Package: gert (via r-universe)

October 18, 2026

Type Package

Title Simple Git Client for R

Version 2.5.0

Description Simple git client for R based on 'libgit2' <https://libgit2.org> with support for SSH and HTTPS remotes. All functions in 'gert' use basic R data types (such as vectors and data-frames) for their arguments and return values. User credentials are shared with command line 'git' through the git-credential store and ssh keys stored on disk or ssh-agent.

License MIT + file LICENSE

URL https://docs.ropensci.org/gert/, https://ropensci.r-universe.dev/gert

BugReports https://github.com/r-lib/gert/issues

Imports askpass, credentials (>= 1.2.1), openssl (>= 2.0.3), rstudioapi (>= 0.11), sys, zip (>= 2.1.0)

Suggests spelling, knitr, rmarkdown, testthat (>= 3.0.0), roxygen2

VignetteBuilder knitr

Encoding UTF-8

SystemRequirements libgit2 (>= 1.0): libgit2-devel (rpm) or libgit2-dev (deb)

Language en-US

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

NeedsCompilation yes

Author Jeroen Ooms [aut, cre] (ORCID: <https://orcid.org/0000-0002-4035-0289>), Maëlle Salmon [aut] (ORCID: <https://orcid.org/0000-0002-2815-0399>), Jennifer Bryan [ctb] (ORCID: <https://orcid.org/0000-0002-6983-2759>)

Maintainer Jeroen Ooms <jeroenooms@gmail.com>

Config/pak/sysreqs libgit2-dev

Repository https://cran.r-universe.dev

Date/Publication 2026-10-14 12:56:35 UTC

RemoteUrl https://github.com/cran/gert

RemoteRef HEAD

RemoteSha 0aa2b4ffad0144e4ea2aabc5863e58adfc0f8d59

Contents

git_archive . . . 2
git_branch . . . 3
git_checkout_pull_request . . . 5
git_commit . . . 6
git_config . . . 8
git_diff . . . 11
git_fetch . . . 12
git_history . . . 14
git_ignore . . . 16
git_merge . . . 17
git_open . . . 18
git_rebase . . . 19
git_remote . . . 20
git_repo . . . 21
git_reset . . . 23
git_restore . . . 23
git_revert . . . 25
git_signature . . . 26
git_stash . . . 28
git_submodule_list . . . 29
git_tag . . . 30
git_worktree . . . 31
libgit2_config . . . 33
user_is_configured . . . 34

Index **35**

`git_archive` *Git Archive*

Description

Exports the files in your repository to a zip file that is returned by the function.

Usage

```
git_archive_zip(file = NULL, repo = ".")
```

Arguments

file	name of the output zip file. Default is returned by the function
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.

Value

path to the zip file that was created

See Also

Other git: git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

---

git_branch *Git Branch*

Description

Create, list, and checkout branches.

Usage

```
git_branch(repo = ".")

git_branch_list(local = NULL, repo = ".")

git_branch_checkout(branch, force = FALSE, orphan = FALSE, repo = ".")

git_branch_switch(branch, force = FALSE, orphan = FALSE, repo = ".")

git_branch_create(
  branch,
  ref = "HEAD",
  checkout = TRUE,
  force = FALSE,
  repo = "."
)

git_branch_delete(branch, repo = ".")

git_branch_move(branch, new_branch, force = FALSE, repo = ".")
```

```
git_branch_fast_forward(ref, repo = ".")

git_branch_set_upstream(upstream, branch = git_branch(repo), repo = ".")

git_branch_exists(branch, local = TRUE, repo = ".")
```

Arguments

repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
local	set TRUE to only check for local branches, FALSE to check for remote branches. Use NULL to return all branches.
branch	name of branch to check out
force	overwrite existing branch
orphan	if branch does not exist, checkout unborn branch
ref	string with a branch/tag/commit
checkout	move HEAD to the newly created branch
new_branch	target name of the branch once the move is performed; this name is validated for consistency.
upstream	remote branch from git_branch_list, for example "origin/master"

Related libgit2 documentation

branch, checkout.

See Also

Other git: git_archive, git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

Examples

```
# Creating a branch
repo <- file.path(tempdir(), "myrepo")
git_init(repo)
writeLines("hello", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
first_commit <- git_commit("First commit", repo = repo)

git_branch_create("new-feat", repo = repo)
writeLines("world", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
second_commit <- git_commit("Second commit", repo = repo)
```

```
# Changing branches
git_branch(repo = repo)
git_branch_switch("main", repo = repo)
git_branch(repo = repo)

git_branch_exists("new-feat", repo = repo)

# Listing branches
git_branch_list(repo = repo)

# Renaming a branch
git_branch_move("new-feat", "better-name", repo = repo)
git_branch_exists("new-feat", repo = repo)
git_branch_list(repo = repo)

# Deleting a branch
git_branch_delete("better-name", repo = repo)
git_branch_exists("better-name", repo = repo)

# clean up
unlink(repo, recursive = TRUE)
# ------------------------
# Creating an orphan branch
repo <- file.path(tempdir(), "myrepo")
git_init(repo)

# Set a user if no default
if (!user_is_configured()) {
  git_config_set("user.name", "Jerry")
  git_config_set("user.email", "jerry@gmail.com")
}
writeLines("hello", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
first_commit <- git_commit("First commit", repo = repo)

writeLines("world", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
second_commit <- git_commit("Second commit", repo = repo)

# Check out first commit
git_branch_checkout(first_commit, orphan = TRUE, repo = repo)
git_status(repo)
# clean up
unlink(repo, recursive = TRUE)
```

`git_checkout_pull_request` *GitHub Wrappers*

Description

Fetch and checkout pull requests.

