giteveryday - A useful minimum set of commands for Everyday Git:

> SYNOPSIS

Everyday Git With 20 Commands Or So

> DESCRIPTION:

Git users can broadly be grouped into four categories for the purposes of describing here a small set of useful command for everyday Git.

- **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.

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-init(1) to create a new repository.
- 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 a 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 .  #add everything under the current directory.
$ git commit -m "import of frotz source tree."
$ git tag v2.43 #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 diff HEAD^ (6)
$ git commit -a --amend (7)
$ git checkout master (8)
$ git merge alsa-audio (9)
$ git log --since='3 days ago' (10)
$ git log v2.43.. curses/ (11)

 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. look at all your changes including the previous commit.
 7. amend the previous commit, adding all your new changes, using
 your original message.
 8. switch to the master branch.
 9. merge a topic branch into your master branch.
 10. review commit logs; other forms to limit output can be
combined
 and include -10 (to show up to 10 commits), --until=2005-12-10,
 etc.
 11. 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.
- git-send-email(1) to send your e-mail submission without corruption by your MUA.
- git-request-pull(1) to create a summary of changes for your upstream to pull.

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
   $ git checkout -b mine master (1)
   $ edit/compile/test; git commit -a -s (2)
   $ git format-patch master (3)
   $ git send-email --to="person <email@example.com>" 00*.patch
(4)
   $ git checkout master (5)
   $ git pull (6)
   $ git log -p ORIG_HEAD.. arch/i386 include/asm-i386 (7)
   $ git ls-remote --heads http://git.kernel.org/.../jgarzik
/libata-dev.git (8)
   $ git pull git://git.kernel.org/pub/.../jgarzik/libata-dev.git
ALL (9)
   $ git reset --hard ORIG_HEAD (10)
   $ git gc (11)
```

1. checkout a new branch mine from master.
2. repeat as needed.
3. extract patches from your branch, relative to master,
4. and email them.
5. return to master, ready to see what's new
6. git pull fetches from origin by default and merges into the current branch.
7. immediately after pulling, look at the changes done upstream since last time we checked, only in the area we are interested in.
8. check the branch names in an external repository (if not known).
9. fetch from a specific branch ALL from a specific repository and merge it.
10. revert the pull.
11. garbage collect leftover objects from reverted pull.

> 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 \
                +refs/heads/*:refs/remotes/satellite/* (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/* remote-tracking branches.
 3. arrange git push to push all local branches to their
 corresponding branch of the mothership machine.
 4. push will stash all our work away on remotes/satellite/*
 remote-tracking branches on the mothership machine. You could
use
 this as a back-up method. Likewise, you can pretend that
mothership
 "fetched" from you (useful when access is one sided).
 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 cherry-pick v2.6.14..private2.6.14 (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". Or longhand

     git format-patch -k -m --stdout v2.6.14..private2.6.14 | git
am -3
     -k
```

An alternate participant submission mechanism is using the git request-pull or pull-request mechanisms (e.g as used on GitHub (www.github.com) to notify your upstream of your contribution.

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**.

This section can also be used by those who respond to git request-pull or pull-request on GitHub (www.github.com) to integrate the work of others into their history. An sub-area lieutenant for a repository will act both as a participant and as an integrator.

- 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

> A typical integrator’s Git day.

```
    $ git status (1)
    $ git branch --no-merged master (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 ./+to-apply (4)
    $ compile/test
    $ git checkout -b hold/linus && git am -3 -i -s ./+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 && for branch in master maint next pu (11)
        do
            git show-branch ko/$branch $branch (12)
        done
    $ git push --follow-tags ko (13)
```

```
 1. see what you were in the middle of doing, if anything.
 2. see which branches haven't been merged into master yet.
Likewise
 for any other integration branches e.g.  maint, next and pu
 (potential updates).
 3. read mails, save ones that are applicable, and save others
that
 are not quite ready (other mail readers are available).
 4. apply them, interactively, with your sign-offs.
 5. create topic branch as needed and apply, again with sign-
offs.
 6. rebase internal topic branch that has not been merged to the
 master or 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 master was not accidentally rewound beyond that
 already pushed out.
 12. In the output from git show-branch, master should have
 everything ko/master has, and next should have everything
ko/next
 has, etc.
 13. push out the bleeding edge, together with new tags that
point
 into the pushed history.
```

> In this example, the ko shorthand points at the Git maintainer's repository at kernel.org, and looks like this:

```
    (in .git/config)
    [remote "ko"]
            url = kernel.org:/pub/scm/git/git.git
            fetch = refs/heads/*:refs/remotes/ko/*
            push = refs/heads/master
```

```
        push = refs/heads/next
        push = +refs/heads/pu
        push = refs/heads/maint
```

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.
- git-http-backend(1) provides a server side implementation Git-over-HTTP ("Smart http") allowing both fetch and push services.
- gitweb(1) provides a web front-end to Git repositories, which can be set-up using the git-instaweb(1) script.

> *update hook howto[1] has a good example of managing a shared central repository.*

> In addition there are a number of other widely deployed hosting browsing and reviewing solutions such as:

`gitolite, gerrit code review, cgit and others.`

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 using git-over-ssh. e.g. those using: `$ git push/pull ssh://host.xz/pub/scm/project`

```
    $ 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 require ssh
access to
     the machine.
     1. 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.
```

GIT Part of the git(1) suite

NOTES

1. update hook howto file:///usr/share/doc/git/html/howto/update-hook-example.html

Git 2.17.1 12/09/2019 GITEVERYDAY(7)