Git

Jump to bottom

Kamil Aliev edited this page on Mar 5 · 9 revisions

Git

Contents

- Get started
  - Setup
  - Help
- Git basics
  - Getting a git repository
  - Recording changes to the repository
  - Viewing the commit history
  - Limiting log output
  - Undoing things
  - Working with remotes
  - Tagging
  - Git Aliases
- Branching
  - Branches in a nutshell
  - Basic branching and merging
  - Branch management
  - Remote branches
  - Rebasing
- Git tools
  - Revision selection
  - Interactive staging

    - Stashing and cleaning
    - Searching
    - Rewriting history
    - Reset demystified
    - Reflog
- Patch workflow
- Plumbing
- Extra
    - Tips and tricks
    - Useful commands
- References

Git get started

Popular Distributed Version Control Systems (DVCS) - Git, Mercurial, Bazaar, Darcs.

In comparison to other VCSs, which use delta-based version control (have a base file and each version is a difference made to that file), git makes a snapshot of all files on every version. If file has not changed, stores a reference a link to previous identical file it has already stored.

Git uses SHA-1 hash mechanism for checksumming for all kind of operations as file change detection, storing, etc.

Three main states that a file can have:

- modified - file modified but not committed
- staged - marked modified file to go into next commit snapshot
- committed - data is stored in local database

Installing git

Setup

Configuration variables that control all aspects of Git can be stored in three different places:

- `/etc/gitconfig` file - values are applied to every user on the system and all their repositories; pass **--system** option to **git config** to operate on values from this file specifically

- `~/.gitconfig` or `~/.config/git/config` file - values specific to user; pass **--global** option to **git config** to operate on this file, affects all repositories of the user
- `.git/config` (file in the repository) - specific to single repository; **--local** option is the default, should be located in corresponding repository

Set identity(this information is added to every commit):

- `git config --global user.name <name>`
- `git config --global user.email <email>`

Git uses specified editor when user needs to type in a message. If not configured, system's default is used:

- `git config --global core.editor vim`

Checking settings:

- `git config --list` - view all settings
- `git config <setting>` - check specific setting
- `git config --show-origin <setting>` - show setting as well as where Git took it from
- `git config --list --show-origin` - view all settings and there they are coming from
- `git config --edit` - edit values in configured editor

Turn on autocompletion in bash:

- move `git-completion.bash` file to home directory
- source it to bashrc with the following line - `. ~/git-completion.bash`

Documentation (also includes list of available options)

Help

Commands to get man pages:

- `git help <verb>`
- `git <verb> --help`
- `man git-<verb>`
- `git <verb> -h` - concise *help* output, refresher

Git basics

Getting a git repository

git init - creates subdirectory (.git), repository skeleton

- **--bare** - creates a bare repository without working directory, thus, can't be used for development, but rather used as storage facility; has same contents as .git subdirectory

`git clone <repo_path> [directory]` - copy existing repository to local machine; optional *directory* argument specifies where repository will be created

- `[--origin], [-o] <remote>` - give other name to refer to remote, other than the default name `origin`

Recording changes to the repository

commit style advice

tips for useful commits

git status - determine status of all files in the current repository, can be tracked (unmodified | modified | stages) or untracked

- `[--short], [-s]` - short version; 1 column is staged area, 2 column - working tree; corresponding signs are:
  - **?** - new files
  - **A** - added to staged area
  - **M** - modified files

git add - adding files to the next commit (new | modified | resolve conflicts)

`.gitignore` - add files or patterns to ignore, examples; it is possible to have multiple gitignores in subdirectories, which will only apply to files inside that directory. `.git/info/exclude` file can be used to specify which files to ignore in current repository, without sharing it with remote.

- start pattern with slash (/) to avoid recursivity
  - `/TODO` - only ignore TODO file in the current directory, not subdir/TODO
- end pattern with slash (/) to specify a directory
  - `build/` - ignore all files in any directory named build
- negate pattern by starting it with an exclamation point (!)