Usage

```
git_checkout_pull_request(pr = 1, remote = NULL, repo = ".")

git_fetch_pull_requests(pr = "*", remote = NULL, repo = ".")
```

Arguments

pr	number with PR to fetch or check out. Use `"*"` to fetch all pull requests.
remote	Optional. Name of a remote listed in `git_remote_list()`. If unspecified and the current branch is already tracking branch a remote branch, that remote is honored. Otherwise, defaults to `origin`.
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.

Details

By default `git_fetch_pull_requests()` will download all PR branches. To remove these again simply use git_fetch(prune = TRUE).

---

git_commit *Stage and commit changes*

Description

To commit changes, start by *staging* the files to be included in the commit using `git_add()` or `git_rm()`. Use `git_status()` to see an overview of staged and unstaged changes, and finally `git_commit()` creates a new commit with currently staged files.

`git_commit_all()` is a convenience function that automatically stages and commits all modified files. Note that `git_commit_all()` does **not** add new, untracked files to the repository. You need to make an explicit call to `git_add()` to start tracking new files.

Usage

```
git_add(files, force = FALSE, repo = ".")

git_rm(files, repo = ".")

git_commit(message, author = NULL, committer = NULL, repo = ".")
```

```
git_commit_all(message, author = NULL, committer = NULL, repo = ".")

git_status(staged = NULL, pathspec = NULL, repo = ".")

git_ls(repo = ".", ref = NULL)
```

Arguments

files	vector of paths relative to the git root directory. Use "." to stage all changed files.
force	add files even if in gitignore
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
message	a commit message
author	A git_signature value, default is git_signature_default().
committer	A git_signature value, default is same as author
staged	return only staged (TRUE) or unstaged files (FALSE). Use NULL or NA to show both (default).
pathspec	character vector with paths to match
ref	revision string with a branch/tag/commit value

Value

- git_status(), git_ls(): A data frame with one row per file
- git_commit(), git_commit_all(): A SHA

Related libgit2 documentation

commit, index, status.

See Also

Other git: git_archive, git_branch(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

Examples

```
oldwd <- getwd()
repo <- file.path(tempdir(), "myrepo")
git_init(repo)
setwd(repo)

# Set a user if no default
```

```
if(!user_is_configured()){
  git_config_set("user.name", "Jerry")
  git_config_set("user.email", "jerry@gmail.com")
}

writeLines(letters[1:6], "alphabet.txt")
git_status()

git_add("alphabet.txt")
git_status()

git_commit("Start alphabet file")
git_status()

git_ls()

git_log()

cat(letters[7:9], file = "alphabet.txt", sep = "\n", append = TRUE)
git_status()

git_commit_all("Add more letters")

# cleanup
setwd(oldwd)
unlink(repo, recursive = TRUE)
```

`git_config` *Get or set Git configuration*

Description

Get, set, or unset Git options, as `git config` does on the command line. **Global** settings affect all of a user's Git operations (`git config --global`), whereas **local** settings are scoped to a specific repository (`git config --local`). When both exist, local options always win.

	local	**global**
get all	`git_config()`	`git_config_global()`
get one (local+global)	`git_config_get()`	`git_config_get()`
get one (local or global only)	`git_config_local_get()`	`git_config_global_get()`
set	`git_config_set()`	`git_config_global_set()`
unset	`git_config_unset()`	`git_config_global_unset()`

Usage

```
git_config(repo = ".")

git_config_global()
```

```
git_config_get(name, repo = ".")

git_config_local_get(name, repo = ".")

git_config_global_get(name)

git_config_set(name, value, add = FALSE, repo = ".")

git_config_global_set(name, value, add = FALSE)

git_config_unset(name, pattern = NULL, repo = ".")

git_config_global_unset(name, pattern = NULL)
```

Arguments

repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
name	Name of the option to get or set
value	Value to set. Must be a string, logical, number or NULL (to unset).
add	if TRUE, append a new entry for name instead of replacing existing one(s). Equivalent to git config --add. Only supported for string values.
pattern	optional regular expression, for matching values to unset in case of multiple values

Value

- git_config(): a data.frame of the Git options "in force" in the context of repo, one row per option. The level column reveals whether the option is determined from global or local config.
- git_config_global(): a data.frame, as for git_config(), except only for global Git options.
- git_config_get(): the value of the named option considering both local and global config (local wins), or NULL if unset.
- git_config_local_get(): as for git_config_get(), but restricted to local (repository-level) config only.
- git_config_global_get(): as for git_config_get(), but for global config only.
- git_config_set(), git_config_global_set(): The previous value(s) of name in local or global config, respectively. If this option was previously unset, returns NULL. Returns invisibly.
- git_config_unset(), git_config_global_unset(): The previous value(s) of name that were unset. Returns invisibly.

Related libgit2 documentation

config.

Note

All entries in the name column are automatically normalised to lowercase (see https://libgit2.org/libgit2/#HEAD/type/git_config_entry for details).

See Also

Other git: git_archive, git_branch(), git_commit(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

Examples

```
# Set and inspect a local, custom Git option
r <- file.path(tempdir(), "gert-demo")
git_init(r)

previous <- git_config_set("aaa.bbb", "ccc", repo = r)
previous
git_config_local_get("aaa.bbb", repo = r)

previous <- git_config_set("aaa.bbb", NULL, repo = r)
previous
git_config_local_get("aaa.bbb", repo = r)

# Get a single named option (returns NULL if unset)
git_config_get("aaa.bbb", repo = r)
git_config_set("aaa.bbb", "ccc", repo = r)
git_config_get("aaa.bbb", repo = r)

unlink(r, recursive = TRUE)

## Not run:
# Set global Git options
git_config_global_set("user.name", "Your Name")
git_config_global_set("user.email", "your@email.com")
git_config_global()

# Get a single global option (returns NULL if unset)
git_config_global_get("user.name")
git_config_global_get("gert.nonexistent")

## End(Not run)
```

`git_diff` *Git Diff*

Description

View changes in a commit or in the current working directory.

