How to create repository

<https://guides.github.com/activities/hello-world/>

Pull Request from UI :

Create a pull request, check two branches (base : on which it has to merge, actual : from which branch code has to merge to base)

If there conflicts : solve

If ready to merge : create pull request

From master branch, we can merge

Delete branch from UI:

Click branches and click delete

Blogs to learn more on github :

<https://github.com/blog>

Create new repo using command line

You cannot create a repo on github using git bash

go to https://github.com/ and create a repository “Study” on github, then run below lines.

### …or create a new repository on the command line

### Create a new empty repository from Ui and follow below steps

echo "# Study" >> README.md

git init

git config –global user.name “GirishVenkatesh1987”

git config –global user.email “girishprofile1987@gmail.com”

git add README.md

git commit -m "first commit"

git remote add origin https://github.com/GirishVenkatesh1987/Study.git

git push -u origin master

**…or push an existing repository from the command line**

git remote add origin https://github.com/GirishVenkatesh1987/Study.git

git push -u origin master

[**http://www.tutorialspoint.com/git/**](http://www.tutorialspoint.com/git/)

* Working Directory and Staging Area or Index

Only those files present in the staging area are considered for commit and not all the modified files.



* Blobs

Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file, and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed

* Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases

Tags are used for creating stable releases. To create a tag for using with the Git Drupal Repository, first, ensure that you're [following the tag naming convention](http://drupal.org/node/1015226) if you're using this tag for making a release. From inside the directory of the project, an example is:

git tag 7.x-1.0

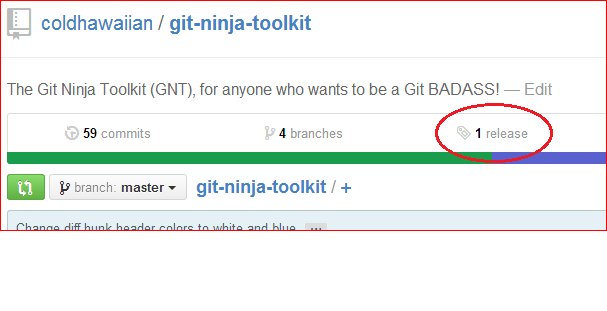
Once the tag is created, you need to push the tag up to the master repository. By itself, push doesn't send the tags up, you also need to tell it to include the tags in the push by appending the --tags flag:

git push origin --tags

If you don't want to push all your tags, you can also be specific:  
Example:

git push origin tag 7.x-1.0

To check and confirm remote tags, the command is  
git tag –l



Once tag is pushed to repo, it can be viewed under releases

How to checkout/clone tag branch :

$ git clone will give you the whole repository.

After the clone, you can list the tags with $ git tag -l and then checkout a specific tag:

$ git checkout tags/<tag\_name>

Even better, checkout and create a branch (otherwise you will be on a branch named after the revision number of tag):

$ git checkout tags/<tag\_name> -b <branch\_name> //create a branch for tag

* HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. The heads of the branches are stored in **.git/refs/heads/** directory.

[CentOS]$ ls -1 .git/refs/heads/

master

[CentOS]$ cat .git/refs/heads/master

570837e7d58fa4bccd86cb575d884502188b0c49

A branch to be a collection of commits, an array of commits. Then you have a pointer to this array, and a name, and more stuff, this results in a branch. Then you can navigate through this array (read through this branch), between different commits, by using the HEAD pointer, which points to the current commit, on the current branch (move it backwards with checkout and #SHA1, move one forwards with commit)

* To change remote url from

cloned with https url, change to ssh

git config remote.origin.url git@github.com:GirishVenkatesh1987/Documents.git

* To see the url details

git remote show origin

* To avoid username and pw popup for each push in windows for https remote url

git config --global credential.helper wincred

* [**https://www.youtube.com/watch?feature=player\_detailpage&v=ZDR433b0HJY#t=2791s**](https://www.youtube.com/watch?feature=player_detailpage&v=ZDR433b0HJY#t=2791s): nice video

git download official site : <https://git-scm.com/>

references : <http://gitref.org/>

books : <https://progit.org/>

Central Version control : one server where data base is present and we clone and push(dependent on one server) ex : clear case

Distributed Version Control : each has his its own database ex : git, so faster .uses local database for all versioning operations, offline is possible(no network)

Git push origin <file://..../> => to file system

Try this.

