GIT GUIDE

Leaving Card

August 20, 2018

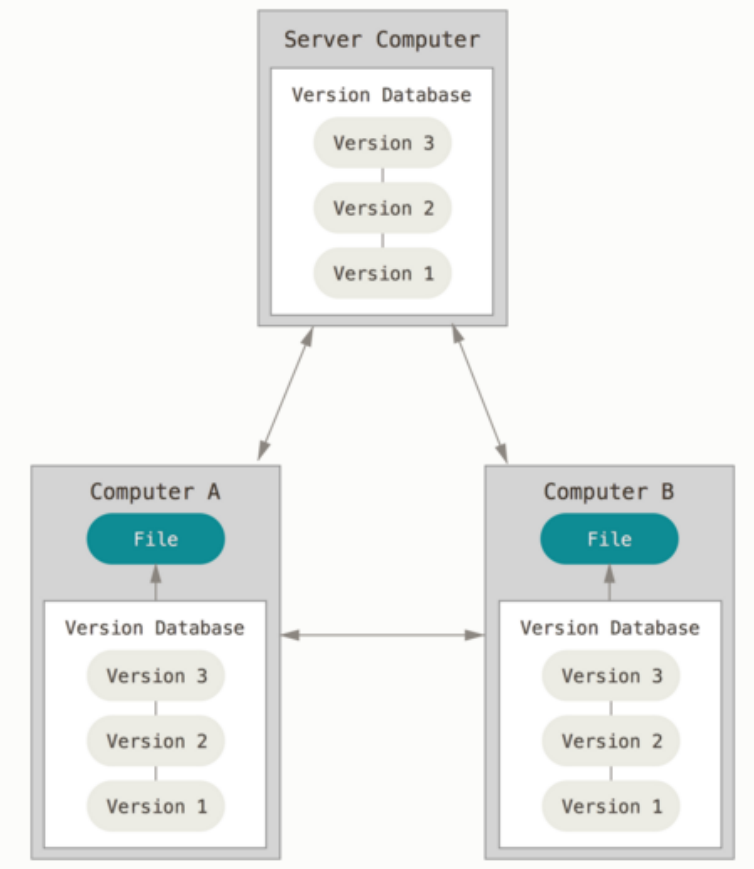
Contents

1 Git

1.1 Basic Concepts

1.1.1 Distributed Version Control Systems

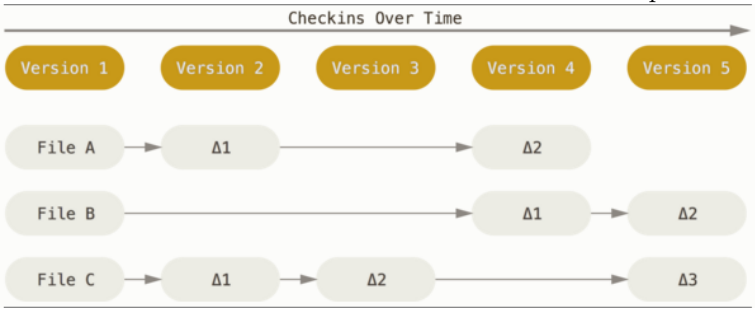
Clients dont just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

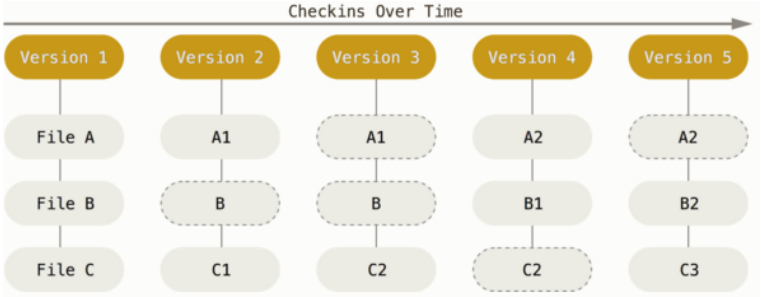


1.1.2 Difference between git an other file systems

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

Git doesnt think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.