    - `*.a` - all files in a repository, including in nested directories
    - `!lib.a` - do track lib.a, even though you ignore *.a files
- use 2 asterisks (**) to match nested directories
    - `doc/**/*.pdf` - ignore all pdf files in the doc/ directory and all its subdirectories
- `*` matches anything, `?` matches any single character (both match all except `/`), `[]` are used to specify ranges (`[a-z]`)

git diff

Shows changes that were introduced between two references; without arguments shows uncommitted (unstaged) changes. Use `git diff HEAD` to show both unstaged and staged changes.

```
$ git diff <ref1>..<ref2>
$ git diff HEAD~2..HEAD
# Changes in main since feature was created from it
$ git diff feature..main
# Changes in current working directory
$ git diff
```

- `--staged` or `--cached` - shows changes in staging area (difference between last commit and staged changes)
- `-w` - ignore white-space changes

git difftool - allows for graphical view in configured editor, run **git difftool --tool-help** to get more details

git commit - commits staged changes, with no arguments launches the chosen (--global core.editor) editor - lines starting with # will be ignored

- `-v` - in addition to default call adds the output of **diff** command
- `-m <message>` - alternative, commit message
- `-a` - skip the staging area - automatically add all files that are already tracked (does not include new, untracked files)

`git rm <file>` - remove file from working directory and stage the removal

- `--cached <file>` - remove file from staging area, but not from working directory (for example forgot to add to gitignore)
    - `log/\*.log` - can pass files, directories and file-glob patterns, backslash before start in necessary, as git applies its own filename expansion in addition to shell's expansion

`git mv <file_from> <file_to>` - convenience command, applies **mv**, removal of old name and staging of new new (will appear as **renamed** in **git status**)

Viewing the commit history

git log - with no arguments lists all commits in chronological order (from most recent to later ones); pretty option with one line or format specifiers work particularly useful with graph option

- `--patch` or `-p` - show difference introduced in each commit
- `--stat` - show changes per file - additions, deletions, etc
- `--shortstat` - same as above, but obviously short
- `--pretty=[oneline | short | full | fuller | format]` - outline info in different format than the default; all except *format* are built-ins, while format lets you create your own (see below)
  - `git log --pretty=format:"%h - %an, %ar : %s"` - example
  - %H - commit hash
  - %h - abbreviated form
  - %T - tree hash
  - %t - abbreviated form
  - %P - parent hashes
  - %p - abbreviated form
  - %an - author name
  - %ae - ... email
  - %ad - ... date
  - %ar - ... date, relative
  - %cn - committer name
  - %ce - ... email
  - %cd - ... date
  - %cr - ... date, relative
  - %s - subject
- `--graph` - nice little ASCII graph
- `--name-only` - show the names of file modified after each commit info
- `--name-status` - same as above with added/modified/deleted info as well
- `--abbrev-commit` - abbreviated form for checksum
- `--relative-date` - display date in relative format
- `--oneline` - shorthand for `--pretty=oneline` --abbrev-commit together

`git show <identifier>` - show `git log -p` output on specified commit through identifier

`git shortlog` - group commits by user

Limiting log output

- `-<n>` - where n is an integer show last `n` commits
- `--since` or `--until` - specify the date; can accept different formats:
  - "2.weeks"
  - "2009-01-15"
  - "2 years 1 day 3 minutes ago"
- `--author` - show commits from particular author
- `--grep` - search for keywords in the commit messages
- `-S <string>` - Git's "pickaxe" option, takes a string and show only those commits that changed number of occurrences
- `-- /path/to/file/or/directory` - output commits that introduced a change to those files
- `--no-merges` - not merge commits

Undoing things

git commit

- `--amend` - redo last commit (in case of forgetting to add some files, changing commit message, making additional changes), in other words take the staging area and uses it for the commit (actually takes last commit away, creates new one and pushes it over); f.e.:

```
$ git commit -m "initial commit"
$ git add <forgotten_file>
$ git commit --amend
```

`git reset HEAD <path/to/file/or/directory>` - unstage the specified file or directory

`git checkout -- <path/to/file/or/directory>` - unmodify a file, revert it to last commit

`git revert <reference>` - creates a revert commit to the specified reference (both staged and committed); docs

Working with remotes

Remote repositories are version of the project that is hosted elsewhere (can be remote server, network, internet, even local machine).

`origin` is the default name Git gives to the server that you clone from. `upstream` is a convention for referencing forked repositories (usually in open source development).

git remote lists configured remote repositories (on **git clone** Git automatically adds remote, named *origin*, pointing to original repo)

- `-v` - show URLs as well (URL can also be a local absolute or relative path)
- `add <remote_name> <remote_path>` - add a remote
- `show <remote_name>` - see more info about particular remote
- `rename <old_name> <new_name>` - rename remote's short name (changes references as well, f.e. `<old_name>/master` to `<new_name>/master`)
- `remove <remote_name>` or **rm** `<remote_name>` - remove remote, all remote-tracking branches and configs are deleted as well

Git fetch pulls down all data from the remote project that you don't have locally. After it you have all references to all branches locally and can merge or inspect at any time. Does not merge data automatically. If current branch is set up to track a remote branch, then **git pull** automatically does **git fetch** and then tries to merge the changes.