In git , remove is loosing pointer to it. Still data exist in database.

Each add, creates a snapshot. (Checksum). Stores in database with key value pair.

Note : rename files will not create new checksum(snapshot). Key same value changes.

During retrival, based on commit ids it renames and gives.

As long as content wont change, the checksum is same for copy, rename operations.

Git add . , git commit “” (two command) or git commit –a -m “” (one command)

**Merge** : git merges easily to all in one branch. Master branch line overwritten to merge branch if there change in same line

**Ex** : git merge blue, git merge email => so blue , email branches changes merged to master branch

Good Practice : Keep repo upto date with master everyday

To view .git => tree –a

./.git/objects => gives checksum of each commits.

Try how to unzip this checksums

Different kinds of git clone (like ssh, https….etc and there singnificance)

Git add => create a snapshot,

Git commit => put that snapshot to local database

Git add –p => to create a snapshot for portion of file. It hels to give one portion of file for review

Check examples

Git checkout –b test master <=>git checkout master, git checkout –b test

Based on checksum, we can see file edited or not by comparing two commits.

If checksum is same, particular file is nt changed as refer to previous commits

Merge : 1. **Fast Forward Merge (merge branch is reachable to master)**

2. **Non Fast Forward Merge (merge branch is not reachable to master)**

Steps to follow when merge conflicts comes

1. Git status
2. Keep which contents required in conflict file
3. Git add –all
4. Git commit –am “solving merge conflicts”

Git branch –D branchname : to delete a branch

Git push origin master : push the current branch data to master of origin URL ⬄ git push remote\_url(ssh/https) master

Git push failures solve :

1. Git fetch

Some one else have pushed to master, its not upto date with local. So get those coomits to ur local and push ur other commits

1. Git push origin master

**Fetch, push is better than pull, push**

**Pull = fetch + Merge**

**Merge v/s Rebase**

**Fetch , Merge v/s Pull**

**Git config –global =>** all configuration details

**HEAD v/s heads**

HEAD is a symbolic reference for current branch pointing to last commit

[**http://www.tutorialspoint.com/git/**](http://www.tutorialspoint.com/git/)

* + Git was initially designed and developed by Linus Torvalds for Linux kernel development.
  + The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages
  + Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git
  + Revision represents the version of the source code
  + Git URL is stored in config file.
  + cat .git/config
    - [core]
    - repositoryformatversion = 0
    - filemode = true
    - bare = false
    - logallrefupdates = true
    - [remote "origin"]
    - url = gituser@git.server.com:project.git
    - fetch = +refs/heads/\*:refs/remotes/origin/\*
  + Installation of Git Client in linux

If you are using Debian base GNU/Linux distribution, then **apt-get** command will do the needful.

[ubuntu ~]$ sudo apt-get install git-core

[sudo] password for ubuntu:

[ubuntu ~]$ git --version

git version 1.8.1.2

And if you are using RPM based GNU/Linux distribution, then use **yum**command as given.

[CentOS ~]$

su -

Password:

[CentOS ~]# yum -y install git-core

[CentOS ~]# git --version

git version 1.7.1

* + Git stores all global configurations in **.gitconfig** file
  + Avoid merge commits for pulling

You pull the latest changes from a remote repository, and if these changes are divergent, then by default Git creates merge commits. We can avoid this via following settings.

git config --global branch.autosetuprebase always

* + Color highlighting

The following commands enable color highlighting for Git in the console.

git config --global color.ui true

git config --global color.status auto

git config --global color.branch auto

Setting default editor

By default, Git uses the system default editor, which is taken from the VISUAL or EDITOR environment variable. We can configure a different one by using git config.