1.1.3 Git is local

Most operations in Git only need local files and resources to operate. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

1.1.4 Integrity

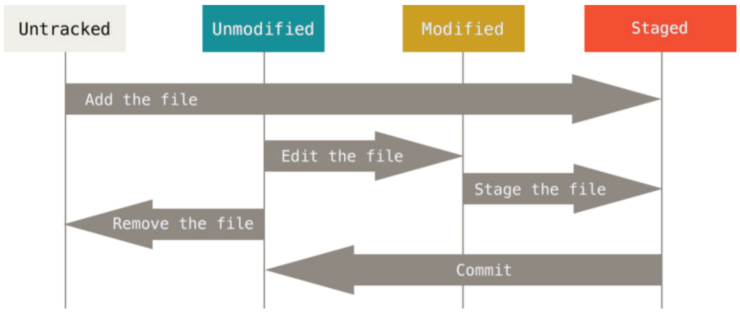
Everything in Git is check-summed before it is stored and is then referred to by that checksum. Git stores everything in its database not by file name but by the hash value of its contents.

1.1.5 Git adds data

When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way.

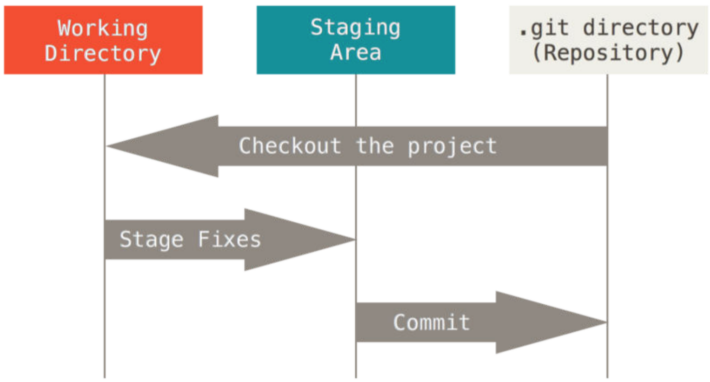
1.1.6 The three states of files in Git

1. **Committed**. Data is safely stored in your local database. If a file is in git directory, it is considered committed.
2. **Modified**. File changed, but not committed. If a file has changed but not staged, it is considered to be modified.
3. **Staged**. Marked a modified file in its current version to go into next commit snapshot. If a file is modified and added to staging area, it is considered to staged.



If you modify a file, it does not automatically go into next commit. It has to be staged for commit first. This ensures that the files are not automatically added to commits. Use git add command to push a file into the staging area.

This leads to three areas of git:



1. **Git directory.** Meta data and object database. This is where git copies to when a repository is cloned.

2. **Working directory.** Working directory is the single checkout of one version/branch of the project. Checkout involves pulling out compressed files from database git directory and placed on the disk for modification.

3. **Staging area.** Staging area (also referred to as index) is an area in git directory that keeps track of all the files that have been changed/added and need to go in the next commit.

1.2 Git setup and basic configuration

git config tool allows configuration and set up of git on a given computer. This controls how git looks and operates. The git env variables that control git are stored in three locations:

1. Global. Apply to every user in the system and all repositories. Command: git config –system. Location: /etc/gitconfig file

2. User level. Apply to all repos for the user. Command: git config –global. Location: USERHOME/.gitconfig

3. Repo level. Applies to current git repo. Command: git config. Location: .git/config

Each level overrides the previous level. For example, variables at repo level will override global variables.
Important settings:

1. **User name.** git config –global user.name "Nauman Mir"

2. **Email.** git config –global user.email "nauman.mir@gmail.com". The user name and email are embedded into every git commit.

3. **Edit to use.** git config –global core.editor vim

4. **Check settings.** git config –list or git config $< propertyname >$

5. **Git help.** git help $< verb >$. For example: git help config

1.3 Initialising a repo

If you have an existing project or directory structure that needs to be added to git, the process is:

1. git init. This creates the skeleton git repository (.git with no files) with all the necessary git configuration.

