example usage looks like:

```
remove_tree( 'foo/bar', 'bar/rat', {error => \my $err} );
if (@$err) {
    for my $diag (@$err) {
        my ($file, $message) = %$diag;
        if ($file eq '') {
            print "general error: $message\n";
        }
        else {
            print "problem unlinking $file: $message\n";
        }
    }
}
else {
    print "No error encountered\n";
}
```

Note that if no errors are encountered, `$err` will reference an empty array. This means that `$err` will always end up TRUE; so you need to test `@$err` to determine if errors occured.

NOTES

`File::Path` blindly exports `mkpath` and `rmtree` into the current namespace. These days, this is considered bad style, but to change it now would break too much code. Nonetheless, you are invited to specify what it is you are expecting to use:

```
use File::Path 'rmtree';
```

The routines `make_path` and `remove_tree` are **not** exported by default. You must specify which ones you want to use.

```
use File::Path 'remove_tree';
```

Note that a side-effect of the above is that `mkpath` and `rmtree` are no longer exported at all. This is due to the way the `Exporter` module works. If you are migrating a codebase to use the new interface, you will have to list everything explicitly. But that's just good practice anyway.

```
use File::Path qw(remove_tree rmtree);
```

API CHANGES

The API was changed in the 2.0 branch. For a time, `mkpath` and `rmtree` tried, unsuccessfully, to deal with the two different calling mechanisms. This approach was considered a failure.

The new semantics are now only available with `make_path` and `remove_tree`. The old semantics are only available through `mkpath` and `rmtree`. Users are strongly encouraged to upgrade to at least 2.08 in order to avoid surprises.

SECURITY CONSIDERATIONS

There were race conditions 1.x implementations of File::Path's `rmtree` function (although sometimes patched depending on the OS distribution or platform). The 2.0 version contains code to avoid the problem mentioned in CVE-2002-0435.

See the following pages for more information:

```
http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=286905
http://www.nntp.perl.org/group/perl.perl5.porters/2005/01/msg97623.html
http://www.debian.org/security/2005/dsa-696
```

Additionally, unless the `safe` parameter is set (or the third parameter in the traditional interface is TRUE), should a `remove_tree` be interrupted, files that were originally in read-only mode may now have their permissions set to a read-write (or "delete OK") mode.

DIAGNOSTICS

FATAL errors will cause the program to halt (`croak`), since the problem is so severe that it would be dangerous to continue. (This can always be trapped with `eval`, but it's not a good idea. Under the circumstances, dying is the best thing to do).

SEVERE errors may be trapped using the modern interface. If the they are not trapped, or the old interface is used, such an error will cause the program will halt.

All other errors may be trapped using the modern interface, otherwise they will be `carp`ed about. Program execution will not be halted.

mkdir [path]: [errmsg] (SEVERE)

> `make_path` was unable to create the path. Probably some sort of permissions error at the point of departure, or insufficient resources (such as free inodes on Unix).

No root path(s) specified

> `make_path` was not given any paths to create. This message is only emitted if the routine is called with the traditional interface. The modern interface will remain silent if given nothing to do.

No such file or directory

> On Windows, if `make_path` gives you this warning, it may mean that you have exceeded your filesystem's maximum path length.

cannot fetch initial working directory: [errmsg]

> `remove_tree` attempted to determine the initial directory by calling `Cwd::getcwd`, but the call failed for some reason. No attempt will be made to delete anything.

cannot stat initial working directory: [errmsg]

> `remove_tree` attempted to stat the initial directory (after having successfully obtained its name via `getcwd`), however, the call failed for some reason. No attempt will be made to delete anything.

cannot chdir to [dir]: [errmsg]

> `remove_tree` attempted to set the working directory in order to begin deleting the objects therein, but was unsuccessful. This is usually a permissions issue. The routine will continue to delete other things, but this directory will be left intact.

directory [dir] changed before chdir, expected dev=[n] ino=[n], actual dev=[n] ino=[n], aborting. (FATAL)

> `remove_tree` recorded the device and inode of a directory, and then moved into it. It then performed a `stat` on the current directory and detected that the device and inode were no longer the same. As this is at the heart of the race condition problem, the program will die at this point.

cannot make directory [dir] read+writeable: [errmsg]

> `remove_tree` attempted to change the permissions on the current directory to ensure that subsequent unlinkings would not run into problems, but was unable to do so. The permissions remain as they were, and the program will carry on, doing the best it can.

cannot read [dir]: [errmsg]

> `remove_tree` tried to read the contents of the directory in order to acquire the names of the directory entries to be unlinked, but was unsuccessful. This is usually a permissions issue.