git config --global core.editor vim

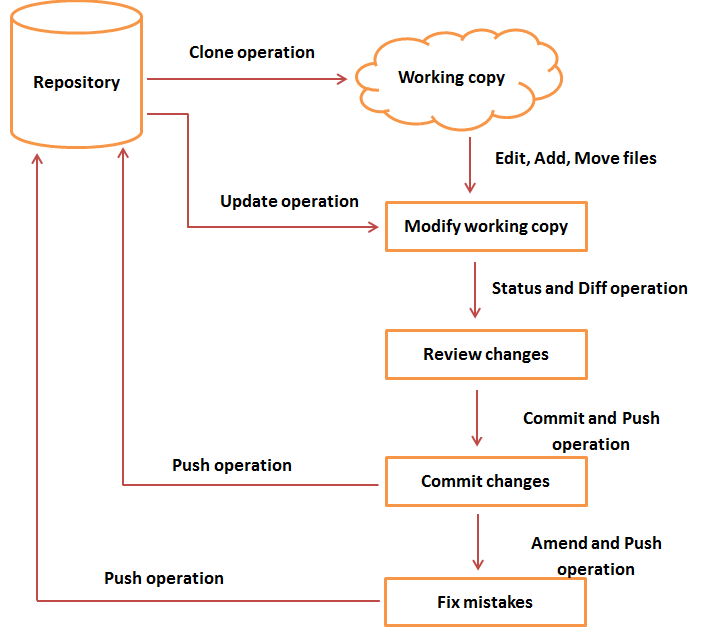
Setting default merge tool

Git does not provide a default merge tool for integrating conflicting changes into your working tree. We can set default merge tool by enabling following settings.

git config --global merge.tool vimdiff

Listing Git settings

git config --list



## Create New User

# add new group

[root@CentOS ~]# groupadd dev

# add new user

[root@CentOS ~]# useradd -G devs -d /home/gituser -m -s /bin/bash gituser

# change password

[root@CentOS ~]# passwd gituser

## Create a Bare Repository [try this ]

Let us initialize a new repository by using **init** command followed by **--bare**option. It initializes the repository without a working directory. By convention, the bare repository must be named as **.git**.

[gituser@CentOS ~]$ pwd

/home/gituser

[gituser@CentOS ~]$ mkdir project.git

[gituser@CentOS ~]$ cd project.git/

[gituser@CentOS project.git]$ ls

[gituser@CentOS project.git]$ git --bare init

Initialized empty Git repository in /home/gituser-m/project.git/

[gituser@CentOS project.git]$ ls

branches config description HEAD hooks info objects refs

git remote add origin gituser@git.server.com:project.git

to clone

git clone gituser@git.server.com:project.git

## Generate Public/Private RSA Key Pair and add key

tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ ssh-keygen

To add private key [it will prompt for password]

ssh-copy-id -i ~/.ssh/id\_rsa.pub gituser@git.server.com

To add public key [it will prompt for password]

ssh-copy-id -i ~/.ssh/id\_rs gituser@git.server.com

**git show** command to view the commit details. The git show command takes **SHA-1** commit ID as a parameter.

[jerry@CentOS project]$ git show cbe1249b140dad24b2c35b15cc7e26a6f02d2277

git diff : shows the difference of each file as compare to current and master branch

To correct his last commit. In this case, **git amend** operation will help. The amend operation changes the last commit including your commit message; it creates a new commit ID. Git log will not show the last commit which will be replaced by git amend commit.

git commit --amend -m 'Changed return type of my\_strlen to size\_t

to review his changes, so he uses the **git show** command to review his changes.

git show d1e19d316224cddc437e3ed34ec3c931ad803958

# Git - Stash Operation

In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time

git stash

your working directory is clean and all the changes are saved on a stack

view a list of stashed changes by using the **git stash list** command.

git stash list

stash@{0}: WIP on master: e86f062 Added my\_strcpy function

**git stash pop** command, to remove the changes from the stack and place them in the current working directory

git stash pop

# Git - Move Operation

git mv string.c src/

# Git - Rename Operation

git mv string.c string\_operations.c

# Git - Add Operation

git add Makefile

# Git - Delete Operation

git rm string\_operations

# Git - Fix Mistakes

Wants the file from origin to my local branch.