2. git add $< filenames >$

3. git commit -m "commit comments"

If you want to clone an existing git repository, the steps are:

1. git clone $< repositoryURL >$

Files that should be ignored are in .gitignore file.

1.4 Git - Basic Commands

1. git status: Gives the status of the files in repo (files untracked, modified, staged etc.)

2. git status -s: A summary of the status

3. git diff: Gives detailed info on the file changes

4. git add: For adding files to repo, for staging files

5. git commit -m "commit message": For committing the files to local git directory

6. git commit -a: Skips the staging area for files that are modified and commits them

7. git commit -a -m "commit comments": Stage and commit in one go

8. git rm: Removes a file from git

9. git rm log/.log: Removes all files from log directory with log extension.

10. git rm -f: To remove a modified file. This is a safety feature in order to prevent the accidental loss of changes

11. git mv file-from file-to: Renames the files in git
12. git log -p -2: -p Shows the difference between commits. -2 limits the log output to two commits.
13. git log –stat: Abbreviated stats for each commit
14. git log –pretty=format:"format options"
15. git log –since=2.weeks: Limits the output of the log
16. git log -Stext-string: Shows only those commits that have added or removed text-string
17. git checkout – file: Removes the changes to the file (basically copies over the file from git directory)
18. git remote -v: Shows the remote repositories that have been cloned
19. To synchronise a working branch with a remote develop branch
    - git fetch origin develop
    - git rebase origin/develop
    - git rebase –continue
    - git push -f
20. git format-patch -1 HEAD: Create a patch from the given number commit(s) starting from HEAD (or any other commit)
21. git am ¡ file.patch
22. To merge from another fork
    - git pull https://git-url/repository.git branchname
    - git push

1.4.1 Undo things in git

1. git commit –amend: Updated the previous commit rather than creating a new commit
   (a) git commit -m "initial commit"
   (b) git add forgotten file
   (c) git commit –amend
2. git reset HEAD file-name: To unstage a file
3. git checkout file-name: To revert a modified file (changed but not modified)

1.4.2 Remote repositories

1. git remote -v: Shows the remote repository
2. git remote add remoteAlias remoteRepoUrl: Adds a remote repo with the given alias (the default alias for remote is origin)
3. git fetch remoteAlias: Brings the information about a remote branch to local machine (updates .git directory). This allows branches to be checked out from local .git directory.
4. git pull: Performs a git fetch and merge. This brings data from remote server to local .git directory and then merges with the current working directory.
5. git push remoteAlias remoteBranch: Pushes the changes from local .git to remote server.
6. git remote show remoteAlias: Shows the remote repo plus branch tracking information.

1.5 Git tags

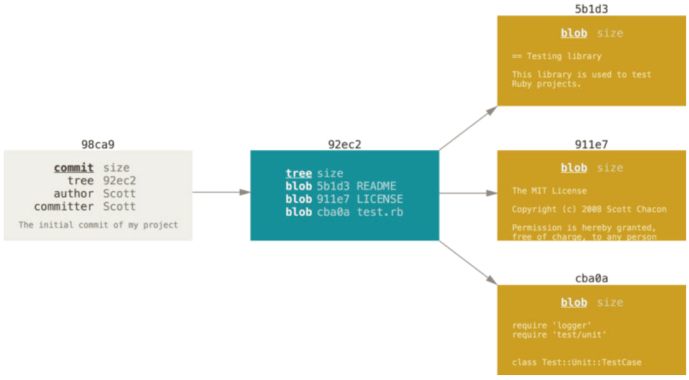
1.6 Branching

To understand Git branching, it is important to understand how git stores and manages files.

