

1 Work

Terminology & state machine:

State	Index	W-Dir	Next
Untracked	?	?	Add
Unmodified	-	-	Edit
Modified	-	M	Stage
Staged	[A M]	[M]?	Commit

You can `rm` from “untracked” or `git rm` from any other state.

git reset

Reset moves HEAD’s branch & HEAD, versus checkout, which only moves HEAD. Below, `<ref> ∈ {SHA1, branch, file path}`:
`git reset <ref>` # move HEAD bch & idx to ref
`git reset --mixed <ref>` # equiv to above
`git reset --soft <ref>` # Move HEAD brch → idx
`git reset --hard` # revert WD, idx & brch
Only the `--hard` option is dangerous to WD.
Revert specific file in working directory:
`git reset <file>` # file in WD Δs to last HEAD
`git reset <SHA> <file>` # revert to specific SHA
`git checkout -- <file>` # “unsafely” revert file

git add

`git add <filename>` # add to index
`git add .` # stage all changed
`git add --all` # stage all
`git add -i` # interactive staging

git commit

`git commit -m 'message'` # commit all staged
`git commit -am 'message'` # commit all tracked
`git commit -a --amend` # redo last commit
`git commit --amend` # tweak last commit

git tag

`git tag` # show tags
`git tag -l` # same as above
`git tag -l <tag_ref>` # show all matches
`git tag "<str>"` # create “lightweight” tag
`git tag -a <str> -m "<str>"` # “annotated” tag
`git tag -a <tag> <hash>` # after-the-fact tag
using tags:
`git show <tag>` # display commit @ <tag>
`git checkout <tag>` # checkout tagged commit
remote tags:
`git push origin <tag>` # explicit push required
`git push <remote> --tags` # push all tags
deleting tags:
`git tag -d <tag>` # delete locally
`git push <remote> --delete <tag>` # remotely
sign tags (must disseminate pub key):
`git tag -s v1.5` # sign

git rm

`git rm <file>` # stage <file>’s removal
`git rm -f <file>` # remove modified / staged
`git rm --cached <file>` # untrack
Can use globs instead of filenames:
`git rm *` # anything ending in ~

git mv

`git mv <old> <new>` # rename to <new>
This is shorthand for:
\$ `git rm _; mv _ _; git add _`
on a tracked file.

git stash

`git stash push` #
`git stash list` #
`git stash apply` #
`git stash drop` # remove from stack
`git stash pop` # apply & drop last

git cherry-pick

Like a rebase for a single commit. Can cherry-pick multiple commits into one prior to integration, or cherry-pick single commit onto master instead of rebasing.
`git cherry-pick <SHA1>` #

2 Observe

git status

`git status` # show w-dir vs index
`git status -s` # short-hand view

git diff

`git diff` # wd vs repo
`git diff --staged` # idx vs repo
`git diff master origin/master` # diff vs remote
`git difftool` # use a different diff

git log

`git log` #
`git log -p` # shows Δ’s (“patches”)
`git log -3` # last 3
`git log --stat` # include summary info
`git log --graph` # graphical
`git log --decorate --all` # show branching
`git log --pretty=oneline` # beautiful
`git log --pretty=format:"<fmt>"` # format

• %H commit hash	• %ae auth. email
• %h abbrev. hash	• %ad auth. date
• %T Tree hash	• %ar relative date
• %t abbrev. tree	• %cn committer name
• %P parent hashes	• %ce commit. email
• %p abbrev. prnt	• %cd commit date
• %an author name	• %cr relative date
	• %s subject