Git checkout filename : retrieves file from origin and replaces in current branch.

git checkout string\_operations.c

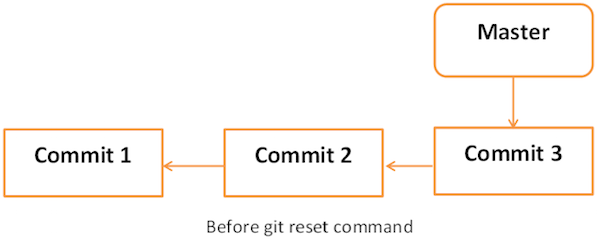
Wants the file back from staging area.

git checkout HEAD -- filename : changes made, but not added to staging area.

In Git, there is one HEAD pointer that always points to the latest commit

git checkout HEAD -- string\_operations.c

The Git reset command is used to reset or revert changes



Soft (just moves the HEAD pointer one below commit)

use Git reset command with --soft option followed by commit ID, then it will reset the HEAD pointer only without destroying anything

git log -1 => shows one latest commit from where HEAD points to

git log -2 => shows last two commits from where HEAD points to

git log –n => shows last n commits from where HEAD points to

git reset --soft HEAD~

Now HEAD will point to second last commit.

Mixed (Staging area code will revert , local wont)

Git reset with --mixed option reverts those changes from the staging area that have not been committed yet. It reverts the changes from the staging area only. The actual changes made to the working copy of the file are unaffected

The default Git reset is equivalent to the git reset -- mixed.

Hard (Both staging area and local changes will revert to original remote repo changes)

If you use --hard option with the Git reset command, it will clear the staging area; it will reset the HEAD pointer to the latest commit of the specific commit ID and delete the local file changes too.

git reset --hard 577647211ed44fe2ae479427a0668a4f12ed71a1

Git reset command succeeded, which will revert the file from the staging area as well as remove any local changes made to the file

Merge (to get back the filel how it was before raising merge, some times merge produces conflicts. So to get back the files without conflict information in that, we use git reset –mixed)

When you modify a file in your repository, the change is initially unstaged. In order to commit it, you must stage it—that is, add it to the index—using git add.

When you make a commit, the changes that are committed are those that have been added to the index.

git reset changes, at minimum, where the current branch (HEAD) is pointing. The difference between --mixed and --soft is whether or not your index is also modified. So, if we're on branch master with this series of commits:

- A - B - C (master)

HEADpoints to C and the index matches C.

When we run git reset --soft B, master (and thus HEAD) now points to B, but the index still has the changes from C; git status will show them as staged.

So if we run git commit at this point, we'll get a new commit with the same changes as C.

If we run git add and git commit => New commit D comes, C will be lost

Okay, so starting from here again:

- A - B - C (master)

Now let's do git reset --mixed B. (Note: --mixed is the default option). Once again, master and HEAD point to B, but this time the index is also modified to match B. If we run git commit at this point, nothing will happen since the index matches HEAD. We still have the changes in the working directory, but since they're not in the index, git status shows them as unstaged. To commit them, you would git add and then commit as usual.