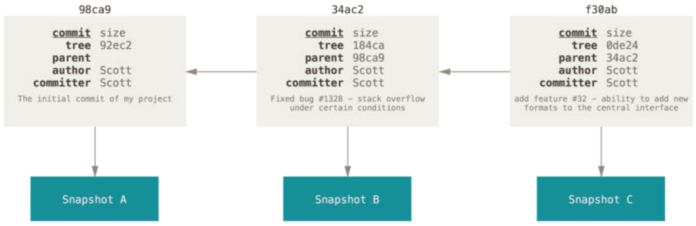
A project is a directory tree on the file system. Every commit in Git creates a snapshot of the directory tree and the whole file system can be re-created with the information in the snapshot. Files are stored in a snapshot as blobs that are referred to by their checksum (SHA-1) for integrity (For efficiency a file that has not changed in a commit, is not physically part of the snapshot - the snapshot just stores a reference to the file - This however, is an implementation detail and from the perspective of the users, each commit in git contains the complete snapshot of the system). Besides the snapshot, the commit also contains a pointer to a commit before it (or in case of merging of multiple branches, a set of pointers).

These commits form a linked list and a branch in git is simply a pointer to a given commit. Creating a branch in git is as simple as creating a pointer object. The pointer in a branch that points to the last commit for that branch is called HEAD.

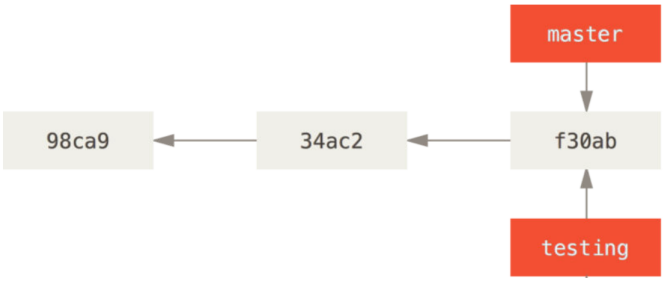
A single commit:



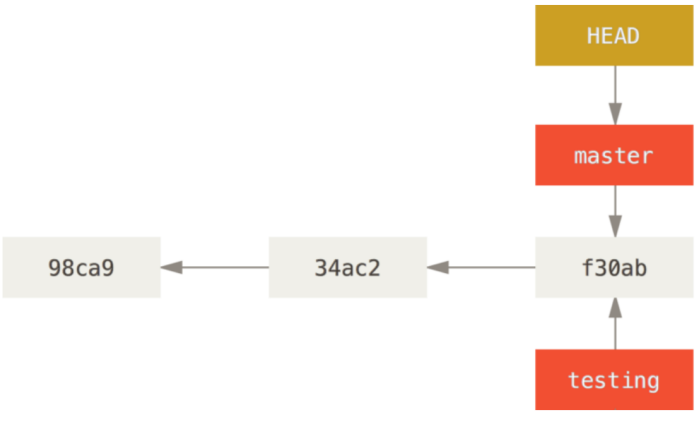
Chain of commits:



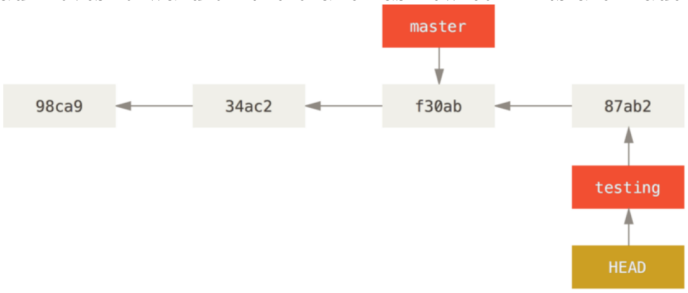
When a new branch is created, it is just another pointer to an existing commit.



The pointer in local git repo that points to the head of the currently checked-out branch is called HEAD.