`git fetch <remote>` - get the data from the remote server

- `<remote_branch>:<local_branch>` - fetch remote branch into local branch (without checking out to that branch)

git push command creates local branches in the destination repository.

```
# push changes to remote master branch
$ git push <remote_name> <branch_name>

# example above is a shortcut to the next command
# use it if local branch has a different name than the remote one
$ git push push <remote_name> <local_branch_name>:<remote_branch_name>
```

Tagging

Tags are used to explicitly mark certain points in history.

To create a lightweight tag don't pass any additional options - `git tag v1.4-lw`

Types of tags:

- lightweight - just like branch that doesn't change, in other words pointed to specific commit
- annotated - stored as full object in Git database (checksummed as well), since it contains additional information - tagger name, email, and date. (can also be signed and verified by GNU Privacy Guard, GPG)

`git tag <pattern>` - list existing tags, optional pattern (enclosed in double quotes) can be passed as well

- `--list` or `-l` - act like default for entire list of tags; if pattern is supplied flag is mandatory for desired behavior
- `-a <tag_name> <commit_checksum>` - create annotated tag; optional `<commit_checksum>` parameter to tag past commits (short or full version of checksum)
- `-m <message>` - specify tagging message
- `-d <tag_name>` - remove a local tag

`git show <tag_name>` - show the tag data alongside the commit info

`git push <remote_name>`

- `<tag_name>` - push tag to remote server
- `--tags` - push all tags at once (pushes both types of tags)
- `--delete <tag_name>` - remove tag from remote server
- `:ref/tags/<tag_name>` - way to interpret - read the value before colon as null, thus pushing null == deleting tag

`git checkout <tag_name>` - checkout to the commit by passing a tag that points to it

Git Aliases

Simple as any other aliasing - just replace "long" command with you alias!

`git config --global alias.<alias> <command>` - if command consists of multiple arguments, its best to enclose them in single quotes

Good examples:

- `git config --global alias.co checkout`
- `git config --global alias.br branch`
- `git config --global alias.ci commit`
- `git config --global alias.st status`
- `git config --global alias.last 'log -1 HEAD'`
- `git config --global alias.lo 'log --oneline'`

Git Branching

Remotes are for people, whereas branches are for topics.

Branches in a nutshell

- Blob - represents the metadata for individual file
- Tree - lists the contents of the directory and specifies which file names are stored as which blobs
- Commit - contains the pointer to the tree and all commit metadata; if its not a root commit, it also contains pointers to parent(s)

Thus when git stages files, it computes checksums for each file and stores that version of the file in the Git repository (blob). When git performs commit command, it checksums each subdirectory of the project and stores them as a tree object. Commit objects contains commit metadata and pointers to the trees

git branch - list all local branches

`git branch <name>` - create a new branch; actually creates a new pointer to the commit you are currently on

To keep track where you are currently on Git uses special pointer `HEAD` - it shows where you are on in the local repository.

`git checkout <tag | commit | branch>` - switch to the provided commit, branch (moves `HEAD`)

`git checkout -` - switch to the previous commit, branch

`git checkout -b <name>` - create and switch to a branch at the same time

Summary: creating a branch in Git is just writing small metadata file, which acts like a pointer to commit.

Basic branching and merging

Examples below contain following setup: one branch checkouts from master to do additional work. Some work was committed to this additional branch and now source branch is what you want to merge(additional branch) and target branch (master) is where you want to merge it

To perform merge checkout out to target branch and type in `git merge <source_branch` .