And finally, --hard is the same as --mixed (it changes your HEAD and index), except that --hard also modifies your working directory. If we're at C and run git reset --hard B, then the changes added in C, as well as any uncommitted changes you have, will be removed, and the files in your working copy will match commit B

**Tags (release code, no one can edit. Product version release)**

**Particular commit to make as tag**

git tag -a 'Release\_1\_0' -m 'Tagged basic string operation code' HEAD

git push origin tag Release\_1\_0

**To view tags**

git tag -l

## Delete Tags

git tag -d Release\_1\_0

## Patch

Create a patch ( a \*.patch text file where only recent commit files changes will be present). Some one else can apply this patch only to get those recent commit file changes)

Create a patch

[jerry@CentOS src]$ git status -s

M string\_operations.c

?? string\_operations

[jerry@CentOS src]$ git add string\_operations.c

[jerry@CentOS src]$ git commit -m "Added my\_strcat function"

[master b4c7f09] Added my\_strcat function

1 files changed, 13 insertions(+), 0 deletions(-)

[jerry@CentOS src]$ git format-patch -1

0001-Added-my\_strcat-function.patch

Apply a patch

[tom@CentOS src]$ git apply 0001-Added-my\_strcat-function.patch

[tom@CentOS src]$ git status -s

M string\_operations.c

?? 0001-Added-my\_strcat-function.patch

## Create Branch

If any specific commit ID is not provided, then the branch will be created with HEAD as its starting point

git checkout -b test\_branch

## Delete a Branch

git branch -D test\_branch

## Rename a Branch

**–m** option followed by the **old branch name** and the **new branch name**.

git branch -m old\_branch new\_name\_branch

## Difference between git merge(ur branch changes sits top on merge branch) and git rebase ( remote branch commits comes below the current branch changes) -> try examples

The Git merge command tries to put the commits from other branches on top of the HEAD of the current local branch. For example, your local branch has commits A−>B−>C−>D and the merge branch has commits A−>B−>X−>Y, then git merge will convert the current local branch to something like A−>B−>C−>D−>X−>Y

The Git rebase command tries to find out the common ancestor between the current local branch and the merge branch. It then pushes the commits to the local branch by modifying the order of commits in the current local branch. For example, if your local branch has commits A−>B−>C−>D and the merge branch has commits A−>B−>X−>Y, then Git rebase will convert the current local branch to something like A−>B−>X−>Y−>C−>D.

When rebase helpful :

When multiple developers work on a single remote repository, you cannot modify the order of the commits in the remote repository. In this situation, you can use rebase operation to put your local commits on top of the remote repository commits and you can push these changes.

## To avoid new line characters treat differently for windows and linux

GNU/Linux and Mac OS uses **line-feed (LF)**, or new line as line ending character, while Windows uses **line-feed and carriage-return (LFCR)**combination to represent the line-ending character.

To avoid unnecessary commits because of these line-ending differences, we have to configure the Git client to write the same line ending to the Git repository.

For Windows system, we can configure the Git client to convert line endings to **CRLF** format while checking out, and convert them back to **LF** format during the commit operation. The following settings will do the needful.

git config --global core.autocrlf true

For GNU/Linux or Mac OS, we can configure the Git client to convert line endings from **CRLF** to **LF** while performing the checkout operation.

git config --global core.autocrlf input

tags are immutable. It means, tag is a branch, which nobody intends to modify. tags are immutable. It means, tag is a branch, which nobody intends to modify

git config --global user.name "Jerry Mouse"

git config --global user.email "jerry@tutorialspoint.com"

Avoid merge commits for pulling

git config --global branch.autosetuprebase always

Color highlighting

git config --global color.ui true

git config --global color.status auto

git config --global color.branch auto

Setting default editor

git config --global core.editor vim

Setting default merge tool

git config --global merge.tool vimdiff

# Git - Different Platforms-To avoid line feed

git config --global core.autocrlf true

## Create New User

# add new group

[root@CentOS ~]# groupadd dev

# add new user

[root@CentOS ~]# useradd -G devs -d /home/gituser -m -s /bin/bash gituser

# change password

[root@CentOS ~]# passwd gituser

## Create a Bare Repository

[gituser@CentOS ~]$ pwd

/home/gituser

[gituser@CentOS ~]$ mkdir project.git

[gituser@CentOS ~]$ cd project.git/

[gituser@CentOS project.git]$ ls

[gituser@CentOS project.git]$ git --bare init

Initialized empty Git repository in /home/gituser-m/project.git/

[gituser@CentOS project.git]$ ls

branches config description HEAD hooks info objects refs

## Adding Keys to authorized\_keys

ssh-copy-id -i ~/.ssh/id\_rsa.pub gituser@git.server.com

# Git - Stash Operation

In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.

