NAME

Git - Perl interface to the Git version control system

SYNOPSIS

```
use Git;

my $version = Git::command_oneline('version');

git_cmd_try { Git::command_noisy('update-server-info') }
'%s failed w/ code %d';

my $repo = Git->repository (Directory => '/srv/git/cogito.git');


my @revs = $repo->command('rev-list', '--since=last monday', '--all');

my ($fh, $c) = $repo->command_output_pipe('rev-list', '--since=last monday', '--all');
my $lastrev = <$fh>; chomp $lastrev;
$repo->command_close_pipe($fh, $c);

my $lastrev = $repo->command_oneline( [ 'rev-list', '--all' ],
STDERR => 0 );

my $sha1 = $repo->hash_and_insert_object('file.txt');
my $tempfile = tempfile();
my $size = $repo->cat_blob($sha1, $tempfile);
```

DESCRIPTION

This module provides Perl scripts easy way to interface the Git version control system. The modules have an easy and well-tested way to call arbitrary Git commands; in the future, the interface will also provide specialized methods for doing easily operations which are not totally trivial to do over the generic command interface.

While some commands can be executed outside of any context (e.g. 'version' or 'init'), most operations require a repository context, which in practice means getting an instance of the Git object using the *repository()* constructor. (In the future, we will also get a *new_repository()* constructor.) All commands called as methods of the object are then executed in the context of the repository.

Part of the "repository state" is also information about path to the attached working copy (unless you work with a bare repository). You can also navigate inside of the working copy using the `wc_chdir()` method. (Note that the repository object is self-contained and will not change working directory of your process.)

TODO: In the future, we might also do

```
my $remoterepo = $repo->remote_repository (Name => 'cogito', Branch => 'master');
       $remoterepo ||= Git->remote_repository ('http://git.or.cz/cogito.git/');
my @refs = $remoterepo->refs();
```

Currently, the module merely wraps calls to external Git tools. In the future, it will provide a much faster way to interact with Git by linking directly to libgit. This should be completely opaque to the user, though (performance increase notwithstanding).

CONSTRUCTORS

repository (OPTIONS)
repository (DIRECTORY)

repository ()

Construct a new repository object. `OPTIONS` are passed in a hash lik e fashion, using key and value pairs. Possible options are:

Repository - Path to the Git repository.

WorkingCopy - Path to the associated working copy; not strictly required as many commands will happily crunch on a bare repository.

WorkingSubdir - Subdirectory in the working copy to work inside. Just left undefined if you do not want to limit the scope of operations.

Directory - Path to the Git working directory in its usual setup. The `.git` directory is searched in the directory and all the parent directories; if found, `WorkingCopy` is set to the directory containing it and `Repository` to the `.git` directory itself. If no `.git` directory was found, the `Directory` is assumed to be a bare repository, `Repository` is set to point at it and `WorkingCopy` is left undefined. If the `$GIT_DIR` environment variable is set, things behave as expected as well.

You should not use both `Directory` and either of `Repository` and `WorkingCopy` - the results of that are undefined.

Alternatively, a directory path may be passed as a single scalar argument to the constructor; it is equivalent to setting only the `Directory` option field.

Calling the constructor with no options whatsoever is equivalent to calling it with `Directory => '.'`. In general, if you are building a standard porcelain command, simply doing `Git->repository()` should do the right thing and setup the object to reflect exactly where the user is right now.

METHODS

command (COMMAND [, ARGUMENTS...])

command ([COMMAND, ARGUMENTS...], { Opt => Val ... })

Execute the given Git `COMMAND` (specify it without the 'git-' prefix), optionally with the specified extra `ARGUMENTS`.

The second more elaborate form can be used if you want to further adjust the command execution. Currently, only one option is supported:

STDERR - How to deal with the command's error output. By default (`undef`) it is delivered to the caller's `STDERR`. A false value (0 or '') will cause it to be thrown away. If you want to process it, you can get it in a filehandle you specify, but you must be extremely careful; if the error output is not very short and you want to read it in the same process as where you called `command()`, you are set up for a nice deadlock!

The method can be called without any instance or on a specified Git repository (in that case the command will be run in the repository context).

In scalar context, it returns all the command output in a single string (verbatim).

In array context, it returns an array containing lines printed to the command's stdout (without trailing newlines).

In both cases, the command's stdin and stderr are the same as the caller's.

command_oneline (COMMAND [, ARGUMENTS...])

command_oneline ([COMMAND, ARGUMENTS...], { Opt => Val ... })

Execute the given `COMMAND` in the same way as *command()* does but always return a scalar string containing the first line of the command's standard output.

command_output_pipe (COMMAND [, ARGUMENTS...])

command_output_pipe ([COMMAND, ARGUMENTS...], { Opt => Val ... })
Execute the given `COMMAND` in the same way as *command()* does but return a pipe filehandle from which the command output can be read.