Performing merge can lead to 2 cases :

- source branch can be reach by following target branch's commit history, that is source branch was directly ahead of target branch - this case lead to fast-forward "merge", git just moves the pointer ahead
- source branch have diverged from target branch and there are no conflicts, so git creates new snapshot (merge commit), which results from two tips of branches and their common ancestor
- source branch have diverged from target branch and there are some conflicts, so git pauses and until you resolve conflicts (then you can add-commit the work); list unmerged files by running `git status`

git mergetool - open graphical tool to resolve conflicts, append tool name to use other than default

- `--tool-help` - get help, f.e. tool is not configured

`git merge --no-ff <branch>` - force a merge commit where Git would normally do a fast-forward

`git branch -d <name>` - delete branch locally, with `-D` flag to force delete (in case of unmerged changes)

`git push --delete origin <branch>` - delete remote branch

`git merge -` - merge previous branch into current one

Branch management

git branch

- `-v` - list branches and their last commits as well

- `--merged` - list branches that are already merged into the branch you are currently on (good candidates to be deleted, since the current branch contains their work)
- `--no-merged` - list branches that contain work that have not been merged into branch you are currently on (trying to delete this branch will result in warning)
- `-D <branch>` - force deleting
- `-r` - list remote branches (always specified in the `<remote-name>/<branch-name>` form, not to be confused with local branches)
- `-m <current_name> <new_name` - rename branch; skip first argument to rename current branch

`--merged` and `--no-merged` are relative to current branch if no argument is passed

Remote branches

`git ls-remote <remote>` or `git remote show <remote>` - get full list of remote references or remote branches with more info respectively.

Remote branch's name takes the form `<remote>/<branch>`, for example `origin/master`.

You have cloned a repository (thus your master and origin/master refer to the same commit), but later on someone has push to server and you have committed further locally. Run `git fetch <remote>` to get the latest version of origin to you local repository.

`git checkout -b <branch> <remote>/<branch>` - merge remote branch into current working branch, thus local branch track the remote one

`git checkout --track <remote>/<branch>` - create and check out to "tracking branch" - local branch that has direct relationship with remote branch (git know where to push, pull, etc)

`git checkout <branch>` - shortcut for the above, if branch does not exist locally (and only exists on one remote) Git creates "tracking-branch" automatically

git branch

- `--set-upstream` or `-u` - set the upstream branch to track for local branch you are on
- `-vv` - also output tracking branches (if local branch is tracking remote one), also if current branch is ahead or behind; does the comparison based on local references, thus do `git fetch --all` to get up date information

git pull - is essentially `git fetch` followed by `git merge`

`git push <remote> --delete <branch>` - delete remote branch

Rebasing

Setup: source branch checks out from target and does some work, while other work was done on target as well. Then source branch tries to perform rebase.

To perform rebase - checkout to source branch and type `git rebase <target_branch>` .

`git rebase <target_branch>` - rebase current branch into target branch

- `--onto <parent_a> <parent_b> <source_branch>` - in the situation when `parent_b` diverged from `parent_a` and `source_branch` further diverged from `parent_b` use the following to rebase `source_branch` to the `parent_a`
- `--continue` - restart rebase after resolving conflicts
- `--abort` - abort the rebase operation

Operation: going to common ancestor, getting the diff introduced by each commit of the branch you are on, saving the diffs to temporary files, resetting the current branch to the same commit as the target branch, and applying each change.

Thus rebasing just replays the changes that took place in one branch in another branch, while classic merge takes endpoint and merges them together. After rebase is done the target branch can be simply fast-forwarded by `git merge <source_branch>`

Do not rebase commits that exist outside your repository and that people may have based work on - golden rule; that is when you rebase you introduce new commits, while abandoning other, while other commits may have been used by others when they merged their work. Try to run `git pull --rebase` or `git fetch ; git rebase <remote>/<branch>` to let git try to rebase your work over rebase other people have performed earlier (and you work was done on top of the old commits that were abandoned because of that rebase). If new rebased commit is nearly identical to the one that was on separate branch, git might be able to successfully apply rebase of your branch as well.

Good rule of thumb is to rebase work you haven't pushed anywhere.

Git tools

Revision selection

Git provides number of ways to refer to single commit, set of commits, or range of commits.

- Single revision

    - 40-character full SHA-1
    - short SHA-1 - can pass short version as long as its not ambiguous (no other commit has it); pass `--abbrev-commit` to `git log` to output shorter versions (but still unique), defaults to 7, but gets longer to keep them unique
    - branch reference - the commit at the tip of the branch; `rev-parse` is a low level operation that name resolves too - `git rev-parse <branch_name>` will output its SHA-1
- multiple revisions
    - `-n <number>` - limit output; for example, `git log HEAD~4..HEAD` is same as `git log -n 4`

Git keeps logs of where your HEAD and branch references have been in the last few months (chronological listing of history, performed actions); this information is strictly local and represents only local changes/info; use the command below (in case of hard reset, this command can be used to retrieve dangling commit):

git reflog

`git show <reference>` - can refer to reflog data as well, examples below (substitute <reference>:

- `HEAD@{5}` - 5th entry from `git reflog`
- `master@{yesterday}` - using syntax to access specific dates

`git log -g` - see reflog info in `git log` output

Ancestry reference uses `^` (caret) symbol. If added at the end of a reference, it is resolved to the parent of that commit - `git show HEAD^` resolved to the commit before current one. Can also specify number after caret to specify which parent (useful for merge commits) - `HEAD^2` - first parent is the branch you were on when performing the merge, while second is the one that is merged.

