Linux Ubuntu 22.4.5 Manual Pages on command 'Git::Wrapper.3pm'

$ man Git::Wrapper.3pm

Git::Wrapper(3pm) User Contributed Perl Documentation Git::Wrapper(3pm)

NAME

Git::Wrapper - Wrap git(7) command-line interface

VERSION

version 0.048

SYNOPSIS

```
my $git = Git::Wrapper->new('/var/foo');
$git->commit(...)
print $_->message for $git->log;
# specify which git binary to use
my $git = Git::Wrapper->new({
  dir        => '/var/foo' ,
  git_binary => '/path/to/git/bin/git' ,
});
```

DESCRIPTION

Git::Wrapper provides an API for git(7) that uses Perl data structures for argument passing, instead of CLI-style "--options" as Git does.

METHOD INVOCATION

Except as documented, every git subcommand is available as a method on a Git::Wrapper object. Replace any hyphens in the git command with underscores (for example, "git init-db" would become "$git->init_db").

Method Arguments

Methods accept a combination of hashrefs and scalars, which is used to build the command used to invoke git. Arguments passed in hashrefs will be automatically parsed into option pairs, but the ordering of these in the resulting shell command is not guaranteed (with the exception of options with a leading '-'; see below). Options that are passed as plain scalars will retain their order. Some examples may help clarify. This code:

```
$git->commit({ message => "stuff" , all => 1 });
```

may produce this shell command:

```
git commit --all --message="stuff"
```

This code, however:

```
$git->commit(qw/ --message "stuff" / , { all => 1 });
```

will always produce this shell command:

```
git commit --message "stuff" --all
```

In most cases, this exact control over argument ordering is not needed and simply passing all options as part of a hashref, and all other options as additional list arguments, will be sufficient. In some cases, however, the ordering of options to particular git sub-commands is significant, resulting in the need for this level of control.

N.b. Options that are given with a leading '-' (with the exception of special options noted below) are applied as arguments to the "git" command itself; options without a leading '-' are applied as arguments to the sub-command. For example:

```
$git->command({ -foo => 1 , bar => 2 });
```

invokes the command line

```
git --foo=1 command --bar=2
```

N.b. Because of the way arguments are parsed, should you need to pass an explicit '0' value to an option (for example, to have the same effect as "--abbrev=0" on the command line), you should pass it with a leading space, like so:

```
$git->describe({ abbrev => ' 0' };
```

To pass content via STDIN, use the -STDIN option:

```
$git->hash_object({ stdin => 1, -STDIN => 'content to hash' });
```

Output is available as an array of lines, each chomped.

```
@sha1s_and_titles = $git->rev_list({ all => 1, pretty => 'oneline' });
```

Passing stringify-able objects as arguments

Objects may be passed in the place of scalars, assuming those objects overload stringification in such a way as to produce a useful value. However, relying on this stringification is discouraged and likely to be officially deprecated in a subsequent release. Instead, if you have an object that stringifies to a meaningful value (e.g., a Path::Class object), you should stringify it yourself before passing it to "Git::Wrapper" methods.

Error handling

If a git command exits nonzero, a "Git::Wrapper::Exception" object will be thrown (via "die") and may be captured via "eval" or Try::Tiny, for example.

The error object has three useful methods:

- error

  Returns the full error message reported by the resulting git command sent to "STDERR". This method should not be used as a success/failure check, as "git" will sometimes produce output on STDERR when a command is successful.

- output

  Returns the full output generated by the git command that is sent to "STDOUT". This method should not be used as a success/failure check, as "git" will frequently not have any output with a successful command.

- status

  Returns the non-zero exit code reported by git on error.

Using Try::Tiny

Try::Tiny is the recommended way to catch exception objects thrown by Git::Wrapper.

```
use Try::Tiny
my $git = Git::Wrapper->new('/path/to/my/repo');
try {
  # equivalent to, "git --non-existent-option=1" on the commandline
  $git->status({ "non-existent-option"=>1 });
}
catch {
  # print STERR from erroneous git command
  print $_->error;
  # print STOUT from git command
  print $_->output;
```

```
    # print non-zero exist status of git processo
    print $_->status;
    # quotes are overloaded, so:
    print "$_"; # equivalent to $_->error
  };
```

Using "eval"

If for some reason you are unable to use Try::Tiny, it is also possible to use the "eval" function to catch exception objects. THIS IS NOT RECOMMENDED!

```
  my $git = Git::Wrapper->new('/path/to/my/repo');
  my $ok = eval {
    # equivalent to, "git --non-existent-option=1" on the commandline
    $git->status({ "non-existent-option"=>1 });
    1;
  };
  if ($@ and ref $@ eq q{Git::Wrapper::Exception}) {
    # print STERR from erroneous git command
    print $@->error;
    # print STOUT from git command
    print $@->output;
    # print non-zero exist status of git processo
    print $@->status;
    # quotes are overloaded, so:
    print "$@"; # equivalent to $@->error
  }
```