The function can return `($pipe, $ctx)` in array context. See`command_close_pipe()` for details.

command_input_pipe (COMMAND [, ARGUMENTS...])
command_input_pipe ([COMMAND, ARGUMENTS...], { Opt => Val ... })
Execute the given `COMMAND` in the same way as *command_output_pipe()* does but return an input pipe filehandle instead; the command output is not captured.

The function can return `($pipe, $ctx)` in array context. See`command_close_pipe()` for details.

command_close_pipe (PIPE [, CTX])
Close the `PIPE` as returned from `command_*_pipe()`, checking whether the command finished successfully. The optional `CTX` argument is required if you want to see the command name in the error message, and it is the second value returned by `command_*_pipe()` when called in array context. The call idiom is:

```
my ($fh, $ctx) = $r->command_output_pipe('status');
while (<$fh>) { ... }
$r->command_close_pipe($fh, $ctx);
```

Note that you should not rely on whatever actually is in `CTX`; currently it is simply the command name but in future the context might have more complicated structure.

command_bidi_pipe (COMMAND [, ARGUMENTS...])
Execute the given `COMMAND` in the same way as *command_output_pipe()* does but return both an input pipe filehandle and an output pipe filehandle.

The function will return return `($pid, $pipe_in, $pipe_out, $ctx)`. See `command_close_bidi_pipe()` for details.

command_close_bidi_pipe (PID, PIPE_IN, PIPE_OUT [, CTX])
Close the `PIPE_IN` and `PIPE_OUT` as returned from `command_bidi_pipe()`, checking whether the command finished successfully. The optional `CTX` argument is required if you want to see the command name in the error message, and it is the fourth value returned by `command_bidi_pipe()`. The call idiom is:

```
my ($pid, $in, $out, $ctx) = $r->command_bidi_pipe('cat-file --batch-check');
print $out "000000000\n";
while (<$in>) { ... }
$r->command_close_bidi_pipe($pid, $in, $out, $ctx);
```

Note that you should not rely on whatever actually is in `CTX`; currently it is simply the command name but in future the context might have more complicated structure.

`PIPE_IN` and `PIPE_OUT` may be `undef` if they have been closed prior to calling this function. This may be useful in a query-response type of commands where caller first writes a query and later reads response, eg:

```
my ($pid, $in, $out, $ctx) = $r->command_bidi_pipe('cat-file --batch-check');
print $out "000000000\n";
close $out;
while (<$in>) { ... }
$r->command_close_bidi_pipe($pid, $in, undef, $ctx);
```

This idiom may prevent potential dead locks caused by data sent to the output pipe not being flushed and thus not reaching the executed command.

command_noisy (COMMAND [, ARGUMENTS...])
Execute the given `COMMAND` in the same way as *command()* does but do not capture the command output - the standard output is not redirected and goes to the standard output of the caller application.

While the method is called *command_noisy()*, you might want to as well use it for the most silent Git commands which you know will never pollute your stdout but you want to avoid the overhead of the pipe setup when calling them.

The function returns only after the command has finished running.

version ()
Return the Git version in use.

exec_path ()
Return path to the Git sub-command executables (the same as `git --exec-path`). Useful mostly only internally.

html_path ()
Return path to the Git html documentation (the same as `git --html-path`). Useful mostly only internally.

get_tz_offset (TIME)
Return the time zone offset from GMT in the form +/-HHMM where HH is the number of hours from GMT and MM is the number of minutes. This is the equivalent of what strftime("%z", ...) would provide on a GNU platform.

If TIME is not supplied, the current local time is used.

prompt (PROMPT , ISPASSWORD)
Query user `PROMPT` and return answer from user.

Honours GIT_ASKPASS and SSH_ASKPASS environment variables for querying the user. If no *_ASKPASS variable is set or an error occoured, the terminal is tried as a fallback. If `ISPASSWORD` is set and true, the terminal disables echo.

repo_path ()
Return path to the git repository. Must be called on a repository instance.

wc_path ()
Return path to the working copy. Must be called on a repository instance.

wc_subdir ()
Return path to the subdirectory inside of a working copy. Must be called on a repository instance.

wc_chdir (SUBDIR)
Change the working copy subdirectory to work within. The `SUBDIR` is relative to the working copy root directory (not the current subdirectory). Must be called on a repository instance attached to a working copy and the directory must exist.

config (VARIABLE)
Retrieve the configuration `VARIABLE` in the same manner as `config` does. In scalar context requires the variable to be set only one time (exception is thrown otherwise), in array context returns allows the variable to be set multiple times and returns all the values.

