Everyday GIT With 20 Commands Or So

[Basic Repository] commands are needed by people who have a repository --- that is everybody, because every working tree of git is a repository.

In addition, [Individual Developer (Standalone)] commands are essential for anybody who makes a commit, even for somebody who works alone.

If you work with other people, you will need commands listed in the [Individual Developer (Participant)] section as well.

People who play the [Integrator] role need to learn some more commands in addition to the above.

[Repository Administration] commands are for system administrators who are responsible for the care and feeding of git repositories.

Basic Repository

Everybody uses these commands to maintain git repositories.

- git-init(1) or git-clone(1) to create a new repository.
- git-fsck(1) to check the repository for errors.
- git-gc(1) to do common housekeeping tasks such as repack and prune.

Examples

Check health and remove cruft.

```
$ git fsck (1)
$ git count-objects (2)
$ git gc (3)
```

1. running without `--full` is usually cheap and assures the repository health reasonably well.
2. check how many loose objects there are and how much disk space is wasted by not repacking.
3. repacks the local repository and performs other housekeeping tasks.

Repack a small project into single pack.

```
$ git gc (1)
```

1. pack all the objects reachable from the refs into one pack, then remove the other packs.

Individual Developer (Standalone)

A standalone individual developer does not exchange patches with other people, and works alone in a single repository, using the following commands.

- git-show-branch(1) to see where you are.
- git-log(1) to see what happened.
- git-checkout(1) and git-branch(1) to switch branches.
- git-add(1) to manage the index file.
- git-diff(1) and git-status(1) to see what you are in the middle of doing.
- git-commit(1) to advance the current branch.
- git-reset(1) and git-checkout(1) (with pathname parameters) to undo changes.
- git-merge(1) to merge between local branches.
- git-rebase(1) to maintain topic branches.
- git-tag(1) to mark known point.

Examples

Use a tarball as a starting point for a new repository.

```
$ tar zxf frotz.tar.gz
$ cd frotz
$ git init
$ git add . (1)
$ git commit -m "import of frotz source tree."
$ git tag v2.43 (2)
```

1. add everything under the current directory.
2. make a lightweight, unannotated tag.

Create a topic branch and develop.

```
$ git checkout -b alsa-audio (1)
$ edit/compile/test
$ git checkout -- curses/ux_audio_oss.c (2)
```

```
$ git add curses/ux_audio_alsa.c (3)
$ edit/compile/test
$ git diff HEAD (4)
$ git commit -a -s (5)
$ edit/compile/test
$ git reset --soft HEAD^ (6)
$ edit/compile/test
$ git diff ORIG_HEAD (7)
$ git commit -a -c ORIG_HEAD (8)
$ git checkout master (9)
$ git merge alsa-audio (10)
$ git log --since='3 days ago' (11)
$ git log v2.43.. curses/ (12)
```

1. create a new topic branch.
2. revert your botched changes in `curses/ux_audio_oss.c`.
3. you need to tell git if you added a new file; removal and modification will be caught if you do `git commit -a` later.
4. to see what changes you are committing.
5. commit everything as you have tested, with your sign-off.
6. take the last commit back, keeping what is in the working tree.
7. look at the changes since the premature commit we took back.
8. redo the commit undone in the previous step, using the message you originally wrote.
9. switch to the master branch.
10. merge a topic branch into your master branch.
11. review commit logs; other forms to limit output can be combined and include `--max-count=10` (show 10 commits), `--until=2005-12-10`, etc.
12. view only the changes that touch what's in `curses/` directory, since `v2.43` tag.

Individual Developer (Participant)

A developer working as a participant in a group project needs to learn how to communicate with others, and uses these commands in addition to the ones needed by a standalone developer.

- git-clone(1) from the upstream to prime your local repository.
- git-pull(1) and git-fetch(1) from "origin" to keep up-to-date with the upstream.
- git-push(1) to shared repository, if you adopt CVS style shared repository workflow.

- git-format-patch(1) to prepare e-mail submission, if you adopt Linux kernel-style public forum workflow.