- `git_diff()` returns a data frame with information about a commit patch.
- `git_diff_patch()` is shortcode for `git_diff()$patch`.

Usage

```
git_diff(ref = NULL, repo = ".")

git_diff_patch(ref = NULL, repo = ".")
```

Arguments

`ref`	a reference such as `"HEAD"`, or a commit id, or `NULL` to the diff the working directory against the repository index.
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.

Value

- `git_diff()` returns a data frame.
- `git_diff_patch()` returns a character vector.

Related libgit2 documentation

`diff`.

See Also

Other git: `git_archive`, `git_branch()`, `git_commit()`, `git_config()`, `git_fetch()`, `git_history`, `git_ignore`, `git_merge()`, `git_rebase()`, `git_remote`, `git_repo`, `git_reset()`, `git_restore()`, `git_revert()`, `git_signature()`, `git_stash`, `git_tag`, `git_worktree`

`git_fetch` *Push and pull*

Description

Functions to connect with a git server (remote) to fetch or push changes. The 'credentials' package is used to handle authentication, the credentials vignette explains the various authentication methods for SSH and HTTPS remotes.

Usage

```
git_fetch(
  remote = NULL,
  refspec = NULL,
  password = askpass,
  ssh_key = NULL,
  prune = FALSE,
  verbose = interactive(),
  repo = "."
)

git_remote_ls(
  remote = NULL,
  password = askpass,
  ssh_key = NULL,
  verbose = interactive(),
  repo = "."
)

git_push(
  remote = NULL,
  refspec = NULL,
  set_upstream = NULL,
  password = askpass,
  ssh_key = NULL,
  mirror = FALSE,
  force = FALSE,
  verbose = interactive(),
  repo = "."
)

git_clone(
  url,
  path = NULL,
  branch = NULL,
  password = askpass,
  ssh_key = NULL,
```

```
  bare = FALSE,
  mirror = FALSE,
  depth = 0,
  verbose = interactive()
)

git_pull(remote = NULL, rebase = FALSE, ..., repo = ".")
```

Arguments

remote	Optional. Name of a remote listed in git_remote_list(). If unspecified and the current branch is already tracking branch a remote branch, that remote is honored. Otherwise, defaults to origin.
refspec	string with mapping between remote and local refs. Default uses the default refspec from the remote, which usually fetches all branches.
password	a string or a callback function to get passwords for authentication or password protected ssh keys. Defaults to askpass which checks getOption('askpass').
ssh_key	path or object containing your ssh private key. By default we look for keys in ssh-agent and credentials::ssh_key_info().
prune	delete tracking branches that no longer exist on the remote, or are not in the refspec (such as pull requests).
verbose	display some progress info while downloading
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
set_upstream	change the branch default upstream to remote. If NULL, this will set the branch upstream only if the push was successful and if the branch does not have an upstream set yet.
mirror	use the --mirror flag
force	use the --force flag
url	remote url. Typically starts with https://github.com/ for public repositories, and https://yourname@github.com/ or git@github.com/ for private repos. You will be prompted for a password or pat when needed.
path	Directory of the Git repository to create. By default, the "humanish" part of the URL. For instance, "git@github.com:someone/myrepo.git" will be cloned to myrepo/.
branch	name of branch to check out locally
bare	use the --bare flag
depth	number of commits to fetch when creating a shallow clone. The default 0 clones the entire history. A positive value truncates the history to the given number of commits from the tip of each branch. Requires libgit2 >= 1.7.0.
rebase	if TRUE we try to rebase instead of merge local changes. This is not possible in case of conflicts (you will get an error).
...	arguments passed to git_fetch()

Details

Use git_fetch() and git_push() to sync a local branch with a remote branch. Here git_pull() is a wrapper for git_fetch() which then tries to fast-forward the local branch after fetching.

Related libgit2 documentation

remote, clone.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

Examples

```
{# Clone a small repository
git_dir <- file.path(tempdir(), 'antiword')
git_clone('https://github.com/ropensci/antiword', git_dir)

# Change into the repo directory
olddir <- getwd()
setwd(git_dir)

# Show some stuff
git_log()
git_branch_list()
git_remote_list()

# Add a file
write.csv(iris, 'iris.csv')
git_add('iris.csv')

# Commit the change
jerry <- git_signature("Jerry", "jerry@hotmail.com")
git_commit('added the iris file', author = jerry)

# Now in the log:
git_log()

# Cleanup
setwd(olddir)
unlink(git_dir, recursive = TRUE)
}
```

---

git_history *View commit history*

Description

- `git_commit_stats()` returns information about commit insertion and deletion
- `git_commit_info()` a list of commit info
- `git_commit_id()` is a shortcut for `git_commit_info()$id`
- `git_log()` shows the most recent commits
- `git_ls()` lists all the files that are being tracked in the repository.
- `git_stat_files()` shows information of when `files` was last modified.

Usage

```
git_commit_info(ref = "HEAD", repo = ".")

git_commit_id(ref = "HEAD", repo = ".")

git_commit_stats(ref = "HEAD", repo = ".")

git_log(ref = "HEAD", max = 100, after = NULL, path = NULL, repo = ".")

git_stat_files(files, ref = "HEAD", max = NULL, repo = ".")
```

Arguments

`ref`	revision string with a branch/tag/commit value
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.
`max`	lookup at most latest n parent commits
`after`	date or timestamp: only include commits starting this date
`path`	character vector with paths to filter on; only commits that touch these paths are included
`files`	vector of paths relative to the git root directory. Use `"."` to stage all changed files.

