Git

Introduction

Today's world is fast moving. Adapting to new tools is real bonus. There are many version control tool available in market like clearcase, cvs, svn, Perforce, git. All posses good & bad. However even after that git popularity is more.

Author: Linus Torvalds

Year : 2005

Before this most people were engaged with **Bitkeeper** but it was not distributed. Linus wanted a repository that could behave same as Bitkeeper but also should be distributed. He sighted the fact that today's world where we have increased number of contributers accross the world, Central Repository system may failed. Hence he came up with a design and now we know as **GIT**. This looks stupid but he completed the task in 3 months and handed this to **Junio Hamano** for maintenance..

Installing GIT on your system

UNIX/LINUX:

Based on system people can use the below link to install.
Unix/Linux: GIT INSTALLER: [1]

Download for Linux and Unix

It is easiest to install Git on Linux using the preferred package manager of your Linux distribution. If you prefer to build from source, you can find the tarballs on kernel.org.

Debian/Ubuntu

For the latest stable version for your release of Debian/Ubuntu

```
# apt-get install git
```

For Ubuntu, this PPA provides the latest stable upstream Git version

```
# add-apt-repository ppa:git-core/ppa # apt update; apt install git
```

Fedora

`# yum install git` (up to Fedora 21)
`# dnf install git` (Fedora 22 and later)

Gentoo

```
# emerge --ask --verbose dev-vcs/git
```

Arch Linux

```
# pacman -S git
```

openSUSE

```
# zypper install git
```

Mageia

```
# urpmi git
```

Nix/NixOS

```
# nix-env -i git
```

FreeBSD

```
# pkg install git
```

Solaris 9/10/11 (OpenCSW)

```
# pkgutil -i git
```

Solaris 11 Express

```
# pkg install developer/versioning/git
```

OpenBSD

```
# pkg_add git
```

Alpine

```
$ apk add git
```

Red Hat Enterprise Linux, Oracle Linux, CentOS, Scientific Linux, et al.

RHEL and derivatives typically ship older versions of git. You can download a tarball and build from source, or use a 3rd-party repository such as the IUS Community Project to obtain a more recent version of git.

Slitaz

```
$ tazpkg get-install git
```

Windows:

Windows Git Installer[2]

Mac:

Mac Git Installer: [3]

Git How to play

Please note that from here i will we explaining the git concepts in linux .

The first step while playing with GIT is to **create a your own work space**. Quite true since other will not allow you to disturb theirs.

Let us say i have a work space named **my_ws.**

mkdir my_ws;

cd my_ws.

Git Initialization:

This is the first mandatory step in git.

git init --> This creates .git folder. The contents of that folder are as below

1.HEAD ==> There is a reference to the current branch in the HEAD file. At the moment it must be the master branch
2.branches ==> Not of Much Importance
3.config ==> It posses the configuration settings for work space.
4.description ==> This has list of all versionised part of your work space.
5.hooks ==> This contains a set of scripts , which GIT internally uses.
6.info ==> This can be used as a tool to ignore files from creating version.
7.objects ==> Git's internal Metadata of blobs, all indexed by SHAs.
8.refs ==> This contains a vital info on the Git branches we are working , Commit Id etc.

Git Clone:

The next step is of-course to have a source to work with. Yes to do so Git uses the command Git Clone.

Basic Syntax

git clone <remote> <dir> ==> Here remote is the location provided by you to download(https or ssh or local). Dir is optional.

If not provided current dir will be used.

Git Status :

The changes we made on a branch in the repository can be seen using

git status --> This will list both the modifications in the existing files and also the newly added files.

Git Diff :

The changes we did in a specific file/branch can be compared with the recently synced mainline using

git diff <file_name> --> gives the diff of a particular file
git diff . --> gives the diff of all changes on the present branch

Git Clean :

The temporary/Auto-Generated files such as object generated while compiling may not be needed to push to Main Line. These can be removed using

git clean -n --> Lists the Untracked/New files on the present branch that would be removed.
git clean -fd --> Removes the Untracked files and echoes the files that are removed.
git clean -fd <file> --> Removes the particular file

Git Add :

The moment we make the changes Git keep them either in unstaged changes or untracked changes.

Let us give some time to have a look into unstaged changes and untracked changes.

Unstaged Changes ---> Git indicates the user that the file has been modified against the last commit.

Before Changes

Git --file 1 ---> vers 3
After Changes

Git --file 1 ---> vers 3 + unstaged changes(Not ver 4 , since yet not committed).

Untracked Changes -----> Git will report this if any new object that has yet not made as a part of the Git Tracker.

git add <object name> --> This indicates the git which all object it can consider for commit.

If you specify a dir , it will search the entire dir and pull out all the files into stagging area.

Git may have given the option of direct commit but as a safe side it broke the work into stagging then to commit.

Some examples :

git add file1
git add dir1
git add dir1 file1
git add . --> Adds all the changes(Modified & Untracked) on the branch to Staging Area

Git Checkout :

We can also sometimes remove the changes that we did accidentally on a branch in the repository through

git checkout <path/file_name> --> removes the modifications done in specific file
git checkout --> removes the modifications done in the repository

Git Commit :

The next step after adding the changes to stagging area is to committing the changes. So when we are ready we can commit the changes.

Please be noted that the commit is to local work space and not the server.

git commit -m "User msg" <repository>
or
git commit --amend <repository> {This will put the commit to last committed}

The moment we throw up the commit command git collect the bundle of files as part of the commit. It creates a Unique Id "Commit Id".
We can use that as a reference to obtain the changes of that commit id.