[jerry@CentOS project]$ git status -s

M string.c

?? string

[jerry@CentOS project]$ git stash

Saved working directory and index state WIP on master: e86f062 Added my\_strcpy function

HEAD is now at e86f062 Added my\_strcpy function

view a list of stashed changes by using the **git stash list** command.

[jerry@CentOS project]$ git stash list

**git stash pop**command, to remove the changes from the stack and place them in the current working directory.

[jerry@CentOS project]$ git status -s

?? string

[jerry@CentOS project]$ git stash pop

To get file from staging area to working copy (added, made local changes. But don’t want those changes. Previously git add file to move back to working copy)

git checkout HEAD -- string\_operations.c

On which all commands merge conflicts comes ? How to get back the original file before executing command ?

How to resolve the merge conflicts ?

* Use git diff to view what all files have difference, then change those file – commit and push

## Git conflicts comes in git merge or git pull

### How do I know which files have conflicts in them?

If your merge failed to even start, there will be no conflicts in files. If git finds conflicts during the merge, it will list all files that have conflicts after the error message. You can also check on which files have merge conflicts by doing a 'git status'.

Example:

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: <Some file>

#

# Changed but not updated:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# unmerged: <file>

#

"Changes to be committed": All committed changes to files that are not affected by the conflict are staged.

"Changed but not updated ... unmerged": All files that have conflicts that must be resolved before repository will be back to working order.

### How do I find conflicts within the file itself?

Conflicts are marked in a file with clear line breaks:

<<<<<<< HEAD:mergetest

This is my third line //local branch code

=======

This is a fourth line I am adding //remote url origin code

>>>>>>> 4e2b407f501b68f8588aa645acafffa0224b9b78:mergetest

**<<<<<<<**: Indicates the start of the lines that had a merge conflict. The first set of lines are the lines from the file that you were trying to merge the changes **into**.

**=======**: Indicates the break point used for comparison. Breaks up changes that user has committed (above) to changes coming from merge (below) to visually see the differences.

**>>>>>>>**: Indicates the end of the lines that had a merge conflict.

### How do I resolve a merge conflict in a file?

You resolve a conflict by editing the file to manually merge the parts of the file that git had trouble merging. This may mean discarding either your changes or someone else's or doing a mix of the two. You will also need to delete the '<<<<<<<', '=======', and '>>>>>>>' in the file.

### What do I do after I've resolved conflicts in all affected files?

git add the file(s), git commit and git push (Push only for branches tracked.)

(**Note** added by Chin - need to commit everything, not just the resolved conflict file.)

## Tools to help you resolve both types of merge conflicts

The following git tools below can help you resolve both simple and more complicated git merges.

### General tools

#### git diff

[**git diff**](http://genomewiki.ucsc.edu/index.php/Working_with_Git#.27git_diff.27): a command that helps find differences between states of a repository/files. Useful in predicting and preventing merge conflicts.

**git diff origin/master <fileName>**: Find the differences between the current index (HEAD) of fileName and what is in the central repository (origin/msater)

diff --git a/mergetest b/mergetest

index 9be56b9..0aeffac 100644

--- a/mergetest

+++ b/mergetest

@@ -1,3 +1,4 @@

hello

+I am also editing this line

This is a test

-This is my third line

+This is a fourth line I am adding

Changes coming from origin/master are marked with **+**, while changes that are in your local repository (HEAD) are marked with **-**. This syntax does not notify which lines are added are deleted but just which lines originate in which state of the file.