Subselect & bracketed logs to show:
`git log --since=<date>` # bracket low
`git log --since=2.weeks` # for example
`git log --until=<date>` # bracket high
`git log --author=<auth>` # by author
`git log --grep <patrn>` # matching pattern
`git log -S <str>` # “pickaxe” option
`git log brnA..brnB` # only brnB, → brnA
`git log --not brnA brnB` # same as above

git describe

Create version # from latest tag:
`git describe master` # yields v1.6-<SHA1>

git shortlog

`git shortlog master --not v1.5` # since 1.5

git grep

- `-p ...` enclosing block
- `-e <str> ...` exists string
- `--and ...` complex matches

git blame

`git blame <file>` # show last Δs, authors
`git blame -L 51,59 <file>` # restrict to line #s
`git blame -C <file>` # find Δ-origins, other files

git reflog

`git reflog` # local history of HEAD, branches

git show

`git show master@{5}` # using reflog labels
`git show master@{yesterday}` # special labels

git bisect

Iteratively search for breaking Δs by repeatedly resetting old commits. Is a commit good or bad?
`git bisect start HEAD v1.0` # from 1.0 to HEAD

3 Branch

Types of commits:

- initial commit ... no parent
- normal commit ... one parent
- merge commit ... multiple parents

Types of branches:

- “Long-running” branches
- “Topic” branches
- “Tracking” branches

Types of history:

- Original, unedited (3-way merges)
- Curated, linearized (rebase, cherry-pick)

git branch

`git branch -v` # show branches
`git branch -vv` # show tracking branches
`git branch --merged` # end-of-road branches
`git branch --no-merged` # show WIP branches
`git branch <bnam>` # new branch @ HEAD
`git branch -d <branch>` # delete
Set upstream branch:
`git branch -u <remote>/<branch>`

git checkout

A checkout moves HEAD to a named branch or SHA commit. You cannot checkout a branch that clashes with changes in your working directory.
`git checkout <branch>` # move HEAD
`git checkout -b <bnam>` # create & checkout
`git switch -c <bnam>` # ibid, new in v2.23
Create remote tracking branch:
`git checkout -b <name> <remote>/<branch>`
`git checkout --track <remote>/<branch>`

git merge

`git merge <branch> [<into>]` # * assumed
A merge is three-way, generally, but a two-way “fast-forward” when <branch> is a direct ancestor of <into>.
`git merge @{u}` # shorthand for upstream
`git mergetool` # visual conflict-resolution
`git merge --squash brnA` # single parent

git rebase

Rebasing replays changes from one line of work onto another in the order they were introduced. Only rebase if it’s the project’s convention to do so and if you want to clean up your clutter. **Never rebase anything you’ve pushed** (although it’s technically still okay if no one else will base their work on those commits).

Rebase <topic> onto <base>:
`git rebase <base> <topic>` # general form
Now, <base> is fast-forwardable:
`git checkout <base>`; `git merge <topic>`
`git rebase master` # rebase * onto master
Rebase c onto a, holding off on b:
`git rebase --onto <a> <c>`
Split, squash, edit, reorder commits:
`git rebase -i ...` # interactive

git merge-base

Finds common ancestor:
`git merge-base brn master` # find
`git diff $(git merge-base brn master)` # use
`git diff master...brn` # shorthand for above

4 Share & Synchronize

Major sync work-flows:

- Centralized – any can push to a shared bare
- Managed – only manager can push
- Hierarchical – multiple levels of pushing

git fetch

`git fetch <remote>` # download latest
`git fetch <remote> <refspec>` # specific rqst

git push

Only works if:

- you have write access to <to>
- no one has pushed since your last fetch

If 2nd condition isn’t true, it is an attempted push of “non-fast-forward” changes.

`git push <to> <from>` # generally
`git push origin master` # specifically
`git push -u origin my_brn` # set remote & push
`git push <refspec>` # eg, frombrn:tobrn
Push message includes a statement like:
1edee6b..fbff5bc master -> master
ie, <oldref>..<newref> <frombr> -> <tobrn>
Pushes on rebased branches require forcing:
`git push -f myfork brnA` # force

git pull

`git pull` # `git fetch` + `git merge`
`git pull <url>` # one-time: don’t set remote

git remote

`git remove -v` # show all remotes
`git remote add [<nickname>?] <url>` # add
`git remote show <remote>` # inquire
`git remote rename <old> <new>` # new alias
`git remote rm` # no longer track remote
`git remote set-head #`
`git remote prune #`
`git remote set-branches #`

git request-pull
Absent GitHub forks & pull-requests, use this to facilitate the integration process. (A fork is just a clone to or push to an accessible repo.)
git request-pull origin/master myfork # text

