**GIT Version Control System**

**Ganesh Palnitkar**

** **

**GIT - Distributed Version Control System**

**Installation:**

**Windows:** Install git by downloading latest version for windows installable from <https://git-scm.com/downloads> URL. On windows GIT is available in three interfaces as ‘GIT Bash’, ‘GIT cmd’ and ‘GIT GUI’.

**Linux:** Install GIT on Linux by running ‘yum’ or ‘apt’ commands as below,

Debian / Ubuntu: $ sudo apt-get install git

**Centos 7:** $ sudo yum install git -y

**Centos 8:**

$ sudo dnf install git -y

$ git --version

**GIT Commands:**

1. Creating / initiating a fresh local repository,

$ git init

$ git init <foldername> … this will create a folder with the given name and initiate a repository inside it.

1. Copy / clone a remote repository to local

$ git clone <remote\_repo\_url>

$ git clone --mirror … creates a fresh tracking folder with all remote ref and tags

$ git clone --bare … create a new clone from remote with all tags but with no future fetch url associated.

$ git clone <remote\_url> --depth <last commits in number>

e.g.

$ git clone <remote\_url> --depth 3

1. Indexing / adding a file / set of files in git repo.

$ git add . … to index all modified / new files to local repo

$ git add <file name> .. index a particular file

1. Move or rename a file in the repository

An existing file in the local repository can be renamed or moved using git mv command.

$ git mv <file name> <new-file-name>

With this command the file is renamed, or moved to a new path with the change getting staged as well. That means the change is also updated in staging area.

1. Commit a file / object to local repository

$ git commit –m “suitable comment” … changes indexed are committed to the repository

1. Now to get a list of commands in git one can run the command,

$ git help –a

1. To find difference in the file changes in two versions.

$ git diff HEAD … or,

$ git diff rev1.0 rev1.1

$ git diff --cached … provides differences data between staging and repository area.

1. To update user name and email address for the user we can use,

$ git config –-local user.name “username”

$ git config –-local user.email “email@address”

.. and if we want to apply this to all repositories on the machine.

$ git config –-global user.email “email@address”

$ git config –-global user.name “username”

To verify the effect of config command, check the file .gitconfig located at user home directory.. (/home/<username>/.gitconfig)

1. Continuing with the configurations, we can use the config command to set the default branch name.

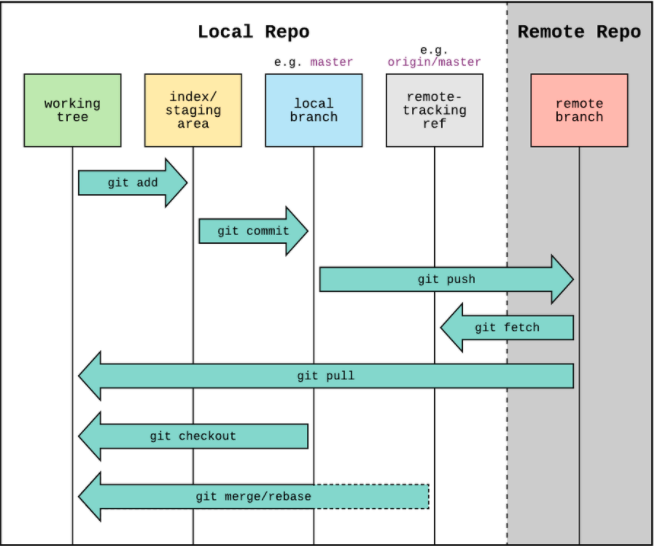
$ git config –global init.defaultBranch main

$ git branch -m main … here we are trying to rename the default branch name to main from ‘master’.

1. $ git log –p … detailed pretty output

$ git log …. List of all commits, version inside git repo.

1. $ git status … this commands gets information about the status of workspace



1. $ git checkout <branch-name> … moves the HEAD pointer from branch to branch.

or,

$ git checkout <commit hash> … checks out a version to working area (detached HEAD)

1. $ git rm -–cached <filename> … removed a file from staging back into working area. Or,

$ git restore -–staged <filename> … to unstage file from staging area

1. Renaming a file in the repository can be done by using $ git mv command.

$ git mv <src-file-name> <target-file-name>

1. To get details of each commit- has details, we use the command, show command.

$ git show <hash-object>

Both these commands have the same effect. Helps to get details of a hash object code that is, 2 commits earlier from the current HEAD.