Head moves forward on the branch as new commits are made:



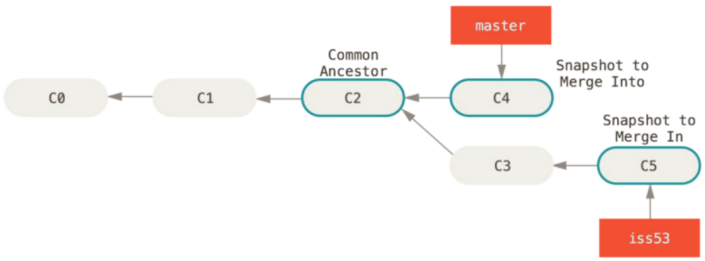
Basic commands for git branching and merging:

1. git branch: Gives the information on the branches
2. git branch -v: Gives the information on the branches along with info on last commit
3. git checkout -b branch-name. Create a new branch. This is same as git branch branch-name and git checkout branch-name.
4. To merge branch X into master, checkout master first (git checkout master) and then use merge command. git merge X
5. git branch -d branch-name: Deletes the branch

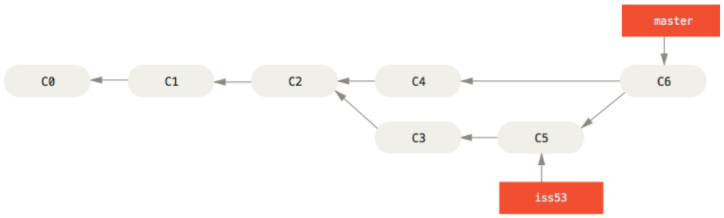
Some concepts regarding branching/merging

Fast-Forwarding. Merging two branches is simply a case of applying commits from one branch to another branch. In case the branch being merged into is a direct ancestor the branch being merged from (i.e. the HEAD of the branch being merged into is the parent commit of the branch being merged from), then git simply moves the HEAD of the branch being merged into to the HEAD of the branch being merged from. This is called fast-forwarding and happens when a branch is created from master and there are no changes in the master itself.

Three-way merge. In case the master has diverged from the point a branch was created from it, a three-way merge needs to be performed while merging the branch back into the master. This involves HEAD pointers of master and branch (pointing to their last commits respectively) and the last commit of master just before the branching (common ancestor) as shown in the figure below:



During merge, git creates a new snapshot on master that contains the changs from the branch being merged from.



1.6.1 Remote Branches

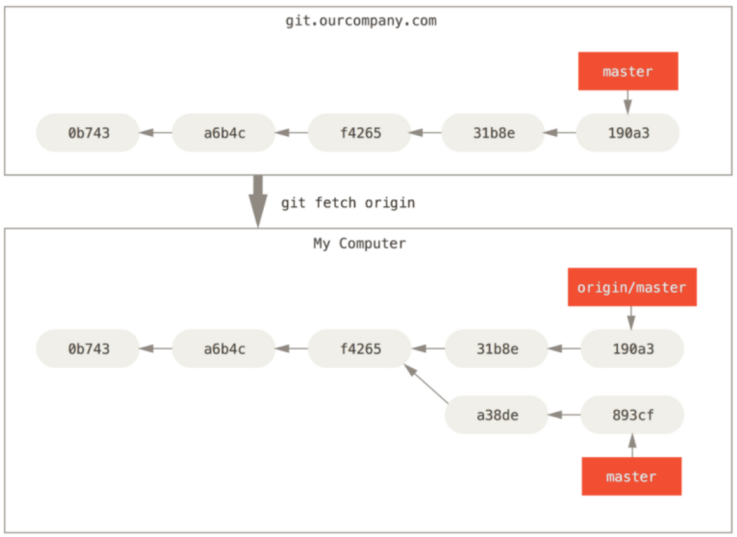
Remote references are references that exist in remote repositories. This includes branches, tags, and so on.

Remote tracking branches: Remote tracking branches are references to the state of remote branches. They are local references that you cannot move. They are moved for you whenever you do any network communication. They take the form (remote server alias)/(branch name). For example, origin/master. Origin is the default alias given to a server when you clone a git repository.