Another ancestry reference is `~` (tilde). Always goes to the first parent, thus `HEAD~2` means *first parent of the first parent*. Two syntaxes can be also combined `HEAD~3^2`

Double-dot syntax - shows range of commits that are reachable from one commit, but aren't from another, f.e.

```
A->B->E->F->master
   |->C->D->experiment
```

`git log master..experiment` - will output D and C, while vice versa will return F and E

`git origin/master..HEAD` - shows what commits are going to be pushed to the server (can omit HEAD, git will substitute it)

`^` or `--not` before any reference means do not show reachable commits from that reference; can specify more than two reference (unlike 2 dot)

`git log refA..refB` is the same as `git log ^refA refB` and same as `git log refB --not refA`

Triple-dot shows commits that are reachable by either of the two references, but not by both; `--left-rght` options helps to determined which commits are reachable by which of the two references:

`` `git log master...experiment` returns F E D C ``

`git log <since>..<until>` - displays the commits reachable from `<until>` but not from `<since>`.

Interactive staging

`git add [-i | --interactive]` - interactive shell mode

Patch option lets you stage separate hunks of file. Patch option `[-p | --patch]` also works with `add`, `reset`, `checkout` and `stash save` commands

Stashing and cleaning

git stash or **git stash push** - stash changed files (dirty state) onto separate stack.

- `list` - show the list of stashed files on the stack; following commands, if not specified the index, assume that operation is to be done on the top most item
- `show` - show stashed files and high-level summary on changes
- `apply [--index] stash@{n}` - apply the file in the current state (branch and directory); does not drop the file from stash stack; `--index` flag tell git to apply it to the staging area; can pass `n` to specify with file to apply
- `drop stash@{n}` - drop item from the stack
- `pop stash@{n}` - apply the stash and drop from stack
- `--include-untracked` or `-u` - include untracked files
- `--all` or `-a` - include all files (also ignored ones)
- `--patch` - interactive stashing
- `branch <new branchname>` - create a new branch from a stashed work; creates new branch -> checks out the commit you were on -> applies stashed items -> drops the stash if applied

successfully

git clean - removes untracked files irreversibly; safer option is to run `git stash --all`

- `[-f]` - force, "really do this" - required flag by Git, unless configuration variable `clean.requiredForce` is not set to false; second `-f` might be needed if you try to removed submodules
- `[-d]` - remove directories that become empty as a result of clean command
- `[--dry-run | -n]` - show what would have been removed; that is put `-n` instead of `-f` to inspect first
- `[-x]` - removed ignored files as well
- `[-i]` - interactive mode

Searching

git grep

Search within working directory or anywhere in the git history

- `--line-number` or `-n` - prints out line numbers where matches were found as well
- `--count` or `-c` - show summary with filenames containing the match and number of matches in each
- `--break` - break each file (insert new line after each file) output
- `--heading` - instead of starting each entry with the filename containing it, output filename on the separate line, then lines with the match
- `add more`

git log

- `-S <string>` - output commits that changes the number of occurrences of provided string
- `-G`
- `-L :<function>:<file_name>` - line log search, git will try to figure out the bounds of the provided function and output the changes made to it throughout whole history; if git cant figure out the bounds, pass the regular expression, for the provided example it would be `git log -L '/unsigned long function/',/^}/:file_name` ; can also pass range of lines or single line

Rewriting history

All the following is preformed locally. Once the work is pushed to the server, it is completely different story.