Value

- `git_commit_info()` and `git_commit_stats()` return a list.

Related libgit2 documentation

`commit`.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

---

`git_ignore` *Git Ignore*

Description

Test if files would be ignored by `.gitignore` rules

Usage

```
git_ignore_path_is_ignored(path, repo = ".")
```

Arguments

`path`	A character vector of paths to test within the repo
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.

Value

A logical vector the same length as `path`, indicating if the paths would be ignored.

Related libgit2 documentation

ignore.

See Also

`usethis::use_git_ignore()` to add a path to `.gitignore`.

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

`git_merge` *Git merge*

Description

Use `git_merge()` to merge a branch into the current head. Based on how the branches have diverged, the function will select a fast-forward or merge-commit strategy.

Other functions are more low-level tools that are used by `git_merge()`:

- `git_merge_find_base()` looks up the commit where two branches have diverged (i.e. the youngest common ancestor).
- `git_merge_analysis()` is used to test if a merge can simply be fast forwarded or not.
- `git_merge_stage_only()` applies and stages changes, without committing or fast-forwarding.
- `git_conflicts()` lists merge conflicts.

Usage

```
git_merge(ref, commit = TRUE, squash = FALSE, repo = ".")

git_merge_stage_only(ref, squash = FALSE, repo = ".")

git_merge_find_base(ref, target = "HEAD", repo = ".")

git_merge_analysis(ref, repo = ".")

git_merge_abort(repo = ".")

git_commit_descendant_of(ancestor, ref = "HEAD", repo = ".")

git_conflicts(repo = ".")
```

Arguments

`ref`	branch or commit that you want to merge
`commit`	automatically create a merge commit if the merge succeeds without conflicts. Set this to `FALSE` if you want to customize your commit message/author.
`squash`	omits the second parent from the commit, which make the merge a regular single-parent commit.
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.
`target`	the branch where you want to merge into. Defaults to current `HEAD`.
`ancestor`	a reference to a potential ancestor commit

Details

By default `git_merge()` automatically commits the merge commit upon success. However if the merge fails with merge-conflicts, or if `commit` is set to `FALSE`, the changes are staged and the repository is put in merging state, and you have to manually run `git_commit()` or `git_merge_abort()` to proceed.

Related libgit2 documentation

`merge`.

See Also

Other git: `git_archive`, `git_branch()`, `git_commit()`, `git_config()`, `git_diff()`, `git_fetch()`, `git_history`, `git_ignore`, `git_rebase()`, `git_remote`, `git_repo`, `git_reset()`, `git_restore()`, `git_revert()`, `git_signature()`, `git_stash`, `git_tag`, `git_worktree`

`git_open` *Open local repository*

Description

Returns a pointer to a libgit2 repository object. This function is mainly for internal use; users should simply reference a repository in gert by by the path to the directory.

Usage

```
git_open(repo = ".")
```

Arguments

`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.

Value

an pointer to the libgit2 repository

Examples

```
r <- tempfile(pattern = "gert")
git_init(r)
r_ptr <- git_open(r)
r_ptr
git_open(r_ptr)
git_info(r)
```

```
# cleanup
unlink(r, recursive = TRUE)
```

`git_rebase` *Cherry-Pick and Rebase*

Description

- `git_cherry_pick()` applies the changes from a given commit (from another branch) onto the current branch. *`git_rebase_commit()` resets the branch to the state of another branch (upstream) and then re-applies your local changes by cherry-picking each of your local commits onto the upstream commit history. *`git_rebase_list()` shows your local commits that are missing from the `upstream` history, and if they conflict with upstream changes.

Usage

```
git_rebase_list(upstream = NULL, repo = ".")

git_rebase_commit(upstream = NULL, repo = ".")

git_cherry_pick(commit, repo = ".")

git_ahead_behind(upstream = NULL, ref = "HEAD", repo = ".")
```

Arguments

`upstream`	branch to which you want to rewind and re-apply your local commits. The default uses the remote upstream branch with the current state on the git server, simulating `git_pull()`.
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.
`commit`	id of the commit to cherry pick
`ref`	string with a branch/tag/commit

Details

To find if your local commits are missing from `upstream`, `git_rebase_list()` first performs a rebase dry-run, without committing anything. If there are no conflicts, you can use `git_rebase_commit()` to rewind and rebase your branch onto `upstream`.

Gert only support a clean rebase; it never leaves the repository in unfinished "rebasing" state. If conflicts arise, `git_rebase_commit()` will raise an error without making changes.

Related libgit2 documentation

rebase, cherrypick, graph.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

---

git_remote *Git Remotes*

Description

List, add, configure, or remove remotes.

Usage

```
git_remote_list(repo = ".")

git_remote_add(url, name = "origin", refspec = NULL, repo = ".")

git_remote_remove(remote, repo = ".")

git_remote_info(remote = NULL, repo = ".")

git_remote_set_url(url, remote = NULL, repo = ".")

git_remote_set_pushurl(url, remote = NULL, add = FALSE, repo = ".")

git_remote_refspecs(remote = NULL, repo = ".")
```

Arguments

repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
url	server url (https or ssh)
name	unique name for the new remote
refspec	optional string with the remote fetch value
remote	name of an existing remote. Default NULL means the remote from the upstream of the current branch.
add	if TRUE, append the push URL instead of replacing it. Equivalent to git remote set-url --push --add.