GitHub
Generally, pull requests don't close right away. The idea is to open, collaborate, & finally merge. Use markdown to reference IDs & prettify:

- #<id> reference branch, commit, or PR
- username#<refid> more specifically
- username/rep#<id> or externally
- <SHA1> unchanging, concretely
- <emo>: enter an emoji
- [X] item A [] item B or checklist
- java""if(true) a=b;"" code snippet
- > when that aprile block quote

"Trackback" event is then created in referenced conversation.

5 Setup

git config
Manipulate configuration settings.
git config --system <cfg> # in /etc/gitconfig
git config --global <cfg> # in ~/.gitconfig
git config --local <cfg> # in ~/.git/config
where <cfg> is <key> <value>, for example:
git config --global core.editor vim
git config <key> # query value
git config --list # show all cfgs
git config --list --show-origin # cfg origin
Creating aliases:
git config --global alias.<als> <cmd>
git config alias.l log --oneline # examples
git config --global alias.unstage 'reset HEAD --'
Simplify long commands:
git config --local remote.pushDefault origin
Run man git-config for more config options.

git help
git help <cmd> # general inquiry
git help config # eg: help on "config"
git <cmd> -h # abridged help

git init
Start new git repository locally.
\$ mkdir /Projects/moustrap # create
\$ cd /Projects/moustrap # relocate
\$ vim innovation.rs # innovate
git init # track
Create a central hub repo, ie one you can push to that lacks a working dir:
\$ mkdir /Projects/central # create
\$ cd /Projects/central # relocate
git init --bare # no working dir.

git clone
Grab & track an existing, remote git repository.
git clone <url> # store locally in \
git clone <url> <name> # set local <name>
git clone <url> -o <name> # set remote name
git clone --bare <url> # no working dir
git clone ssh://[user@]server/project.git # SSH

.gitignore
Pattern matching rules:

- comment out line with #
- standard globs (*, ?) work recursively
- avoid recursion by prefacing with \
- match directory recursion by **
- specify directory by suffixing with \
- negate a pattern with !
- case matching with [abc] or [a-c]

For example:
(file) *.a # ignore all *.a
(file) !lib.a # do track lib.a
(file) /TODO # ignore top-level /TODO
(file) build/ # ignore anything under build/
(file) doc/*.txt # ignore doc txt's
(file) doc/**/*.pdf # ignore all pdf's

Key Management
ssh-keygen -o # create pub/prv key pair
cat ~/.ssh/id_rsa.pub # view pub key

Server Administration
There are several ways to serve git (and cons):

1. The local filesystem (no redundancy)
2. An SSH server (cumbbersome user admin)
3. GIT protocol (no user admin possible)
4. "Smart" HTTP (complex web server)
5. Self-hosting, eg GitLab (time to set up)
6. Commercial hosting, eg Github (money)

Set up remote access SSH server:
mkdir .ssh && chmod 700 .ssh && /
touch .ssh/authorized_keys && /
chmod 600 /ssh/authorized_keys
Allow jack access to ssh user:
cat /tmp/id_rsa.jack.pub » /
~/.ssh/authorized_keys
Set up Git-protocol server:
git daemon --reuseaddr <path>
cd prj_path/ && touch git-daemon-export-ok

6 Patching
Partial changes
Use --patch with: git [add|reset|checkout|stash]:
git add --patch # interactive patch staging
git format-patch
Generate emailable patches, one per commit
past indicated branch (here origin/master):
git format-patch -M origin/master

git [send]
cat *.patch | git imap-send
git send-email *.patch

git [apply|am]
Use former if user submitted diff format "patch"; latter if user submitted format-patch patches. Latter is better: already contains commit metadata.
git apply /tmp/pname.patch
git am pname.patch
git am -3 pname.patch # attempt 3-way merge