`git commit --ammend` - modify last commit message; opens configured editor and loads it once editor is closed; can also be used to change the contents - simply make changes, stage them and then perform the command (changes the hash of commit)

- `--no-edit` - avoid editor session (for example adding files or changes, and there is no need to change the message)

To change multiple commits use `git rebase` command with `-i`, for interactive, option; do not specify the commits already on the server, because they will be rewritten. The following example changes last 3 commits (could also specify `HEAD~2^`); reference is the new base.

```
git rebase -i HEAD~3
```

The command returns the list of commits (but in reverse order than `git log`):

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

```
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Git will perform the specified commands from top to bottom(when you close the editor)

- change commit message - change `pick` to `edit`, git will stop there, perform usual `git commit --amend`, continue with `git rebase --continue`
- reorder commits - simply reorder / delete commit entries
- squash commits - change `pick` to `squash` on commit that you want to squash to previous one, in the following example last two commits are squashed into first; git then puts you into editor to merge commit messages (if want to push updated commits to remote finish with `git push --force-with-lease origin` command):

```
pick <hash> commit message one
squash <hash> commit message two
squash <hash> commit message three
```

- split commit - change `pick` to `edit`; git will perform parent commits and the one indicated as `edit` also stop right after, then you can `git reset HEAD^` and redo commit(s) finishing with `git rebase --continue` when done

git filter-branch - rewrite large portions of history, though python script `git-filter-repo` is recommended and can be found here + `--tree-filter <shell_command> HEAD` - runs specified shell command on each checkout, and recommits the result + `--all` - run in all branches + `--subdirectory-filter <directory> HEAD` - makes the specified directory a root directory for every commit; git also automatically removes all commits that did not affect the subdirectory [NEEDS_CHECK] + `--commit-filter` - change the email in all commits, example below (changes hashes of all commits, not just where the change applied) `shell git filter-branch --commit-filter ' if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ]; then GIT_AUTHOR_NAME="Scott Chacon"; GIT_AUTHOR_EMAIL="schacon@example.com"; git commit-tree "$@"; else git commit-tree "$@"; fi' HEAD`

Cherry-pick

Commonly used to apply a bug fix for multiple versions/revisions or to capture some commits from inactive branch. Creates a copy of a referenced commit and applies it to head.

```
$ git checkout <reference> # where to apply the commit
$ git cherry-pick <commit_hash>
```

Reset demystified

3 trees (workspaces) that git system manages and manipulates:

1. HEAD - pointer to the last commit on the branch(current branch reference)
2. Index - proposed next commit(staging area)
3. Working directory (working tree) - while previous two trees store contents in `.git`, this one unpacks them so you can easily work with files; refer to this tree as sandbox, where you can try and do all the changes

Workflow goes as follows: |<--------(checkout the project)--|------------------HEAD |----------------------------Index---(commit)-->| working directory--(stage files)->|-----------------|

You make changes in the current working directory, then run `git add`, which moves the modifications to staging area (Index), then `git commit` creates a snapshot, which finally makes all trees to agree.

git reset

- `--soft <reference> <file>` - moves the *HEAD* (also branch it is pointing to) to specified commit; `git reset --soft HEAD~` (go to parent of *HEAD*) moves the branch to the previous commit (undo), *Index* and *Working directory* does not change
- `--mixed <reference> <file>` - moves the *HEAD* and updates Index as well(unstage changes) - opposite of `git add`; the default `git reset` assumes `git --mixed HEAD~` to all files - effectively unstages all files
- `--hard <reference> <file>` - moves the *HEAD* and updates both *Index* and *Working directory*; commit you moved from can be recovered from Git DB, but otherwise all data is overwritten; one way to retrieve overwritten commit is to find the commit ID using **git reflog**, checkout to that commit, create a branch and finally merge it with the original branch

git reset also accepts `--patch` option to unstage on a hunk-by-hunk basis.

git reset file-path behaviour matches committed snapshot (reference) and *Index* (staging area); example below unstages specified file (does not alter *Working direcroty* or move the current branch):
`shell $ git reset HEAD <file>`

git checkout file-path behaviour matches commited snapshot and *Working directory*; example below reverts an individual file to match the specified commit without switching branches ```shell $ git checkout

```
# Drop current changes in working directory
$ git checkout HEAD <file>
```

Patch workflow

Overall workflow is as follows: developer creates a topic branch, makes changes, sends patches to repository maintainer, who incorporates patches to the local topic branch, then merges it with local stable branch and then publishes updated branch to the remote. Other developers, including the author of the change can then fetch updates from remote.

git format-patch

Creates a file (`gitsendmail` format) to re-create a commit.

```
# Creates patches for all commits not present on the branch_name
$ git format-patch <branch_name>
```

git send-mail

Takes the patches specified on command line and sends them out. Documentation

git am

Applies patch file to the current branch, effectively recreating the commit.