Related libgit2 documentation

remote.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

---

git_repo *Create or discover a local Git repository*

Description

- git_init() creates a new repository
- git_find() to discover an existing local repository.
- git_info() shows basic information about a repository, such as the SHA and branch of the current HEAD.

Usage

```
git_init(path = ".", bare = FALSE)

git_find(path = ".")

git_info(repo = ".")
```

Arguments

path	the location of the git repository, see the "path" section.
bare	if true, a Git repository without a working directory is created
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.

Value

- git_find() and git_init(): the path to the Git repository.
- git_info(): A list of information of the Git repository.

Path

For `git_init()` the `path` parameter sets the directory of the git repository to create. If this directory already exists, it must be empty. If it does not exist, it is created, along with any intermediate directories that don't yet exist.

For `git_find()`, the `path` parameter specifies the directory at which to start the search for a git repository. If it is not a git repository itself, then its parent directory is consulted, then the parent's parent, and so on.

Detached head

If `git_info()$shorthand` is equal to `HEAD`, it means the repository is in a detached head state.

Related libgit2 documentation

`repository`.

See Also

Other git: `git_archive`, `git_branch()`, `git_commit()`, `git_config()`, `git_diff()`, `git_fetch()`, `git_history`, `git_ignore`, `git_merge()`, `git_rebase()`, `git_remote`, `git_reset()`, `git_restore()`, `git_revert()`, `git_signature()`, `git_stash`, `git_tag`, `git_worktree`

Examples

```
# directory does not yet exist
r <- tempfile(pattern = "gert")
git_init(r)
git_find(r)
git_info(r)

# create a child directory, then a grandchild, then search
r_grandchild_dir <- file.path(r, "aaa", "bbb")
dir.create(r_grandchild_dir, recursive = TRUE)
git_find(r_grandchild_dir)

# cleanup
unlink(r, recursive = TRUE)

# directory exists but is empty
r <- tempfile(pattern = "gert")
dir.create(r)
git_init(r)
git_find(r)

# cleanup
unlink(r, recursive = TRUE)
```

git_reset *Reset your repo to a previous state*

Description

- `git_reset_hard()` resets the index and working tree
- `git_reset_soft()` does not touch the index file or the working tree
- `git_reset_mixed()` resets the index but not the working tree.

Usage

```
git_reset_hard(ref = "HEAD", repo = ".")

git_reset_soft(ref = "HEAD", repo = ".")

git_reset_mixed(ref = "HEAD", repo = ".")
```

Arguments

`ref`	string with a branch/tag/commit
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.

Related libgit2 documentation

`reset`.

See Also

Other git: `git_archive`, `git_branch()`, `git_commit()`, `git_config()`, `git_diff()`, `git_fetch()`, `git_history`, `git_ignore`, `git_merge()`, `git_rebase()`, `git_remote`, `git_repo`, `git_restore()`, `git_revert()`, `git_signature()`, `git_stash`, `git_tag`, `git_worktree`

git_restore *Restore working tree files*

Description

Restores specified paths in the working tree from a given ref, equivalent to `git restore --source=<ref> <path>` (or the older `git checkout <ref> -- <path>`). The ref must be reachable from the current HEAD.
By default restores from HEAD, discarding any local modifications.

Usage

```
git_restore(path, ref = "HEAD", repo = ".")
```

Arguments

path	character vector with file paths to restore, relative to the repository root. Use "." to restore all tracked files.
ref	revision string with a branch/tag/commit to restore from. Defaults to "HEAD".
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.

Value

Invisibly, the git_status() after restoring.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_revert(), git_signature(), git_stash, git_tag, git_worktree

Examples

```
repo <- file.path(tempdir(), "myrepo")
git_init(repo)

# Set a user if no default
if (!user_is_configured()) {
  git_config_set("user.name", "Jerry")
  git_config_set("user.email", "jerry@gmail.com")
}

writeLines("hello", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
git_commit("First commit", repo = repo)

# Modify the file, then restore it from HEAD
writeLines("oops", file.path(repo, "hello.txt"))
git_restore("hello.txt", repo = repo)
readLines(file.path(repo, "hello.txt"))  # "hello"

unlink(repo, recursive = TRUE)
```

`git_revert` *Revert a commit*

Description

Applies the inverse of the changes introduced by a given commit, equivalent to `git revert <commit>`. The commit must be reachable from the current HEAD.

Usage

```
git_revert(ref, commit = TRUE, ..., repo = ".")
```

Arguments

`ref`	revision string with a branch/tag/commit value
`commit`	if `FALSE`, stage the reverted changes without creating a commit. Default is `TRUE`, that is to say, by default a commit is made.
`...`	parameters passed to `git_commit` such as `message` or `author`
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.

Details

By default, a new revert commit is created immediately. Set `commit = FALSE` to only stage the reverted changes without committing, leaving you free to amend or combine them before calling `git_commit()`.

Value

The SHA of the new revert commit (invisibly), or `NULL` when `commit = FALSE`.

Related libgit2 documentation

`revert`.