$ git show HEAD~2

$ git show HEAD^^

**GIT Internals**

At core GIT is a map. It is a table with key and value control.

Value is any sequence of bytes. It is converted into a hash code (a key) with SHA1 algorithm.

A ‘sha1’ code can be generated using below syntax on the git bash prompt,

$ echo “string” | git hash-object --stdin

**Object Modelling.**

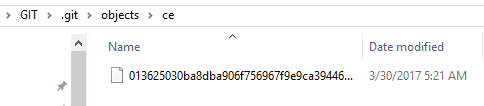
Every object in GIT has its own SHA1 value.

SHA1 values are unique in the Universe. There are very unlikely chances that there are two identical SHA1 code for different string value.

Example:

$ echo “hello” | git hash-object --stdin –w ….. This will write the sha1 value to repository by creating an object.

If we dig inside the .git/objects directory, we get to see an object as shown under,



**‘Objects’ is the object database directory…** the file starting with 013 is called the blob data file.

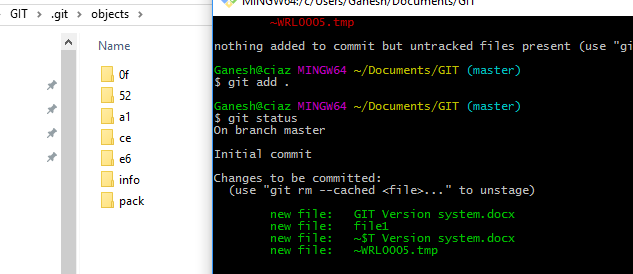
$ git cat-file <ce013….> -t …. Displays the file data type



$ git cat-file <013….> -p … displays the file contents.

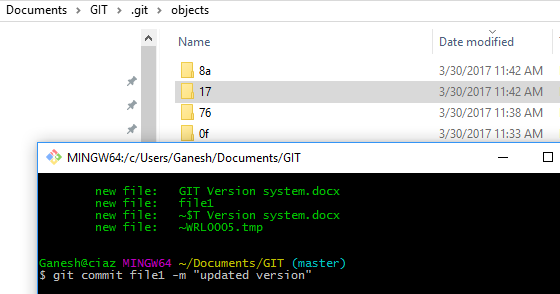
****

When a file is added to staging area, that’s when the object related to each file is created in the .git directory. This is as shown in the below image.

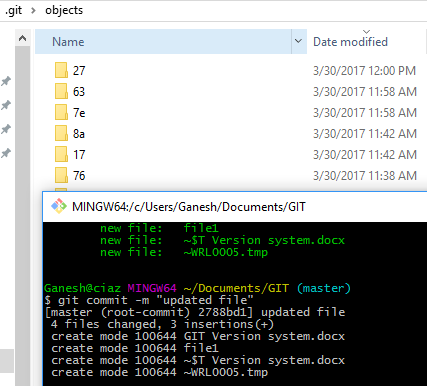


If a file is changed and committed to the repository, a new entry is added to the object folder for the new version of file. This means that that GIT creates a snapshot or a blob object with a ‘SHA1 hash’ for each version of file and preserves it.

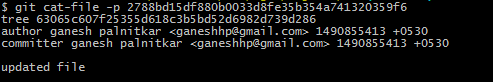
To reinstate a file to it earlier version the SHA1 has can be referred.



For every action of update to the git repository, GIT creates a SHA1 (snapshot) for the file version.

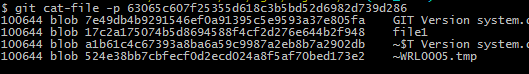


If we dig inside the 27\* directory and query the file sha1 code, we get below output.



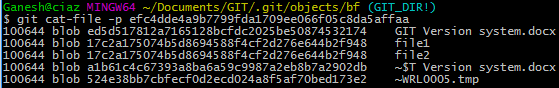
The commit ‘sha1’ code includes information about the committer, author and also the tree information as to which blob this commit is related to.