Examples

Clone the upstream and work on it. Feed changes to upstream.

```
$ git clone git://git.kernel.org/pub/scm/.../torvalds/linux-2.6 my2.6
$ cd my2.6
$ edit/compile/test; git commit -a -s (1)
$ git format-patch origin (2)
$ git pull (3)
$ git log -p ORIG_HEAD.. arch/i386 include/asm-i386 (4)
$ git pull git://git.kernel.org/pub/.../jgarzik/libata-dev.git ALL (5)
$ git reset --hard ORIG_HEAD (6)
$ git gc (7)
$ git fetch --tags (8)
```

1. repeat as needed.
2. extract patches from your branch for e-mail submission.
3. `git pull` fetches from `origin` by default and merges into the current branch.
4. immediately after pulling, look at the changes done upstream since last time we checked, only in the area we are interested in.
5. fetch from a specific branch from a specific repository and merge.
6. revert the pull.
7. garbage collect leftover objects from reverted pull.
8. from time to time, obtain official tags from the `origin` and store them under `.git/refs/tags/`.

Push into another repository.

```
satellite$ git clone mothership:frotz frotz (1)
satellite$ cd frotz
satellite$ git config --get-regexp '^(remote|branch)\.' (2)
remote.origin.url mothership:frotz
remote.origin.fetch refs/heads/*:refs/remotes/origin/*
branch.master.remote origin
branch.master.merge refs/heads/master
satellite$ git config remote.origin.push \
           master:refs/remotes/satellite/master (3)
satellite$ edit/compile/test/commit
satellite$ git push origin (4)

mothership$ cd frotz
mothership$ git checkout master
mothership$ git merge satellite/master (5)
```

1. mothership machine has a frotz repository under your home directory;

clone from it to start a repository on the satellite machine.

2. clone sets these configuration variables by default. It arranges `git pull` to fetch and store the branches of mothership machine to local `remotes/origin/*` tracking branches.
3. arrange `git push` to push local `master` branch to `remotes/satellite/master` branch of the mothership machine.
4. push will stash our work away on `remotes/satellite/master` tracking branch on the mothership machine. You could use this as a back-up method.
5. on mothership machine, merge the work done on the satellite machine into the master branch.

Branch off of a specific tag.

```
$ git checkout -b private2.6.14 v2.6.14 (1)
$ edit/compile/test; git commit -a
$ git checkout master
$ git format-patch -k -m --stdout v2.6.14..private2.6.14 |
  git am -3 -k (2)
```

1. create a private branch based on a well known (but somewhat behind) tag.
2. forward port all changes in `private2.6.14` branch to `master` branch without a formal "merging".

Integrator

A fairly central person acting as the integrator in a group project receives changes made by others, reviews and integrates them and publishes the result for others to use, using these commands in addition to the ones needed by participants.

- git-am(1) to apply patches e-mailed in from your contributors.
- git-pull(1) to merge from your trusted lieutenants.
- git-format-patch(1) to prepare and send suggested alternative to contributors.
- git-revert(1) to undo botched commits.
- git-push(1) to publish the bleeding edge.

Examples

My typical GIT day.

```
$ git status (1)
$ git show-branch (2)
$ mailx (3)
& s 2 3 4 5 ./+to-apply
& s 7 8 ./+hold-linus
& q
$ git checkout -b topic/one master
$ git am -3 -i -s -u ./+to-apply (4)
$ compile/test
$ git checkout -b hold/linus && git am -3 -i -s -u ./+hold-linus (5)
$ git checkout topic/one && git rebase master (6)
$ git checkout pu && git reset --hard next (7)
$ git merge topic/one topic/two && git merge hold/linus (8)
$ git checkout maint
$ git cherry-pick master~4 (9)
$ compile/test
$ git tag -s -m "GIT 0.99.9x" v0.99.9x (10)
$ git fetch ko && git show-branch master maint 'tags/ko-*' (11)
$ git push ko (12)
$ git push ko v0.99.9x (13)
```