See Also

Other git: `git_archive`, `git_branch()`, `git_commit()`, `git_config()`, `git_diff()`, `git_fetch()`, `git_history`, `git_ignore`, `git_merge()`, `git_rebase()`, `git_remote`, `git_repo`, `git_reset()`, `git_restore()`, `git_signature()`, `git_stash`, `git_tag`, `git_worktree`

Examples

```
repo <- file.path(tempdir(), "myrepo")
git_init(repo)

# Set a user if no default
if (!user_is_configured()) {
  git_config_set("user.name", "Jerry")
  git_config_set("user.email", "jerry@gmail.com")
}

writeLines("hello", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
git_commit("First commit", repo = repo)

writeLines("world", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
bad_commit <- git_commit("Second commit", repo = repo)

# Default: revert and commit with an auto-generated message
git_revert(bad_commit, repo = repo)
git_log(repo = repo)

# Revert with a custom message
writeLines("oops", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
bad_commit2 <- git_commit("Third commit", repo = repo)
git_revert(bad_commit2, message = "Undo third commit\n", repo = repo)
git_log(repo = repo)

# Stage the revert without committing
writeLines("again", file.path(repo, "hello.txt"))
git_add("hello.txt", repo = repo)
bad_commit3 <- git_commit("Fourth commit", repo = repo)
git_revert(bad_commit3, commit = FALSE, repo = repo)
git_status(repo = repo)

unlink(repo, recursive = TRUE)
```

---

git_signature *Author Signature*

Description

A signature contains the author and timestamp of a commit. Each commit includes a signature of the author and committer (which can be identical).

Usage

```
git_signature_default(repo = ".")

git_signature(name, email, time = NULL)

git_signature_parse(sig)
```

Arguments

repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
name	Real name of the committer
email	Email address of the committer
time	timestamp of class POSIXt or NULL
sig	string in proper "First Last <your@email.com>" format, see details.

Details

A signature string has format "Real Name <email> timestamp tzoffset". The timestamp tzoffset piece can be omitted in which case the current local time is used. If not omitted, timestamp must contain the number of seconds since the Unix epoch and tzoffset is the timezone offset in hhmm format (note the lack of a colon separator)

Related libgit2 documentation

signature.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_stash, git_tag, git_worktree

Examples

```
# Your default user
try(git_signature_default())

# Specify explicit name and email
git_signature("Some committer", "sarah@gmail.com")

# Create signature for an hour ago
(sig <- git_signature("Han", "han@company.com", Sys.time() - 3600))

# Parse a signature
git_signature_parse(sig)
```

```
git_signature_parse("Emma <emma@mu.edu>")
```

`git_stash` *Stashing changes*

Description

Temporary stash away changed from the working directory.

Usage

```
git_stash_save(
  message = "",
  keep_index = FALSE,
  include_untracked = FALSE,
  include_ignored = FALSE,
  repo = "."
)

git_stash_pop(index = 0, repo = ".")

git_stash_drop(index = 0, repo = ".")

git_stash_list(repo = ".")
```

Arguments

`message`	optional message to store the stash
`keep_index`	changes already added to the index are left intact in the working directory
`include_untracked`	untracked files are also stashed and then cleaned up from the working directory
`include_ignored`	ignored files are also stashed and then cleaned up from the working directory
`repo`	The path to the git repository. If the directory is not a repository, parent directories are considered (see `git_find()`). To disable this search, provide the filepath protected with `I()`. When using this parameter, always explicitly call by name (i.e. `repo = `) because future versions of gert may have additional parameters.
`index`	The position within the stash list. 0 points to the most recent stashed state.

Related libgit2 documentation

`stash`.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_tag, git_worktree

---

`git_submodule_list` *Submodules*

Description

Interact with submodules in the repository.

Usage

```
git_submodule_list(repo = ".")

git_submodule_info(submodule, repo = ".")

git_submodule_init(submodule, overwrite = FALSE, repo = ".")

git_submodule_set_to(submodule, ref, checkout = TRUE, repo = ".")

git_submodule_add(url, path = basename(url), ref = "HEAD", ..., repo = ".")

git_submodule_fetch(submodule, ..., repo = ".")
```

Arguments

repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
submodule	name of the submodule
overwrite	overwrite existing entries
ref	a branch or tag or hash with
checkout	actually switch the contents of the directory to this commit
url	full git url of the submodule
path	relative of the submodule
...	extra arguments for git_fetch() for authentication things

Related libgit2 documentation

submodule.

git_tag *Git Tag*

Description

Create and list tags.

Usage

```
git_tag_list(match = "*", repo = ".")

git_tag_create(name, message, ref = "HEAD", repo = ".")

git_tag_delete(name, repo = ".")

git_tag_push(name, ..., repo = ".")
```

Arguments

match	pattern to filter tags (use * for wildcard)
repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
name	tag name
message	tag message
ref	target reference to tag
...	other arguments passed to git_push()

Related libgit2 documentation

tag.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_worktree

`git_worktree` *Git Worktrees*

Description

Worktrees represent an alternative location to checkout a branch into. Rather than checking out a branch in your main working tree (which changes the branch you are currently on and forces you to stash any existing work), you can instead check that branch out into a separate *linked worktree* with its own working tree. Practically, a worktree is just a separate folder that a branch is checked out into, with some extra git metadata that links it back to the main working tree.

`git_worktree_list()` returns a data frame of information about the worktrees linked to the main working tree.

`git_worktree_exists()` lets you check whether or not a worktree by the name of `name` exists for this `repo`.

`git_worktree_path()` returns the file path to the worktree.

`git_worktree_add()` creates a new worktree called `name` in the folder pointed to by `path`, and checks `branch` out into it.

`git_worktree_remove()` removes a worktree. It does so by deleting the folder provided as the `path` to `git_worktree_add()`, and then cleaning up some git metadata in the main working tree that linked the main working tree to the removed worktree. The `branch` checked out by the worktree is not deleted. Note that this is just a wrapper around `git_worktree_prune()` that sets some desirable defaults for aggressive removal.