config_bool (VARIABLE)
Retrieve the bool configuration `VARIABLE`. The return value is usable as a boolean in perl (and `undef` if it's not defined, of course).

config_path (VARIABLE)
Retrieve the path configuration `VARIABLE`. The return value is an expanded path or `undef` if it's not defined.

config_int (VARIABLE)
: Retrieve the integer configuration `VARIABLE`. The return value is simple decimal number. An optional value suffix of 'k', 'm', or 'g' in the config file will cause the value to be multiplied by 1024, 1048576 (1024^2), or 1073741824 (1024^3) prior to output. It would return `undef` if configuration variable is not defined,

get_colorbool (NAME)
: Finds if color should be used for NAMEd operation from the configuration, and returns boolean (true for "use color", false for "do not use color").

get_color (SLOT, COLOR)
: Finds color for SLOT from the configuration, while defaulting to COLOR, and returns the ANSI color escape sequence:

```
print $repo->get_color("color.interactive.prompt", "underline blue white");
print "some text";
print $repo->get_color("", "normal");
```

remote_refs (REPOSITORY [, GROUPS [, REFGLOBS]])
: This function returns a hashref of refs stored in a given remote repository. The hash is in the format `refname =\ hash>`. For tags, the `refname` entry contains the tag object while a `refname{}` entry gives the tagged objects.

  `REPOSITORY` has the same meaning as the appropriate `git-ls-remote` argument; either a URL or a remote name (if called on a repository instance). `GROUPS` is an optional arra yref that can contain 'tags' to return all the tags and/or 'heads' to return all the heads. `REFGLOB` is an optional array of strings containing a shell-like glob to further limit the refs returned in the hash; the meaning is again the same as the appropriate `git-ls-remote` argument.

  This function may or may not be called on a repository instance. In the former case, remote names as defined in the repository are recognized as repository specifiers.

ident (TYPE | IDENTSTR)
ident_person (TYPE | IDENTSTR | IDENTARRAY)
: This suite of functions retrieves and parses ident information, as stored in the commit and tag objects or produced by `var GIT_type_IDENT` (thus `TYPE` can be either *author* or *committer*; case is insignificant).

  The `ident` method retrieves the ident information from `git var` and either returns it as a scalar string or as an array with the fields parsed. Alternatively, it can take a prepared ident string (e.g. from the commit object) and just parse it.

  `ident_person` returns the person part of the ident - name and email; it can take the same arguments as `ident` or the array returned by `ident`.

  The synopsis is like:

```
my ($name, $email, $time_tz) = ident('author');
"$name <$email>" eq ident_person('author');
"$name <$email>" eq ident_person($name);
$time_tz = /\d+ [+-]\d{4}$/;
```

hash_object (TYPE, FILENAME)
: Compute the SHA1 object id of the given `FILENAME` considering it is of the `TYPE` object type (`blob`, `commit`, `tree`).

  The method can be called without any instance or on a specified Git repository, it makes zero difference.

  The function returns the SHA1 hash.

hash_and_insert_object (FILENAME)
: Compute the SHA1 object id of the given `FILENAME` and add the object to the object database.

  The function returns the SHA1 hash.

cat_blob (SHA1, FILEHANDLE)
: Prints the contents of the blob identified by `SHA1` to `FILEHANDLE` and returns the number of bytes printed.

credential_read(FILEHANDLE)
: Reads credential key-value pairs from `FILEHANDLE`. Reading stops at EOF or when an empty line is encountered. Each line must be of the form `key=value` with a non-empty key. Function returns hash with all read values. Any white space (other than new-line character) is preserved.

credential_write(FILEHANDLE, CREDENTIAL_HASHREF)
: Writes credential key-value pairs from hash referenced by `CREDENTIAL_HASHREF` to `FILEHANDLE`. Keys and values cannot contain new-lines or NUL bytes characters, and key cannot contain equal signs nor be empty (if they do Error::Simple is thrown). Any white space is preserved. If value for a key is `undef`, it will be skipped.

  If `'url'` key exists it will be written first. (All the other key-value pairs are written in sorted order but you should not depend on that). Once all lines are written, an empty line is printed.

credential(CREDENTIAL_HASHREF [, OPERATION])
credential(CREDENTIAL_HASHREF, CODE)
: Executes `git credential` for a given set of credentials and specified operation. In both forms `CREDENTIAL_HASHREF` needs to be a reference to a hash which stores credentials. Under certain conditions the hash can change.