The program will continue, but the files in this directory will remain after the call.

cannot reset chmod [dir]: [errmsg]

`remove_tree`, after having deleted everything in a directory, attempted to restore its permissions to the original state but failed. The directory may wind up being left behind.

cannot remove [dir] when cwd is [dir]

The current working directory of the program is */some/path/to/here* and you are attempting to remove an ancestor, such as */some/path*. The directory tree is left untouched.

The solution is to `chdir` out of the child directory to a place outside the directory tree to be removed.

cannot chdir to [parent-dir] from [child-dir]: [errmsg], aborting. (FATAL)

`remove_tree`, after having deleted everything and restored the permissions of a directory, was unable to chdir back to the parent. The program halts to avoid a race condition from occurring.

cannot stat prior working directory [dir]: [errmsg], aborting. (FATAL)

`remove_tree` was unable to stat the parent directory after have returned from the child. Since there is no way of knowing if we returned to where we think we should be (by comparing device and inode) the only way out is to `croak`.

previous directory [parent-dir] changed before entering [child-dir], expected dev=[n] ino=[n], actual dev=[n] ino=[n], aborting. (FATAL)

When `remove_tree` returned from deleting files in a child directory, a check revealed that the parent directory it returned to wasn't the one it started out from. This is considered a sign of malicious activity.

cannot make directory [dir] writeable: [errmsg]

Just before removing a directory (after having successfully removed everything it contained), `remove_tree` attempted to set the permissions on the directory to ensure it could be removed and failed. Program execution continues, but the directory may possibly not be deleted.

cannot remove directory [dir]: [errmsg]

`remove_tree` attempted to remove a directory, but failed. This may because some objects that were unable to be removed remain in the directory, or a permissions issue. The directory will be left behind.

cannot restore permissions of [dir] to [0nnn]: [errmsg]

After having failed to remove a directory, `remove_tree` was unable to restore its permissions from a permissive state back to a possibly more restrictive setting. (Permissions given in octal).

cannot make file [file] writeable: [errmsg]

`remove_tree` attempted to force the permissions of a file to ensure it could be deleted, but failed to do so. It will, however, still attempt to unlink the file.

cannot unlink file [file]: [errmsg]

`remove_tree` failed to remove a file. Probably a permissions issue.

cannot restore permissions of [file] to [0nnn]: [errmsg]

After having failed to remove a file, `remove_tree` was also unable to restore the permissions on the file to a possibly less permissive setting. (Permissions given in octal).

unable to map [owner] to a uid, ownership not changed");

`make_path` was instructed to give the ownership of created directories to the symbolic name [owner], but `getpwnam` did not return the corresponding numeric uid. The directory will be created, but ownership will not be changed.

unable to map [group] to a gid, group ownership not changed

`make_path` was instructed to give the group ownership of created directories to the symbolic name [group], but `getgrnam` did not return the corresponding numeric gid. The directory will be created, but group ownership will not be changed.

SEE ALSO

- *File::Remove*

  Allows files and directories to be moved to the Trashcan/Recycle Bin (where they may later be restored if necessary) if the operating system supports such functionality. This feature may one day be made available directly in `File::Path`.

- *File::Find::Rule*

  When removing directory trees, if you want to examine each file to decide whether to delete it (and possibly leaving large swathes alone), *File::Find::Rule* offers a convenient and flexible approach to examining directory trees.

BUGS

Please report all bugs on the RT queue:

http://rt.cpan.org/NoAuth/Bugs.html?Dist=File-Path

You can also send pull requests to the Github repository:

https://github.com/dland/File-Path

ACKNOWLEDGEMENTS

Paul Szabo identified the race condition originally, and Brendan O'Dea wrote an implementation for Debian that addressed the problem. That code was used as a basis for the current code. Their efforts are greatly appreciated.

Gisle Aas made a number of improvements to the documentation for 2.07 and his advice and assistance is also greatly appreciated.

AUTHORS

Tim Bunce and Charles Bailey. Currently maintained by David Landgren <*david@landgren.net*>.

COPYRIGHT

LICENSE