`git_worktree_prune()` is more cautious than `git_worktree_remove()`. It refuses to prune *valid* or *locked* worktrees by default, and also refuses the delete the working tree of the worktree by default (i.e. the folder at `path`). It is automatically run by git itself on periodic intervals to prune outdated worktrees. For interactive usage, you typically want `git_worktree_remove()` instead. `git_worktree_is_prunable()` lets you check if a worktree is prunable with the given options.

`git_worktree_lock()`, `git_worktree_unlock()`, and `git_worktree_is_locked()` help you manage whether or not a worktree is *locked*. When a worktree is locked, it is not automatically cleaned up by `git_worktree_prune()` (and git itself) on periodic intervals, even when it looks *invalid*. This is typically only useful when your worktree is on a hard drive that isn't always connected (which can make it look *invalid* when disconnected, typically making it a candidate for automatic pruning).

`git_worktree_is_valid()` checks whether a worktree is valid or not. A *valid* worktree requires both the git data structures inside the main working tree and this worktree to be present.

Usage

```
git_worktree_list(repo = ".")

git_worktree_exists(name, repo = ".")

git_worktree_path(name, repo = ".")

git_worktree_add(name, path, branch, lock = FALSE, local = TRUE, repo = ".")
```

```
git_worktree_remove(name, repo = ".")

git_worktree_prune(
  name,
  prune_valid = FALSE,
  prune_locked = FALSE,
  prune_working_tree = FALSE,
  repo = "."
)

git_worktree_is_prunable(
  name,
  prune_valid = FALSE,
  prune_locked = FALSE,
  repo = "."
)

git_worktree_lock(name, repo = ".")

git_worktree_unlock(name, repo = ".")

git_worktree_is_locked(name, repo = ".")

git_worktree_is_valid(name, repo = ".")
```

Arguments

repo	The path to the git repository. If the directory is not a repository, parent directories are considered (see git_find()). To disable this search, provide the filepath protected with I(). When using this parameter, always explicitly call by name (i.e. repo =) because future versions of gert may have additional parameters.
name	The name of the worktree.
path	The path to checkout branch into. Importantly, the path up to the folder name must exist, but the folder name itself must not exist yet and will be created.
branch	The branch to checkout into path.
lock	Whether or not to lock the worktree on creation.
local	set TRUE to only check for local branches, FALSE to check for remote branches. Use NULL to return all branches.
prune_valid	Whether or not to forcibly prune a *valid* worktree.
prune_locked	Whether or not to forcibly prune a *locked* worktree.
prune_working_tree	Whether or not to also remove the folder that the worktree was using, i.e. the path supplied to git_worktree_add().

Related libgit2 documentation

worktree.

See Also

Other git: git_archive, git_branch(), git_commit(), git_config(), git_diff(), git_fetch(), git_history, git_ignore, git_merge(), git_rebase(), git_remote, git_repo, git_reset(), git_restore(), git_revert(), git_signature(), git_stash, git_tag

Examples

```
repo <- git_init(tempfile("gert-examples-repo"))

writeLines("hello", file.path(repo, 'hello.txt'))
git_add('hello.txt', repo = repo)
git_commit("First commit", author = "jeroen <jeroen@blabla.nl>", repo = repo)

# Create a branch that is going to be used for the worktree,
# but don't check it out!
git_branch_create(branch = "branch", checkout = FALSE, repo = repo)

path <- tempfile("gert-examples-worktree")

# Add a worktree for this branch
git_worktree_add(
  name = "worktree",
  path = path,
  branch = "branch",
  repo = repo
)

# Worktree info
git_worktree_list(repo = repo)

# Note how the files are checked out here
dir(path, all.files = TRUE)

# And the branch that we are on at `path` is `"branch"`
git_branch(repo = path)

# Cleanup worktree, and the folder at `path`
git_worktree_remove("worktree", repo = repo)

# Cleanup repo
unlink(repo, recursive = TRUE)
```

---

libgit2_config *Show libgit2 version and capabilities*

Description

`libgit2_config()` reveals which version of libgit2 gert is using and which features are supported, such whether you are able to use ssh remotes.

Usage

```
libgit2_config()
```

Examples

```
libgit2_config()
```

`user_is_configured` *Test if a Git user is configured*

Description

This function exists mostly to guard examples that rely on having a user configured, in order to make commits. `user_is_configured()` makes no distinction between local or global user config.

Usage

```
user_is_configured(repo = ".")
```

Arguments

`repo`	An optional `repo`, in the sense of `git_open()`.

Value

`TRUE` if `user.name` and `user.email` are set locally or globally, `FALSE` otherwise.

Examples

```
user_is_configured()
```

Index

∗ **git**
git_archive, 2
git_branch, 3
git_commit, 6
git_config, 8
git_diff, 11
git_fetch, 12
git_history, 14
git_ignore, 16
git_merge, 17
git_rebase, 19
git_remote, 20
git_repo, 21
git_reset, 23
git_restore, 23
git_revert, 25
git_signature, 26
git_stash, 28
git_tag, 30
git_worktree, 31