If we try to get information about the tree sha1 code, we get details about all the commits that included in it. For example,



./ commit (Tree)-🡪 (blob) file1 🡪 blob content. 🡪 (Tree) 🡪 …….

If we create files with same content the blob for these objects would be same. As shown in below image file1 and file2 has same contents thus the SHA1 is also the same. So GIT uses the information from the git database.



So a blob is not about the file but the contents of the file. The author and permission information about a file is stored in the **tree sha1** code of the file.

GIT stores all data in the form of SHA1 hash (snapshot) and not any more information about the file. This makes the GIT database very light weight. This is the total size of objects in the database.

**GIT count-objects:**

The count-objects command helps to get information on number of objects and files size.



**Tree Objects:**

Git stores file contents in a manner similar to a UNIX file system, but a bit simplified. All contents are stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more entries, each of which is the SHA-1 hash of a blob or sub tree with its associated mode, type, and filename.

$ git cat-file -p master^{tree}

100644 blob 9ae7bca0fcb3390675e369adc223fabd87bf8b08 .classpath

100644 blob 3217d4e4f09fa055870ad8caba9c7964cd5fdb5f .gitignore

100644 blob 7cf6fe3490d539dd25415162fa02bb2eebfae78b .gitlab-ci.yml

040000 tree 14617dd1f0ad007e824fe5e8d9f0727d2915d5d6 .gradle

100644 blob 7c88c1529f78b5cb63f7a042f6b0addb8a7158dd .project

040000 tree ae6ded6253f738f48854d95f08e953bbbd1431fb .scannerwork

040000 tree ff36a41064904ec22efad4f5fc6bc400ccad978a .settings

100644 blob e4a2a1a32fbe7171631bf745098a927cafa037f1 Dockerfile

100644 blob 854c284b50855c4c3f87ea7fb1b70fc1e7eb13ef Helloworldwebapp-dev.war

100644 blob 05cf3273053a9e25ce24176297a40ce8fa0fa057 Jenkinsfile

100644 blob a1af257fb3e8f0ef5df00851280d63180c41a943 Jenkinsfile.declarative

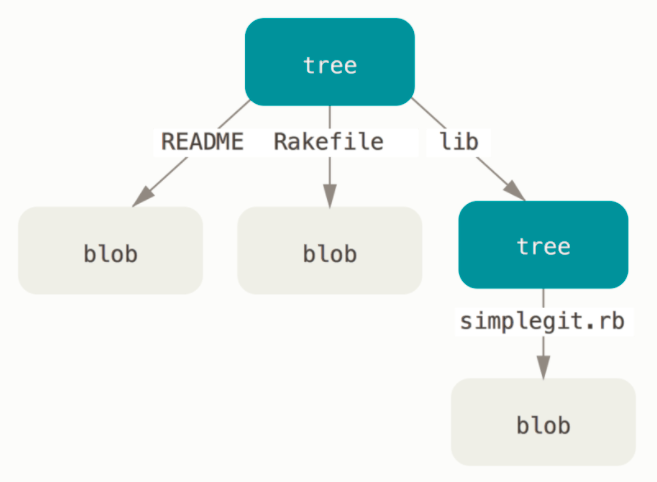
100644 blob f8d14b406f1eb24171799b35e00ca730859fd495 README.md

100644 blob c83063ffbb2d2b34befdc89d6982393f07b8f2f0 pom.xml

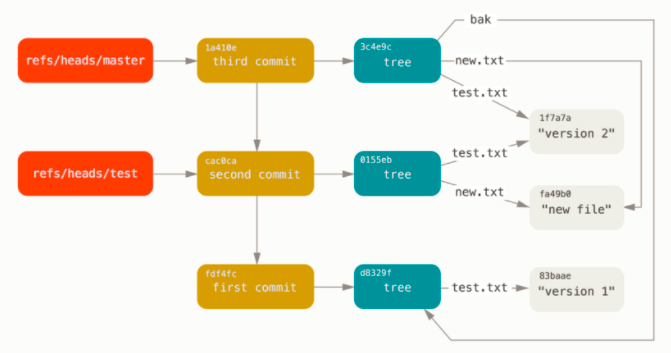
040000 tree 280d3d1517cef9c31c1acf5e851e3df7f6f76641 src