7 Advanced | Miscellaneous

git submodule
Add (remove here) a submodule to project:
git submodule add <url> # add submodule
Getting a project with submodules:
git clone --recurse-submodule <url> # init all
... or, if you forget to --recurse-submodule,
you can initialize each SM manually:
git submodule init # initialize single SM
git submodule update # fetch & checkout
where "update" is the usual wrapper:
cd <SM>; git fetch; git merge # ≡ update
git diff --submodule # show SM Δs
Pull (or push) a repo with Δed submodules:
git pull --recurse-submodules # or use update
Merge or rebase (--rebase) local & remote Δs:
git submodule update --remote --merge
Do same Δs across all SMs:
git submodule foreach '<cmd>' # cmd is bash
Handy configurations:

- diff.submodule
- submodule.<SM>.branch
- status.submodulesummary
- submodule.recurse***
- push.recurseSubmodules

git rerere
Automatically resolve conflicts similar to ones you've fixed in previous merges. Rerere essentially caches pre- and post-merge diffs.
git config --global rerere.enabled true
git rerere # direct invocation

Porcelain

archive	create tarball, etc of .git filesystem
bundle	package & share w/o network
fsck	
gc	manually invoke git garb. collector
replace	Δ hashes for similar commits

Plumbing
"Object" ∈ {blob (file), tree, commit, tag}
"Reference" ∈ {head (branch), tag, remote}

cat-file	inspect git object (try: -p -t)
commit-tree	requires 1 tree (content) & n objects
filter-branch	Δ all history!!
hash-object	create [& write] objects into git db
ls-files	
ls-remote	
ls-tree	
read-tree	load tree into index
rev-list	
rev-parse	
show-branch	
show-graph	
symbolic-ref	set HEAD
update-index	Δ & add to index from file or gitdb
update-ref	set branch, HEAD, tag pointers
verify-pack	show how git packs Δs into "packfiles"
write-tree	create tree from index

Refspec
Syntax is: [+<src>:<dest>] Often, can use <refspec> instead of a <ref> like a branch, tag, etc.

8 Ancestry

~# is #th ancestor;
^# is either:
1 - merge into or
2 - merge from

Complex exclusions:
git log ^<ref> # synonym for --not
git log brA brB ^brC # from A or B not C

Paths

A	A^0	
B	A^1	A~1
C	A^2	
D	A^1^1	A~2
E	A^2^1	
F	A^3	
G	A^1^1^1	A~3
H	B^2	A^^^2
I	F^1	A^^3^1
J	F^2	A^^3^2

Ranges

Args	Expanded	Selected
D		G H D
D F		G H I J D F
^G D		H D
^D B		E I J F B
^D B C		E I J F B C
C		I J F C
B..C	= ^B C	C
B...C	= B ^F C	G H D E B C
B^..	= B^..B	
	= ^B^1 B	E I J F B
C^@	= C^1	
	= F I J F	
B^@	= B^1 B^2 B^3	D G H E F I J
C^!	= D E F	
	= C ^C^@	
	= C ^C^1	C
	= C ^F	
B^!	= B ^B^@	B
	= B ^D ^E ^F	G H D F
F^! D	= F ^! ^J D	

9 Scripting
Hooks
Client Side: Pre and post- events like commit, push, etc, git can run script in .git/hooks/ directory. These are excluded from version control. **Server Side:** Similar to client side, but are included in pushes, and used to implement controlled team processes. **GitHub:** Upon <event> (often a push), GitHub emits an HTTP post to indicated url. The server listening there can parse, interpret & act according to arbitrary logic.
Github API
GET a public HTTP end point like this:
git curl <EP> # basic api GET request
Or authenticate (token or OAuth) then POST:
curl -H "ContentType: application/json" \
-H "Authorization: token TOK \"
--data '<json>' <EP>
Top level end-points:

- activity
- data
- gists
- actions
- interactions
- issues
- organizations
- projects
- PRs
- reactions
- teams
- users