METHODS

new

```
  my $git = Git::Wrapper->new($dir);
  my $git = Git::Wrapper->new({ dir => $dir , git_binary => '/path/to/git' });
  # To force the git binary location
  my $git = Git::Wrapper->new($dir, 'git_binary' => '/usr/local/bin/git');
  # prints the content of OUT and ERR to STDOUT and STDERR
  # after a command is run
  my $git = Git::Wrapper->new($dir, autoprint => 1);
```

git

print $git->git; # /path/to/git/binary/being/used

dir

print $git->dir; # /var/foo

version

my $version = $git->version; # 1.6.1.4.8.15.16.23.42

branch

my @branches = $git->branch;

This command intentionally disables ANSI color highlighting in the output. If you want ANSI color highlighting, you'll need to bypass via the RUN() method (see below).

log

my @logs = $git->log;

Instead of giving back an arrayref of lines, the "log" method returns a list of "Git::Wrapper::Log" objects.

There are five methods in a "Git::Wrapper::Log" objects:

? id

? author

? date

? message

? modifications

Only populated with when "raw => 1" option is set; see "Raw logs" below.

Raw logs

Calling the "log" method with the "raw => 1" option set, as below, will do additional parsing to populate the "modifications" attribute on each "Git::Wrapper::Log" object. This method returns a list of "Git::Wrapper::File::RawModification" objects, which can be used to get filenames, permissions, and other metadata associated with individual files in the given commit. A short example, to loop over all commits in the log and print the filenames that were changed in each commit, one filename per file:

```
my @logs = $git->log({ raw => 1 });
foreach my $log ( @logs ) {
    say "In commit '" . $log->id . "', the following files changed:";
```

```
        my @mods = $log->modifications;
        foreach my $mod ( @mods ) {
            say "\t" . $mod->filename;
        }
    }
```

Note that some commits (e.g., merge commits) will not contain any file changes. The "modifications" method will return an empty list in that case.

Custom log formats

"log" will throw an exception if it is passed the "--format" option. The reason for this has to do with the fact that the parsing of the full log output into "Git::Wrapper::Log" objects assumes the default format provided by `git` itself. Passing "--format" to the underlying `git log` method affects this assumption and the output is no longer able to be processed as intented.

If you wish to specify a custom log format, please use the RUN method directly. The caller will be supplied with the full log output. From there, the caller may process the output as it wishes.

has_git_in_path

This method returns a true or false value indicating if there is a 'git' binary in the current $PATH.

supports_status_porcelain

supports_log_no_abbrev_commit

supports_log_no_expand_tabs

supports_log_raw_dates

supports_hash_object_filters

These methods return a true or false value (1 or 0) indicating whether the git binary being used has support for these options. (The '--porcelain' option on 'git status', the '--no-abbrev-commit', '--no-expand-tabs', and '--date=raw' options on 'git log', and the '--no-filters' option on 'git hash-object' respectively.)

These are primarily for use in this distribution's test suite, but may also be useful when writing code using Git::Wrapper that might be run with different versions of the underlying git binary.

status

When running with an underlying git binary that returns false for the

"supports_status_porcelain" method, this method will act like any other wrapped command: it will return output as an array of chomped lines.

When running with an underlying git binary that returns true for the "supports_status_porcelain" method, this method instead returns an instance of Git::Wrapper::Statuses:

```
my $statuses = $git->status;
```

Git::Wrapper:Statuses has two public methods. First, "is_dirty":

```
my $dirty_flag = $statuses->is_dirty;
```

which returns a true/false value depending on whether the repository has any uncommitted changes.

Second, "get":

```
my @status = $statuses->get($group)
```

which returns an array of Git::Wrapper::Status objects, one per file changed.

There are four status groups, each of which may contain zero or more changes.

? indexed : Changed & added to the index (aka, will be committed)

? changed : Changed but not in the index (aka, won't be committed)

? unknown : Untracked files

? conflict : Merge conflicts

Note that a single file can occur in more than one group. E.g., a modified file that has been added to the index will appear in the 'indexed' list. If it is subsequently further modified it will additionally appear in the 'changed' group.