040000 tree 2da8cc1ecb9a2e370edec68351e657a61396c6e3 target

The master^{tree} syntax specifies the tree object that is pointed to by the last commit on your master branch. Notice that the src, target subdirectory isn’t a blob but a pointer to another tree:

****

If you follow all the internal pointers, you get an object graph something like this:

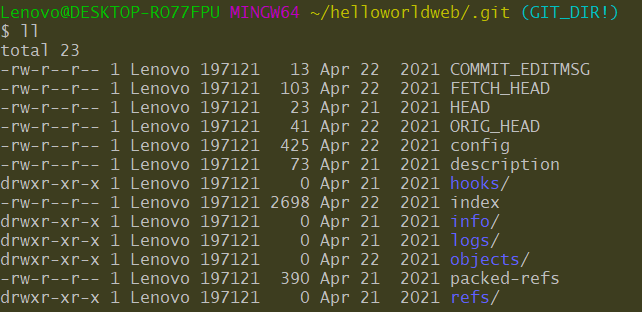


GIT directory objects with branch head references included.

**GIT Branches:**

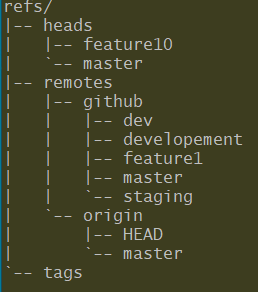
Branch is a reference or a pointer to a commit hash.

GIT keep all branch related data in the ref folder inside the .git folder. Inside the refs folder there are heads and tags folder.



Current active branch reference.

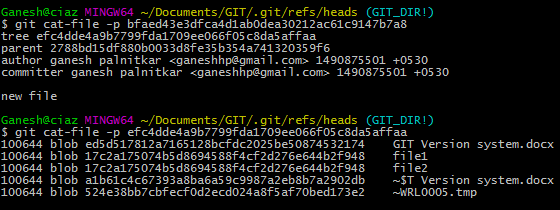
Refs folder contains branch head information and branch remote repository reference.



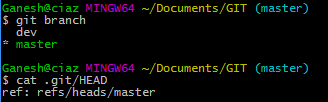
Let’s move into Refs folder.



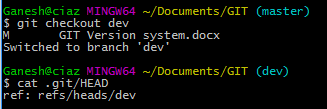
The SHA1 code is actually for the commit. If we try to get information about the SHA1, we see that the master branch points to a tree which in turn points to multiple blobs referring to files.



If we create a new branch ‘dev’ and then run the git branch command, it shows that we are currently on the ‘master’ branch. GIT has an entry for the current branch in the HEAD file.



If we switch the branch the ref in the HEAD file changes as shown below.



Switching between branches can be done using command,

$ git switch <branch\_name>

A checkout mean to GIT is to move head to new branch and update the workarea.

To restore the HEAD to an earlier status, we first need to know the exact SHA1 code of the commit to which we want to reset the HEAD to. This can be done by running the git reset command.

A branch can also be renamed. While you are in a branch dev, we want to rename the dev branch to development, then we use below command.

$ git branch –m development

**Branch Merge** Operations:

When we want to merge the feature / dev branch to the master branch, we do this by first switching the branch to master, using command,

$ git switch master … this will switch the current branch to ‘master’. Once in master branch we then can use the command,

$ git merge dev … considering the branch ‘dev’ is to be merged with master. We might see merge conflict depending on what files are merged to master from dev. GIT will try to auto resolve the conflict and if auto conflict is not possible, GIT will open the file in default editor and prompt the user for making changes to the conflicting file. **Once the file is edited and saved, the file has to be 🡪 staged and 🡪 committed to the repository.** Thus making the merge action complete.