  In the first form, `OPERATION` can be `'fill'`, `'approve'` or `'reject'`, and function will execute corresponding `git credential` sub-command. If it's omitted `'fill'` is assumed. In case of `'fill'` the values stored in `CREDENTIAL_HASHREF` will be changed to the ones returned by the `git credential fill` command. The usual usage would look something like:

```
my %cred = (
'protocol' => 'https',
'host' => 'example.com',
'username' => 'bob'
);
Git::credential \%cred;
if (try_to_authenticate($cred{'username'}, $cred{'password'})) {
Git::credential \%cred, 'approve';
... do more stuff ...
} else {
Git::credential \%cred, 'reject';
}
```

  In the second form, `CODE` needs to be a reference to a subroutine. The function will execute `git credential fill`to fill the pro vided credential hash, then call `CODE` with `CREDENTIAL_HASHREF` as the sole argument. If `CODE`'s return value is defined, the function will execute `git credential approve` (if return value yields true) or `git credential reject` (if return value is false). If the return value is undef, nothing at all is executed; this is useful, for example, if the credential could neither be verified nor rejected due to an unrelated network error. The return value is the same as what `CODE` returns. With this form, the usage might look as follows:

```
if (Git::credential {
'protocol' => 'https',
'host' => 'example.com',
'username' => 'bob'
}, sub {
my $cred = shift;
return !!try_to_authenticate($cred->{'username'},
$cred->{'password'});
}) {
... do more stuff ...
}
```

temp_acquire (NAME)

Attempts to retrieve the temporary file mapped to the string `NAME`. If an associated temp file has not been created this session or was closed, it is created, cached, and set for autoflush and binmode.

Internally locks the file mapped to `NAME`. This lock must be released with `temp_release()` when the temp file is no longer needed. Subsequent attempts to retrieve temporary files mapped to the same `NAME` while still locked will cause an error. This locking mechanism provides a weak guarantee and is not threadsafe. It does provide some error checking to help prevent temp file refs writing over one another.

In general, the File::Handle returned should not be closed by consumers as it defeats the purpose of this caching mechanism. If you need to close the temp file handle, then you should use File::Temp or another temp file faculty directly. If a handle is closed and then requested again, then a warning will issue.

temp_is_locked (NAME)

Returns true if the internal lock created by a previous `temp_acquire()` call with `NAME` is still in effect.

When temp_acquire is called on a `NAME`, it internally locks the temporary file mapped to `NAME`. That lock will not be released until `temp_release()` is called with either the original `NAME` or the File::Handle that was returned from the original call to temp_acquire.

Subsequent attempts to call `temp_acquire()` with the same `NAME` will fail unless there has been an intervening `temp_release()` call for that `NAME` (or its corresponding File::Handle that was returned by the original `temp_acquire()` call).

If true is returned by `temp_is_locked()` for a `NAME`, an attempt to `temp_acquire()` the same `NAME` will cause an error unless `temp_release` is first called on that `NAME` (or its corresponding File::Handle that was returned by the original `temp_acquire()` call).

temp_release (NAME)
temp_release (FILEHANDLE)

Releases a lock acquired through `temp_acquire()`. Can be called either with the `NAME` mapping used when acquiring the temp file or with the `FILEHANDLE` referencing a locked temp file.

Warns if an attempt is made to release a file that is not locked.

The temp file will be truncated before being released. This can help to reduce disk I/O where the system is smart enough to detect the truncation while data is in the output buffers. Beware that after the temp file is released and truncated, any operations on that file may fail miserably until it is re-acquired. All contents are lost between each release and acquire mapped to the same string.

temp_reset (FILEHANDLE)
Truncates and resets the position of the `FILEHANDLE`.

temp_path (NAME)
temp_path (FILEHANDLE)
Returns the filename associated with the given tempfile.

ERROR HANDLING

All functions are supposed to throw Perl exceptions in case of errors. See the Error module on how to catch those. Most exceptions are mere Error::Simple instances.

However, the `command()`, `command_oneline()` and `command_noisy()` functions suite can throw `Git::Error::Command` exceptions as well: those are thrown when the external command returns an error code and contain the error code as well as access to the captured command's output. The exception class provides the usual `stringify` and `value` (command's exit code) methods and in addition also a `cmd_output` method that returns either an array or a string with the captured command output (depending on the original function call context; `command_noisy()` returns `undef`) and $<cmdline> which returns the command and its arguments (but without proper quoting).

Note that the `command_*_pipe()` functions cannot throw this exception since it has no idea whether the command failed or not. You will only find out at the time you `close` the pipe; if you want to have that automated, use `command_close_pipe()`, which can throw the exception.

git_cmd_try { CODE } ERRMSG
This magical statement will automatically catch any `Git::Error::Command` exceptions thrown by `CODE` and make your program die with `ERRMSG` on its lips; the message will have `%s` substituted for the command line and `%d` for the exit status. This statement is useful mostly for producing more user-friendly error messages.

In case of no exception caught the statement returns `CODE`'s return value.

Note that this is the only auto-exported function.

COPYRIGHT