1. see what I was in the middle of doing, if any.
2. see what topic branches I have and think about how ready they are.
3. read mails, save ones that are applicable, and save others that are not quite ready.
4. apply them, interactively, with my sign-offs.
5. create topic branch as needed and apply, again with my sign-offs.
6. rebase internal topic branch that has not been merged to the master, nor exposed as a part of a stable branch.
7. restart `pu` every time from the next.
8. and bundle topic branches still cooking.
9. backport a critical fix.
10. create a signed tag.
11. make sure I did not accidentally rewind master beyond what I already pushed out. `ko` shorthand points at the repository I have at kernel.org, and looks like this:

```
$ cat .git/remotes/ko
URL: kernel.org:/pub/scm/git/git.git
Pull: master:refs/tags/ko-master
Pull: next:refs/tags/ko-next
Pull: maint:refs/tags/ko-maint
Push: master
Push: next
Push: +pu
Push: maint
```

In the output from `git show-branch`, `master` should have everything `ko-master` has, and `next` should have everything `ko-next` has.

12. push out the bleeding edge.
13. push the tag out, too.

Repository Administration

A repository administrator uses the following tools to set up and maintain access to the repository by developers.

- git-daemon(1) to allow anonymous download from repository.
- git-shell(1) can be used as a *restricted login shell* for shared central repository users.

update hook howto has a good example of managing a shared central repository.

Examples

We assume the following in /etc/services

```
$ grep 9418 /etc/services
git             9418/tcp                # Git Version Control System
```

Run git-daemon to serve /pub/scm from inetd.

```
$ grep git /etc/inetd.conf
git     stream  tcp     nowait  nobody \
  /usr/bin/git-daemon git-daemon --inetd --export-all /pub/scm
```

The actual configuration line should be on one line.

Run git-daemon to serve /pub/scm from xinetd.

```
$ cat /etc/xinetd.d/git-daemon
# default: off
# description: The git server offers access to git repositories
service git
{
        disable = no
        type            = UNLISTED
        port            = 9418
        socket_type     = stream
        wait            = no
        user            = nobody
        server          = /usr/bin/git-daemon
        server_args     = --inetd --export-all --base-path=/pub/scm
        log_on_failure  += USERID
}
```

Check your xinetd(8) documentation and setup, this is from a Fedora system. Others might be different.

Give push/pull only access to developers.

```
$ grep git /etc/passwd (1)
alice:x:1000:1000::/home/alice:/usr/bin/git-shell
bob:x:1001:1001::/home/bob:/usr/bin/git-shell
cindy:x:1002:1002::/home/cindy:/usr/bin/git-shell
david:x:1003:1003::/home/david:/usr/bin/git-shell
$ grep git /etc/shells (2)
/usr/bin/git-shell
```

1. log-in shell is set to /usr/bin/git-shell, which does not allow anything but `git push` and `git pull`. The users should get an ssh access to the machine.
2. in many distributions /etc/shells needs to list what is used as the login shell.

CVS-style shared repository.

```
$ grep git /etc/group (1)
git:x:9418:alice,bob,cindy,david
$ cd /home/devo.git
$ ls -l (2)
  lrwxrwxrwx   1 david git    17 Dec  4 22:40 HEAD -> refs/heads/master
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 branches
  -rw-rw-r--   1 david git    84 Dec  4 22:40 config
  -rw-rw-r--   1 david git    58 Dec  4 22:40 description
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 hooks
  -rw-rw-r--   1 david git 37504 Dec  4 22:40 index
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 info
  drwxrwsr-x   4 david git  4096 Dec  4 22:40 objects
  drwxrwsr-x   4 david git  4096 Nov  7 14:58 refs
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 remotes
$ ls -l hooks/update (3)
  -r-xr-xr-x   1 david git  3536 Dec  4 22:40 update
$ cat info/allowed-users (4)
refs/heads/master       alice\|cindy
refs/heads/doc-update   bob
refs/tags/v[0-9]*       david
```

1. place the developers into the same git group.
2. and make the shared repository writable by the group.
3. use update-hook example by Carl from Documentation/howto/ for branch policy control.
4. alice and cindy can push into master, only bob can push into doc-update. david is the release manager and is the only person who can create and push version tags.

HTTP server to support dumb protocol transfer.

```
dev$ git update-server-info (1)
dev$ ftp user@isp.example.com (2)
```

```
ftp> cp -r .git /home/user/myproject.git
```

1. make sure your info/refs and objects/info/packs are up-to-date
2. upload to public HTTP server hosted by your ISP.

Last updated 2009-07-01 02:31:11 UTC