Important git commands:

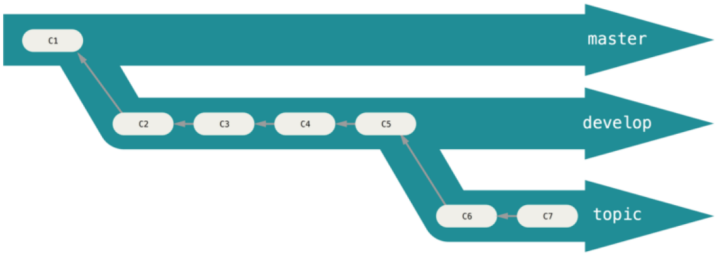
- git remote show: For finding about remote branches
- git ls-remote: For finding about all remote objects
- git fetch origin: Synchronises local database and moves the pointer to the latest state of the remote origin.
- git merge origin/branch-name: This command will merge origin/branch-name into local branch-name. This command needs to be run after the fetch command (which brings the changes from the server's branch into local remote branch).
- git pull: Does both fetch and merge
- git remote add: To add a new remote server with a given alias
- git push (remote server alias) (branch name): git push origin myfeature-branch
- git config –global credential.helper cache: Set up the local password cache so that you do not have to type in the password every time you talk to the server

git fetch command updates the origin/master pointer in the local database (note that work was done on the master branch on the local machine, but this is separate from the remote origin master)

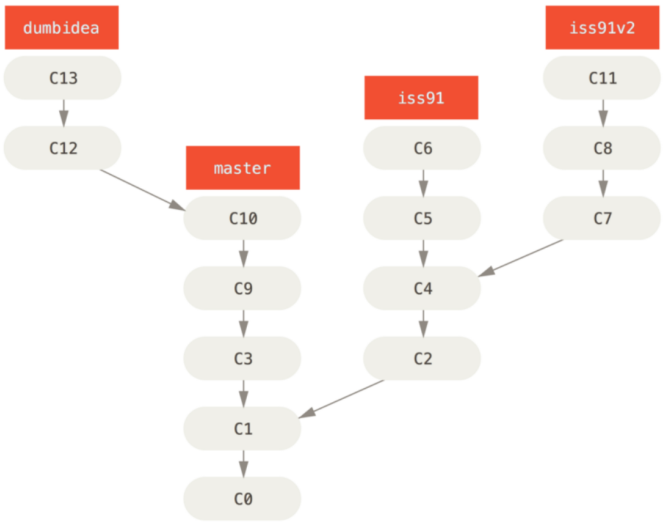


1.6.2 Branching workflows and tips

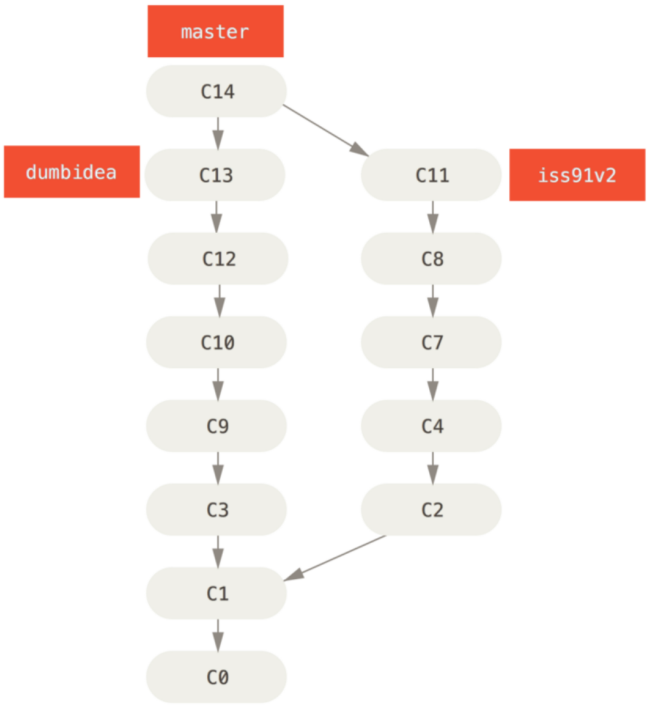
Typically git has got various branches for a project. A common usage scenario is to have three levels of branche. Master branch for very stable code that is released into production. A develop branch that is used to aggregate features while new functionality is being added. And finally, short-lived feature branches.