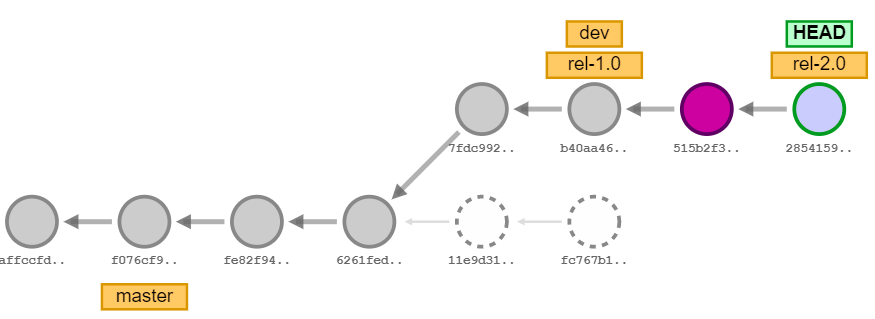
The commit hash created after merge operation will have two parent hash ref.

Check the ‘git-Merge-Vs-Rebase.docx’ for details.

Related to working with branches, there are three rules followed by git.,

1. The current branch tracks new commits.
2. When we move to another branch using switch or checkout, Git updates the working directory contents.
3. Unreachable objects and garbage collected.

**Branch Rebase** operation: Reapply commits on top of another base tip. Avoid rebasing a commit (branch) if it is shared to a repository (remote).

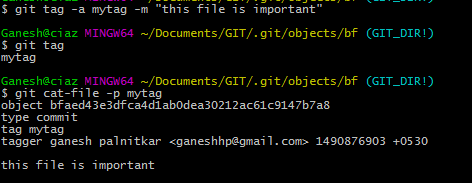


Detached commits hashes then deleted / garbage collected

New commits created as results of rebase

**GIT TAGS**

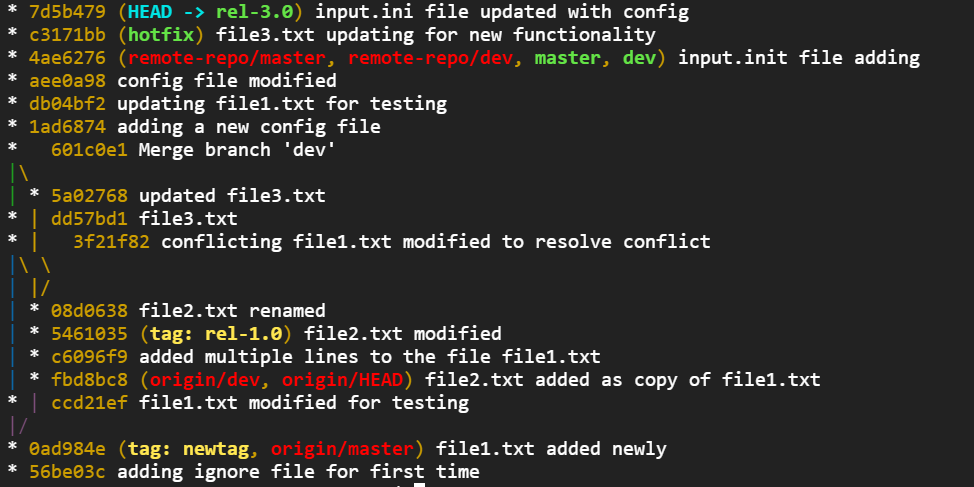
**Annotated** **tag**. This is the one that comes with a comment. Here a tag that shows the comment and also metadata information about the tag and points to an object, in this case a commit object. So, a tag is a simple label in GIT.

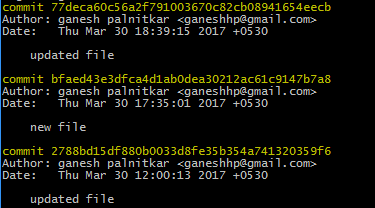


**List / view version log**

$ git log …. This command lists out all history with commit statements, date and time of commit and owner of commit, etc. this is as shown below.

$ git log --graph --oneline –decorate





Reset to

Current HEAD

**Reset** to a version in repository:

The log always shows the current HEAD first. Now if we want to reset the HEAD to the state as pointed by the arrow, we can use the command,

$ git reset bfaed43 …. This will reset the HEAD to the commit status as shown. Care has to be taken while resetting the HEAD to earlier state as all changes after this commit would be lost.

Reset command has three options, Hard, Mixed and Soft.

$ git reset --hard <hash-code> … file and hash object is removed from work directory and repo database.