A Git::Wrapper::Status object has three methods you can call:

```
my $from = $status->from;
```

The file path of the changed file, relative to the repo root. For renames, this is the original path.

```
my $to = $status->to;
```

Renames returns the new path/name for the path. In all other cases returns an empty string.

```
my $mode = $status->mode;
```

Indicates what has changed about the file.

Within each group (except 'conflict') a file can be in one of a number of modes, although some modes only occur in some groups (e.g., 'added' never appears in the 'unknown' group).

? modified

? added

? deleted

? renamed

? copied

? conflict

All files in the 'unknown' group will have a mode of 'unknown' (which is redundant

but at least consistent).

The 'conflict' group instead has the following modes.

? 'both deleted' : deleted on both branches

? 'both added' : added on both branches

? 'both modified' : modified on both branches

? 'added by us' : added only on our branch

? 'deleted by us' : deleted only on our branch

? 'added by them' : added on the branch we are merging in

? 'deleted by them' : deleted on the branch we are merging in

See git-status man page for more details.

Example

```
my $git = Git::Wrapper->new('/path/to/git/repo');
my $statuses = $git->status;
for my $type (qw<indexed changed unknown conflict>) {
    my @states = $statuses->get($type)
        or next;
    print "Files in state $type\n";
    for (@states) {
        print '  ', $_->mode, ' ', $_->from;
        print ' renamed to ', $_->to
            if $_->mode eq 'renamed';
        print "\n";
    }
}
```

RUN

This method bypasses the output rearranging performed by some of the wrapped

methods described above (i.e., "log", "status", etc.). This can be useful in various situations, such as when you want to produce a particular log output format that isn't compatible with the way "Git::Wrapper" constructs "Git::Wrapper::Log", or when you want raw "git status" output that isn't parsed into a "Git::Wrapper::Status" object.

This method should be called with an initial string argument of the "git" subcommand you want to run, followed by a hashref containing options and their values, and then a list of any other arguments.

Example

```
my $git = Git::Wrapper->new( '/path/to/git/repo' );
# the 'log' method returns Git::Wrapper::Log objects
my @log_objects = $git->log();
# while 'RUN('log')' returns an array of chomped lines
my @log_lines = $git->RUN('log');
# getting the full of commit SHAs via `git log` by using the '--format' option
my @log_lines = $git->RUN('log', '--format=%H');
```

AUTOPRINT($enabled)

If set to "true", the content of "OUT" and "ERR" will automatically be printed on, respectively, STDOUT and STDERR after a command is run.

ERR

After a command has been run, this method will return anything that was sent to "STDERR", in the form of an array of chomped lines. This information will be cleared as soon as a new command is executed. This method should *NOT* be used as a success/failure check, as "git" will sometimes produce output on STDERR when a command is successful.

OUT

After a command has been run, this method will return anything that was sent to "STDOUT", in the form of an array of chomped lines. It is identical to what is returned from the method call that runs the command, and is provided simply for symmetry with the "ERR" method. This method should *NOT* be used as a success/failure check, as "git" will frequently not have any output with a successful command.

COMPATIBILITY

On Win32 Git::Wrapper is incompatible with msysGit installations earlier than Git-1.7.1-preview20100612 due to a bug involving the return value of a git command in cmd/git.cmd. If you use the msysGit version distributed with GitExtensions or an earlier version of msysGit, tests will fail during installation of this module. You can get the latest version of msysGit on the Google Code project page: <http://code.google.com/p/msysgit/downloads>

ENVIRONMENT VARIABLES

Git::Wrapper normally uses the first 'git' binary in your path. The original override provided to change this was by setting the GIT_WRAPPER_GIT environment variable. Now that object creation accepts an override, you are encouraged to instead pass the binary location (git_binary) to new on object creation.

SEE ALSO

VCI::VCS::Git is the git implementation for VCI, a generic interface to version-control systems.

Other Perl Git Wrappers <https://metacpan.org/module/Git::Repository#OTHER-PERL-GIT-WRAPPERS> is a list of other Git interfaces in Perl. If Git::Wrapper doesn't scratch your itch, possibly one of the modules listed there will.

Git itself is at <http://git.or.cz>.

REPORTING BUGS & OTHER WAYS TO CONTRIBUTE

The code for this module is maintained on GitHub, at <https://github.com/genehack/Git-Wrapper>. If you have a patch, feel free to fork the repository and submit a pull request. If you find a bug, please open an issue on the project at GitHub. (We also watch the <http://rt.cpan.org> queue for Git::Wrapper, so feel free to use that bug reporting system if you prefer)

AUTHORS

- Hans Dieter Pearcey <hdp@cpan.org>
- Chris Prather <chris@prather.org>
- John SJ Anderson <genehack@genehack.org>

COPYRIGHT AND LICENSE