NAME

File::Path - Create or remove directory trees

VERSION

This document describes version 2.04 of File::Path, released 2007-11-13.

SYNOPSIS

```
use File::Path;

# modern
mkpath( 'foo/bar/baz', '/zug/zwang', {verbose => 1} );

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

# traditional
mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);
rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);
```

DESCRIPTION

The `mkpath` function provides a convenient way to create directories of arbitrary depth. Similarly, the `rmtree` function provides a convenient way to delete an entire directory subtree from the filesystem, much like the Unix command `rm -r`.

Both functions may be called in one of two ways, the traditional, compatible with code written since the dawn of time, and modern, that offers a more flexible and readable idiom. New code should use the modern interface.

FUNCTIONS

The modern way of calling `mkpath` and `rmtree` is with a list of directories to create, or remove, respectively, followed by an optional hash reference containing keys to control the function's behaviour.

mkpath

The following keys are recognised as parameters to `mkpath`. The function returns the list of files actually created during the call.

```
my @created = mkpath(
  qw(/tmp /flub /home/nobody),
  {verbose => 1, mode => 0750},
);
print "created $_\n" for @created;
```

mode

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

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

error

If present, will be interpreted as a reference to a list, and 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.

rmtree

verbose

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

safe

When set to a true value, will cause `rmtree` 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

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.

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

result

If present, will be interpreted as a reference to a list, and will be used to store the list of all files and directories unlinked during the call. If nothing is unlinked, a reference to an empty list is returned (rather than `undef`).

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

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

error

If present, will be interpreted as a reference to a list, and 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 `rmtree` 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.

TRADITIONAL INTERFACE

The old interfaces of `mkpath` and `rmtree` take a reference to a list of directories (to create or remove), followed by a series of positional, numeric, modal parameters that control their behaviour.

This design made it difficult to add additional functionality, as well as posed the problem of what to do when the calling code only needs to set the last parameter. Even though the code doesn't care how the initial positional parameters are set, the programmer is forced to learn what the defaults are, and specify them.

Worse, if it turns out in the future that it would make more sense to change the default behaviour of the first parameter (for example, to avoid a security vulnerability), all existing code will remain hard-wired to the wrong defaults.

Finally, a series of numeric parameters are much less self-documenting in terms of communicating to the reader what the code is doing. Named parameters do not have this problem.

In the traditional API, `mkpath` takes three arguments:

- The name of the path to create, or a reference to a list of paths to create,
- a boolean value, which if TRUE will cause `mkpath` to print the name of each directory as it is created (defaults to FALSE), and
- the numeric mode to use when creating the directories (defaults to 0777), to be modified by the current umask.

It returns a list of all directories (including intermediates, determined using the Unix '/' separator) created. In scalar context it returns the number of directories created.

If a system error prevents a directory from being created, then the `mkpath` function throws a fatal error with `Carp::croak`. This error can be trapped with an `eval` block:

```
eval { mkpath($dir) };
if ($@) {
  print "Couldn't create $dir: $@";
}
```

In the traditional API, `rmtree` takes three arguments:

- the root of the subtree to delete, or a reference to a list of roots. All of the files and directories below each root, as well as the roots themselves, will be deleted. If you want to keep the roots themselves, you must use the modern API.
- a boolean value, which if TRUE will cause `rmtree` to print a message each time it examines a file, giving the name of the file, and indicating whether it's using `rmdir` or `unlink` to remove it, or that it's skipping it. (defaults to FALSE)
- a boolean value, which if TRUE will cause `rmtree` to skip any files to which you do not have delete access (if running under VMS) or write access (if running under another OS). This will change in the future when a criterion for 'delete permission' under OSs other than VMS is settled. (defaults to FALSE)

It returns the number of files, directories and symlinks successfully deleted. Symlinks are simply deleted and not followed.

Note also that the occurrence of errors in `rmtree` using the traditional interface can be determined *only* by trapping diagnostic messages using `$SIG{__WARN__}`; it is not apparent from the return value. (The modern interface may use the `error` parameter to record any problems encountered).

ERROR HANDLING

If `mkpath` or `rmtree` 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 result is a reference to a list of hash references. For each hash reference, the key is the name of the file, and the value is the error message (usually the contents of `$!`). An example usage looks like:

```
rmpath( 'foo/bar', 'bar/rat', {error => \my $err} );
for my $diag (@$err) {
  my ($file, $message) = each %$diag;
  print "problem unlinking $file: $message\n";
}
```

If no errors are encountered, `$err` will point to an empty list (thus there is no need to test for `undef`). If a general error is encountered (for instance, `rmtree` attempts to remove a directory tree that does not exist), the diagnostic key will be empty, only the value will be set:

```
rmpath( '/no/such/path', {error => \my $err} );
for my $diag (@$err) {
  my ($file, $message) = each %$diag;
  if ($file eq '') {
    print "general error: $message\n";
  }
}
```

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';
```

HEURISTICS

The functions detect (as far as possible) which way they are being called and will act appropriately. It is important to remember that the heuristic for detecting the old style is either the presence of an array reference, or two or three parameters total and second and third parameters are numeric. Hence...

```
mkpath 486, 487, 488;
```

... will not assume the modern style and create three directories, rather it will create one directory verbosely, setting the permission to 0750 (488 being the decimal equivalent of octal 750). Here, old style trumps new. It must, for backwards compatibility reasons.

If you want to ensure there is absolutely no ambiguity about which way the function will behave, make sure the first parameter is a reference to a one-element list, to force the old style interpretation:

```
mkpath [486], 487, 488;
```

and get only one directory created. Or add a reference to an empty parameter hash, to force the new style:

```
mkpath 486, 487, 488, {};
```

... and hence create the three directories. If the empty hash reference seems a little strange to your eyes, or you suspect a subsequent programmer might *helpfully* optimise it away, you can add a parameter set to a default value:

```
mkpath 486, 487, 488, {verbose => 0};
```

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 `rmtree` 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)

> `mkpath` 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

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

cannot fetch initial working directory: [errmsg]

> `rmtree` 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]

> `rmtree` 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]

> `rmtree` 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] inode=[n], actual dev=[n] ino=[n], aborting. (FATAL)

> `rmtree` 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]

> `rmtree` 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]

> `rmtree` 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]

`rmtree`, 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 chdir to [parent-dir] from [child-dir]: [errmsg], aborting. (FATAL)

`rmtree`, after having deleted everything and restored the permissions of a directory, was unable to chdir back to the parent. This is usually a sign that something evil this way comes.

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

`rmtree` 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] inode=[n], actual dev=[n] ino=[n], aborting. (FATAL)

When `rmtree` 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), `rmtree` 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]

`rmtree` 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, `rmtree` 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]

`rmtree` 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]

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

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

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

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

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.

AUTHORS

Tim Bunce <*Tim.Bunce@ig.co.uk*> and Charles Bailey <*bailey@newman.upenn.edu*>. Currently maintained by David Landgren <*david@landgren.net*>.

COPYRIGHT

LICENSE