$ git reset --mixed <hash-code> … files are moved into work directory, but hash object is removed from repo database (garbage collected)

$ git reset --soft <hash-code> … files are moved into staging and hash object for subsequent commits are removed (garbage collected).

**Stashoperations:**

$ git stash … this command gets applied to the files in work area. Once applied the files are moved to temporary storage area which git manages.

$ git stash apply … the apply option with stash gets the files back into work area.

$ git stash drop … the drop option deletes all files that area stashed.

$ list and show option are used to get details about the stashed operations performed.

$ git stash list … this will list all stashes as stash@{0}, stash@{1} ….

$ git stash show -p stash@{1} … this command shows details about a particular stash.

The latest stash created is stored in *refs/stash*. Older stashed are found in reflog of this reference and can be named using the usual reflog syntax *(e.g. stash@{0}* is the most recently created stash, *stash@{1}* is the one before it, *stash@{2.hours.ago}* is also possible).

Few more with Stash

$ git stash save "describe it" # give the stash a name

$ git stash clear # delete a stashed commit

$ git stash save --keep-index # stash only unstaged files

**Resolving Conflict:**



Open the conflicting file in editor to check and resolve conflict

Merge operation complete on committing the changes done with manually resolving conflicting text in the file

Merge operation reporting conflict

**Fork and Up-stream:**

**Origin**

**Pull and Push or Fetch**

**plusforum/helloworldweb**

**ganeshhp/helloworldweb**

**fork**

**Pull Request**

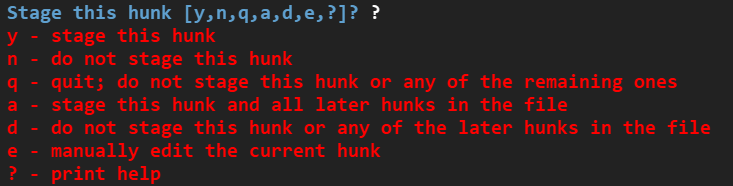
**Pull but no Push from upstream**

**Some of the ‘not-so-usual’ GIT commands:**

**Split a file (Hunk) while adding to staging:**

$ git add --patch <file-name>

--patch options allows the staging of selected file with selected part of the file only, we can also split the hunk into smaller parts so that we can decide to stage only part of the file (hunk).



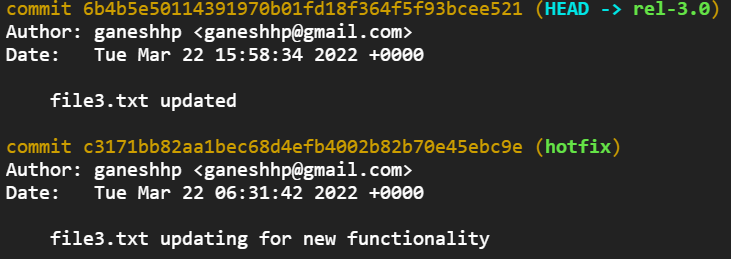
**Split a subfolder out into a new repository:**

Sometimes you may want to turn a specific folder within your Git repo into a [brand new repo](https://help.github.com/articles/splitting-a-subfolder-out-into-a-new-repository/). This can be done with git filter-branch:

**Working with History:**

$ git commit --amend this works only with latest commits … this will help to make changes to the last commit by allowing to add additional object that are staged to it. In this operation what git does, is it copies the last commit objects and adds new objects to it and creates a new commit. And the last commit that we want to amend gets garbage collected.

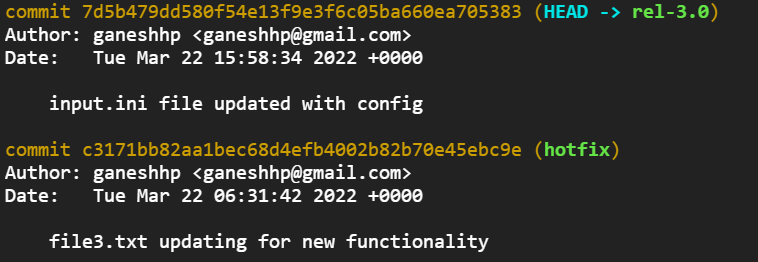
Before amending the last commit,



Commit to be amended

After running the commit with --amend option. We see the new change gets added to the previous commit and replaces the earlier commit hash.