```
$ git am < <name>.patch
```

Plumbing

Tree object is Git's representation of the "snapshots". They record state of a directory at any given point notion of time or author. Each tree object is then wrapped into *commit object* with a parent reference, which is just another commit. *Tree* contains *blobs* and may also contains other *trees*, which are folders. *Blob object* contains plain data file. Name of the file is stored in a *tree* object that contains the *blob*, not in the *blob* itself. *Tag object* is simply a reference to a *commit object*.

All Git objects have associated SHA checksum, which gives each object unique identification and ensures it's contents aren't corrupted. Since same file would produce same checksum, Git shares *blobs* across trees, instead of duplicating them (when contents of the file has not changed). As soon as anything has changed in a file, new *blob* is created.

git cat-file

Show information on Git objects - one of `commit`, `tree`, `blob`, or `tag`.

- `-t` - show object type, f.e. `git cat-file -t master`
- `-p` - pretty print the object, e.g `git cat-file -p <hash>`

```
$ git cat-file commit HEAD
```

git ls-tree

Display a pretty version of the specified tree object (use ID from **git cat-file** command).

```
$ git ls-tree <ID>
```

All Git objects are stored in `.git/objects` and split across directories, based on first 2 letters of a SHA checksum (f.e. `7acjsu3192` would reside in directory `7a`). The remaining of a checksum is used as a filename.

As repository grows Git may automatically transfers object files into a more compact form, *pack file*. Garbage collection is done by compressing individual object files into faster and *pack file* and by removing dangling commits (deleted, unmerged branches, etc).

git gc

Perform garbage collection on the object database.

Low level commands for typical workflow

1. **git index-update**

Stage the specified file.

- `--add` - used for adding new files

```
$ git index-update index.html
$ git index-update --add new_item.html
```

2. **git write-tree**

   Generate a *tree object* from an *Index* and store it in object database. Returns the ID of the new *tree object*.

3. `git log --oneline -n 1` - find out commit ID of the parent

4. **git commit-tree**

   Create a *commit object* from the given *tree object* and parent commit. It also waits for input that will be used as commit message (use `Ctrl-D` to send "End-of-file" character to end the input). Returns the ID of the new *commit object*.

   ```
   $ git commit-tree <tree_id> -p <parent_commit_id>
   ```

5. update branch reference - got to `.git/refs/heads/branch_name` or to `.git/packed-refs`, if previous file is not present (means `git gc` was run or Git did it automatically)

Tips and tricks

archive repository

Saves current snapshot omitting `.git` directory. Format can be `zip` or `tar`.

```
$ git archive <branch_name> --format=zip --output=<file_name>.zip
```

bundle repository

Turns repository into a single file with the versioning information of the entire project. Similar to pushing a branch to remote. Resulting file can even be used in `git clone` command.

```
$ git bundle create <file_name>.bundle <branch_name>
$ git clone <file_name>.bundle -b <branch_name>
```

hooks

Git hooks are scripts that can run when particular event occurs, such as new push. Samples are located under git `hooks` directory.

Documentation

- **bisect** command which is used to find where "the feature" was broken first
- can pass the script to check it
- **git stash push path/to/file -** stash individual file
- **git stash -p -** ask before every stash
- resolving merge conflicts
- rewriting history

Useful commands

View introduced changes

```
$ git log start..end --stat
$ git log HEAD~4..HEAD --stat
```

Make history changes (squashing, changing commit messages, etc) in the development branch (forked from main); substitute *master* to target branch name

```
$ git rebase -i master
```

In case of working on outdated branch and trying to push changes to remote, first, fetch changes, then rebase on remote:

```
$ git fetch origin
$ git rebase origin/master
$ git push origin master
```

References

- Git book
- Git tutorial

- Ry's Git Tutorial
- Checkout the Previous Branch in Git
- How to write a Git commit message
- List of supported languages for markdown code blocks

Pages 52

Find a page...

Ansible
ArgoCD
AWS
AWS_Compute
AWS_Data
AWS_Databases
AWS_Developer_Tools
AWS_Integration
AWS_Monitoring
AWS_Networking
AWS_Security
AWS_Storage
Azure
Azure_DevOps
Azure_DevTest_Labs

Show 37 more pages...

Clone this wiki locally

```
https://github.com/kamialie/knowledge_corner.wiki.git
```