askpass, *13*

credentials::ssh_key_info(), *13*

fast-forward, *14*

git_add *(*git_commit*)*, 6
git_ahead_behind *(*git_rebase*)*, 19
git_archive, 2, *4*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_archive_zip *(*git_archive*)*, 2
git_branch, 3
git_branch(), *3*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_branch_checkout *(*git_branch*)*, 3
git_branch_create *(*git_branch*)*, 3
git_branch_delete *(*git_branch*)*, 3
git_branch_exists *(*git_branch*)*, 3
git_branch_fast_forward *(*git_branch*)*, 3
git_branch_list, *4*
git_branch_list *(*git_branch*)*, 3
git_branch_move *(*git_branch*)*, 3
git_branch_set_upstream *(*git_branch*)*, 3
git_branch_switch *(*git_branch*)*, 3
git_checkout_pull_request, 5
git_cherry_pick *(*git_rebase*)*, 19
git_clone *(*git_fetch*)*, 12
git_commit, 6
git_commit(), *3, 4*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_commit_all *(*git_commit*)*, 6
git_commit_descendant_of *(*git_merge*)*, 17
git_commit_id *(*git_history*)*, 14
git_commit_info *(*git_history*)*, 14
git_commit_stats *(*git_history*)*, 14
git_config, 8
git_config(), *3, 4*, *7*, *11*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_config_get *(*git_config*)*, 8
git_config_global *(*git_config*)*, 8
git_config_global_get *(*git_config*)*, 8
git_config_global_set *(*git_config*)*, 8
git_config_global_unset *(*git_config*)*, 8
git_config_local_get *(*git_config*)*, 8
git_config_set *(*git_config*)*, 8
git_config_unset *(*git_config*)*, 8
git_conflicts *(*git_merge*)*, 17
git_diff, 11
git_diff(), *3, 4*, *7*, *10*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_diff_patch *(*git_diff*)*, 11
git_fetch, 12
git_fetch(), *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_fetch(prune = TRUE), *6*
git_fetch_pull_requests *(*git_checkout_pull_request*)*, 5
git_find *(*git_repo*)*, 21

git_find(), *3, 4*, *6, 7*, *9*, *11*, *13*, *15–21*, *23–25*, *27–30*, *32*
git_history, *3, 4*, *7*, *10, 11*, 14, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_ignore, *3, 4*, *7*, *10, 11*, *14*, 16, *16*, *18*, *20–25*, *27*, *29, 30*, *33*
git_ignore_path_is_ignored *(git_ignore)*, 16
git_info *(git_repo)*, 21
git_init *(git_repo)*, 21
git_log *(git_history)*, 14
git_ls *(git_commit)*, 6
git_merge, 17
git_merge(), *3, 4*, *7*, *10, 11*, *14*, *16*, *20–25*, *27*, *29, 30*, *33*
git_merge_abort *(git_merge)*, 17
git_merge_analysis *(git_merge)*, 17
git_merge_find_base *(git_merge)*, 17
git_merge_stage_only *(git_merge)*, 17
git_open, 18
git_open(), *34*
git_pull *(git_fetch)*, 12
git_pull(), *14*, *19*
git_push *(git_fetch)*, 12
git_push(), *14*, *30*
git_rebase, 19
git_rebase(), *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *21–25*, *27*, *29, 30*, *33*
git_rebase_commit *(git_rebase)*, 19
git_rebase_list *(git_rebase)*, 19
git_remote, *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, 20, *20*, *22–25*, *27*, *29, 30*, *33*
git_remote_add *(git_remote)*, 20
git_remote_info *(git_remote)*, 20
git_remote_list *(git_remote)*, 20
git_remote_list(), *6*, *13*
git_remote_ls *(git_fetch)*, 12
git_remote_refspecs *(git_remote)*, 20
git_remote_remove *(git_remote)*, 20
git_remote_set_pushurl *(git_remote)*, 20
git_remote_set_url *(git_remote)*, 20
git_repo, *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20*, 21, *21*, *23–25*, *27*, *29, 30*, *33*
git_reset, 23
git_reset(), *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–22*, *24, 25*, *27*, *29, 30*, *33*
git_reset_hard *(git_reset)*, 23
git_reset_mixed *(git_reset)*, 23
git_reset_soft *(git_reset)*, 23
git_restore, 23
git_restore(), *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–23*, *25*, *27*, *29, 30*, *33*
git_revert, 25
git_revert(), *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–24*, *27*, *29, 30*, *33*
git_rm *(git_commit)*, 6
git_signature, *7*, 26
git_signature(), *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *29, 30*, *33*
git_signature_default *(git_signature)*, 26
git_signature_default(), *7*
git_signature_parse *(git_signature)*, 26
git_stash, *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, 28, *30*, *33*
git_stash_drop *(git_stash)*, 28
git_stash_list *(git_stash)*, 28
git_stash_pop *(git_stash)*, 28
git_stash_save *(git_stash)*, 28
git_stat_files *(git_history)*, 14
git_status *(git_commit)*, 6
git_status(), *24*
git_submodule_add *(git_submodule_list)*, 29
git_submodule_fetch *(git_submodule_list)*, 29
git_submodule_info *(git_submodule_list)*, 29
git_submodule_init *(git_submodule_list)*, 29
git_submodule_list, 29
git_submodule_set_to *(git_submodule_list)*, 29
git_tag, *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, *29*, 30, *33*
git_tag_create *(git_tag)*, 30
git_tag_delete *(git_tag)*, 30
git_tag_list *(git_tag)*, 30
git_tag_push *(git_tag)*, 30
git_worktree, *3, 4*, *7*, *10, 11*, *14*, *16*, *18*, *20–25*, *27*, *29, 30*, 31
git_worktree_add *(git_worktree)*, 31
git_worktree_exists *(git_worktree)*, 31
git_worktree_is_locked *(git_worktree)*, 31
git_worktree_is_prunable

(git_worktree), 31
git_worktree_is_valid (git_worktree), 31
git_worktree_list (git_worktree), 31
git_worktree_lock (git_worktree), 31
git_worktree_path (git_worktree), 31
git_worktree_prune (git_worktree), 31
git_worktree_remove (git_worktree), 31
git_worktree_unlock (git_worktree), 31

I(), 3, 4, 6, 7, 9, 11, 13, 15–21, 23–25, 27–30, 32

libgit2_config, 33

user_is_configured, 34