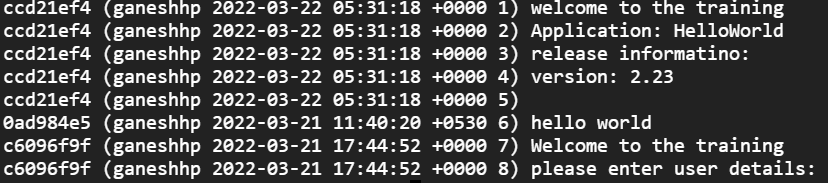
In fact, Git does not amend the last existing commit (Commits are immutable), but copies the contents of earlier commit and adds a new file hash object to it and creates a new commit hash.



New hash object added with last commit and new changes

**Git blame:**

$ git blame file1.txt …this command option gives details about when a file was updated and by whom with timestamp.



**Interactive Rebase**

$ git rebase --interactive

Allows us to run some commands (squash, pick, reword, drop) interactively while performing a Rebase.

**Rewriting Large Chunks of History**

Scenarios can be such that we have a large file in the repository that we want to get rid of. Just deleting the file may not help as the file has many versions in the repository database. One more scenario can be such that we have added a file in the GIT repo that contents critical information like passwords and we want to get rid of such file from all previous versions.

$ git filter-repo --path file\_name --invert-paths … in this command the file specified with the file\_name will be removed from all commits

$ git filter-branch --prune-empty --subdirectory-filter <folderName> master

# Filter the master branch to your directory and remove empty commits

Rewrite 48dc599c80e20527ed902928085e7861e6b3cbe6 (89/89)

Ref 'refs/heads/master' was rewritten

$ git filter-branch --index-filter 'git rm --cached --ignore-unmatch filename' HEAD ….. To remove a file from all commits:

The repository now contains all the files that were in the specified subfolder. Although all of your previous files have been removed, they still exist within the Git history. You can now push your new local repo to the remote.

**GIT cherry-pick**

Cherry picking in git means to choose a commit from one branch and apply it onto another.  If someone wants to commit specific *commits* in one branch to a target branch, then cherry-pick is used.

Checkout (switch to) target branch.

$ git cherry-pick <commit id>

Here commit id is activity id of another branch e.g.

$ git cherry-pick 4370c48c18aa560a3f8f716f16d11da94b8e31e6

$ git cherry-pick master

This applies the change introduced by commit at the tip of master and creates a new commit in target branch.

$ git cherry-pick master~4 master~2

Apply the changes introduced by the fifth and third last commits pointed to by master and create 2 new commits with these changes.

**GIT Clean**

$ git clean –fd … forcefully remove untracked files and folders

$ git clean –nfd .. list files and folders that will be removed

Seating up remote repository as default for PUSH and PULL operations.

$ git branch --set-upstream-to myfork/master

**GIT rev-list**

$ git rev-list foo bar ^baz

means "list all the commits which are reachable from foo or bar, but not from baz".

A special notation "<commit1>..<commit2>" can be used as a short-hand for "^'<commit1>' <commit2>". For example, either of the following may be used interchangeably:

$ git rev-list origin..HEAD

$ git rev-list HEAD ^origin

Another special notation is "<commit1>…​<commit2>" which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The following two commands are equivalent:

$ git rev-list A B --not $(git merge-base --all A B)

$ git rev-list A...B

**Distribution Model:**

Peer to Peer:

All local repositories can be pushed and pulled from other local repos.

Remote-local:

Have a common remote repository (blessed – mostly a Bare repo) to which all developers push and pull from. Remote repository becomes a source of common and all in one repo. Contributors who may not have write access to a repo, branch, use the PullRequest method to allow changes to be reviewed and merged into main branch.

**Designing branches:**

**Stable branch** and **Unstable branch:** a stable branch is the one that has the tip / Head of branch with working copy of code. Unstable opposite to the stable criteria.

**Integration Branch:** The branch to which changed diverged for development ultimately merged.

**Parallel branch:** Parallel branches although may have been derived from a common branch, run mostly parallel and merged only when code sharing is required.