Multiple branches being created:

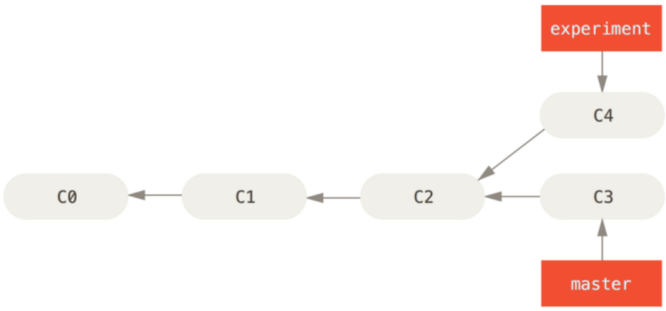


After merging of dumbidea and issue91v2 branches into master:

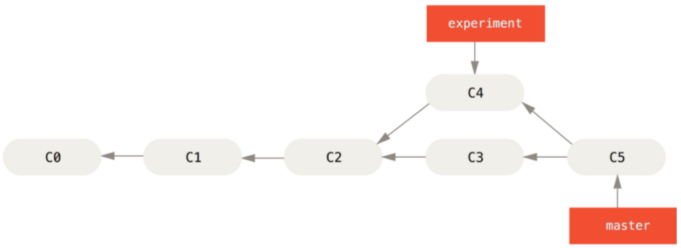


1.6.3 Merging and rebasing

Starting point. experiment was branched off from master but now both have diverged.



We want to propagate changes from branch to master. We use merging. Merging takes the common ancestor (C2) and latest commits of the two branches (C3 for master and C4 for branch), and performs a three way merge and creates a new commit in master.



If we want to update our branch from master (to ensure that our branch get the latest state of the master), we use rebasing. With rebase, you can take all the changes introduced in one branch, and apply them on another branch.

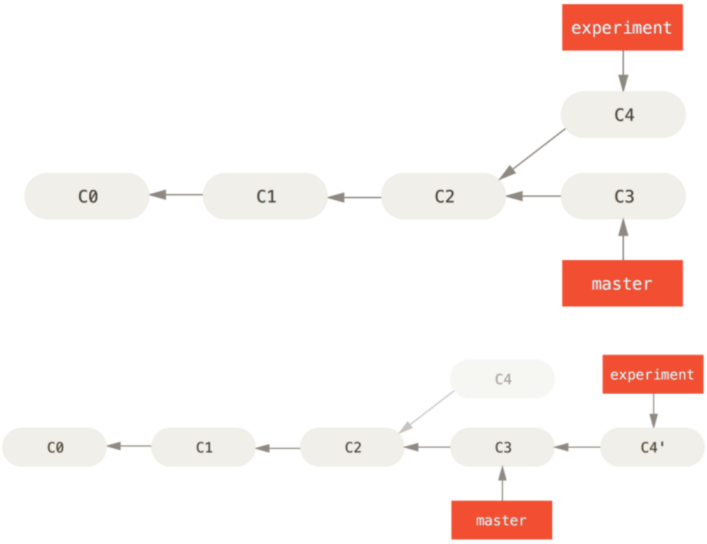
To rebase the branch with master, we use the rebase command. Rebase command does three things:

1. Scan of branch commits: It rewinds the HEAD of the branch back to the point the branch was created and makes a note of all the commits
2. Reset the HEAD of the branch to be same as the HEAD of the master
3. Apply the branch commits gathered in step 1 one by one

The patch of change introduced in C4 is reapplied on top of C3.

This process basically starts off by going to the latest state of the master and then applying the changes that were introduced in the branch. The end result is that the branch is up to date with master along with the changes that were introduced in the branch.

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```



Note that the snapshot pointed to by the final commit you end up with, whether its the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot its only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

1.7 Recovering deleted files

1. git log – ¡file-name¿: This will give the commits that have the file. Pick the commit that added the file
2. git checkout ¡commit¿ – ¡file-name¿: To recover the file

git check