As a simple example Commit Id can be same a Exam Roll no. Just like by providing the Roll no we get access to all Info on the students ,
same way we can access the info of changes using commit id.

Git Log :

If we want to know the commits done on a specific file, we can use the

git log <path/file_name> --> lists the commits with commit message, commit id, commiter.

Git reflog :

If we want to know the commits done on a specific branch of a repository, We can use

git reflog --> Lists all the commits with commit id's and commit message and also lists the sync done in the repository.

Git Reset :

Case 1:

If you want to remove a file that is added to Staging Area, We cannot do ' git checkout <file>' as it was not in Unstaged state.
So, first we have to Unstage that file and do 'git checkout <file>'. To unstage a file from Staging Area,

git reset HEAD <File> --> Unsstages the file from staging area.

Now you can do git checkout <file>, if it is not a newly added file else we have to do git clean -fd <file>

Case 2:

We have already committed the changes to repository, but we have to remove a file from the committed patch which was accidentally pushed to repository.

This can be done using,

git reset --soft HEAD --> This will shows us the pushed changes in the previous commit as Staged Changes.

Now, can we can use the case 1 to remove the modification or else if it is a new file we have to remove it.
After the Needed change we can proceed with 'git commit --amend <message>' and then push to remote repository.

Git Branch :

We can create multiple branches in our repository by,
git branch -b <branch_name>

We can switch the branches by,
git checkout <brnach_name>

We can delete the branches by,
git branch -d <branch_name> --> Note you cannot delete the branch on which you are present

We can see the branches in our repository by,
git branch

Git Stash :

We need to commit/checkout the changes before switching to other branch in the repository, There is another way through which we can switch the branches in the
repository with out committing or loosing the changes by stashing the changes. Stash will save your changes or record your changes in a list and revert the working
directory to match the HEAD commit.

git stash <stash_name> --> will record/stash your uncommitted changes
git stash list --> Lists the stashed changes in the repository as WIP on <branch on which the changes are stashed>
git stash apply --> Applies the last stashed changes.

Note : The new/untracked files will not be stashed and will not be effected by branch switching (i.e, they will be present on that branch too).

We can save multiple stashes and apply multiple stashes when required on the branch and we can choose specific stash by,

git stash apply <index>

Sometimes after applying stashed changes we might get merge conflicts which have to be handled manually and after applying the stashed changes we can remove that
stash by

git stash drop/pop <index>

Git Tips :

NAME:

git-blame - Show what revision and author last modified each line of a file.

DESCRIPTION:

Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.

Apart from supporting file annotation, Git also supports searching the development history for when a code snippet occurred in a change. This makes it possible

to track when a code snippet was added to a file, moved or copied between files, and eventually deleted or replaced.It works by searching for a text string in the d iff.

Example:

To find out who changed a file, you can run git blame against a single file, and you get a breakdown of the file, line-by-line, with the change that

last affected that line. It also prints out the timestamp and author information as well:

```
$ git blame file
566a0863 (Alex Blewitt 2011-07-12 09:43:39 +0100 1) First line
ed0a7c55 (Alex Blewitt 2011-07-12 09:43:51 +0100 2) Second line
8372b725 (Alex Blewitt 2011-07-12 09:44:06 +0100 3) Third line
ed0a7c55 (Alex Blewitt 2011-07-12 09:43:51 +0100 4)
```

The timestamps and abbreviated commit hashes show the changes were introduced sequentially, but in this contrived example it's easy to see.

Since Git has full information about the committer, it can show you the person's name, or the persons e-mail address:

```
$ git blame -e file
566a0863 ( 2011-07-12 09:43:39 +0100 1) First line
ed0a7c55 ( 2011-07-12 09:43:51 +0100 2) Second line
8372b725 ( 2011-07-12 09:44:06 +0100 3) Third line
ed0a7c55 ( 2011-07-12 09:43:51 +0100 4)
```

Sometimes you get extra noise when whitespace differences are introduced into the file. You can use -w to suppress reporting on changes that only affected whitespace.

There are also changes you can show on a subset of ranges as well (like git diff, but with lines annotated with the commit hash). For example:

```
$ git blame ed0a..566a -- file
^566a086 (Alex Blewitt 2011-07-12 09:43:39 +0100 1) First line
```

Finally, if you have large files, it can often generate more output than is necessary. You can post-filter the results, but it's more efficient to tell Git

which lines you want to see so that it doesn't need to do more work than is necessary finding out. You can specify lines with an explicit line number,

(start/end), an offset from the start line (valid for end only) or a regular expression. This can be useful to blame a specific function, if you are following

a C-like formatting where the function begins in column zero. This allows us to see the blame of a specific function (that starting with ^bar) and to the next

closing brace (ending with ^} after the ^bar):

```
$ git blame tst.c
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100  1) foo() {
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100  2)    // the foo function
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100  3) }
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100  4)
0cdc3645 (Alex Blewitt 2011-07-12 09:58:15 +0100  5) bar() {
0cdc3645 (Alex Blewitt 2011-07-12 09:58:15 +0100  6)    // the bar function
0cdc3645 (Alex Blewitt 2011-07-12 09:58:15 +0100  7) }
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100  8)
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100  9) main() {
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100 10)    // the main function
6bee4066 (Alex Blewitt 2011-07-12 09:57:54 +0100 11) }
$ git blame -L/^bar/,/^}/ tst.c
0cdc3645 (Alex Blewitt 2011-07-12 09:58:15 +0100 5) bar() {
0cdc3645 (Alex Blewitt 2011-07-12 09:58:15 +0100 6)     // the bar function
0cdc3645 (Alex Blewitt 2011-07-12 09:58:15 +0100 7) }
```