Good example to solve conflict

==============================

Here's a probable use-case, from the top:

You're going to pull some changes, but oops, you're not up to date:

git fetch origin

git pull origin master

From ssh://gitosis@example.com:22/projectname

\* branch master -> FETCH\_HEAD

Updating a030c3a..ee25213

error: Entry 'filename.c' not uptodate. Cannot merge.

So you get up-to-date and try again, but have a conflict:

git add filename.c

git commit -m "made some wild and crazy changes"

git pull origin master

From ssh://gitosis@example.com:22/projectname

\* branch master -> FETCH\_HEAD

Auto-merging filename.c

CONFLICT (content): Merge conflict in filename.c

Automatic merge failed; fix conflicts and then commit the result.

So you decide to take a look at the changes:

git mergetool

Oh me, oh my, upstream changed some things, but just to use my changes...no...their changes...

git checkout --ours filename.c

git checkout --theirs filename.c

Careful! The meaning of --ours and --theirs is reversed. --ours == the remote. --theirs == local. See git merge --help

git add filename.c

git commit -m "using theirs"

And then we try a final time

git pull origin master

From ssh://gitosis@example.com:22/projectname

\* branch master -> FETCH\_HEAD

Already up-to-date.

ours an theirs meaning

===========================

* Git merge [current branch ---🡪 remote branch ] [ours -🡪theirs ]

(takes the code from local branch and puts on top of remote branch)

* + git branch

\*current\_branch

remolte\_branch

* + git merge –ours //avoid using it, manually edit file and fix conflicts.

They usually want to salvage as much automerge result as possible in

a conflicted file, and want to take the local branch (current\_branch) change only in the

conflicted part

* + git merge –theirs //avoid using it, manually edit file and fix conflicts.

They usually want to salvage as much automerge result as possible in

a conflicted file, and want to take the remote branch (remlote\_branch) change only in the

conflicted part

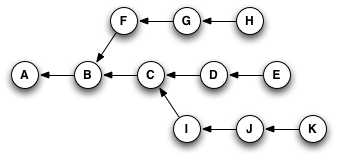
in case of rebase [remote branch ---🡪 local branch ] [ours -🡪theirs ]

(take code from remote and puts on top of local branch) : --theirs : points to current\_branch

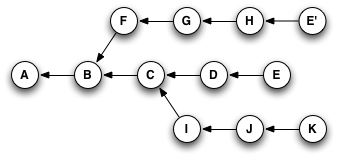
--ours : points to remote\_branch

Cherry pick of commits examples ?

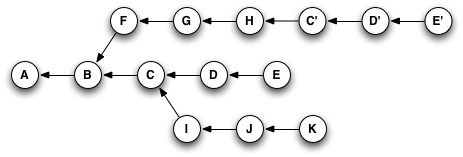
Cherry pick -> picks the other commit id changes and creates a new commit in current branch.



If you were at node H in this graph, and you typed **git cherry-pick E** *(yes, you'd actually type part or all of the SHA for the commit, but for simplicity's sake, I'll just use the labels that are already here)*, you'd wind up with a copy of commit E—let's call it "E prime" or **E'**—that pointed to **H** as its parent, like so:



Or, if you typed something like **git cherry-pick C D E**, you'd wind up with this when you were done:



## EXAMPLES

**git cherry-pick master**

Apply the change introduced by the commit at the tip of the master branch and create a new commit with this change.

**git cherry-pick ..master**

**git cherry-pick ^HEAD master**

Apply the changes introduced by all commits that are ancestors of master but not of HEAD to produce new commits.

**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.

How to cherry pick remote branch commit into current branch ?

git checkout remote\_branch  
- git pull origin remote\_branch  
- git checkout current\_branch  
- git pull origin current\_branch  
- git cherry-pick remote\_branch\_commit\_id

**How to configure git mergetool ?**

git config --global merge.tool vimdiff

or install any other tool. Change vimdiff to that tool

git config --global merge.tool kdiff3

git config --global mergetool.kdiff3.cmd '"C:\\Program Files (x86)\\KDiff3\\kdiff3" $BASE $LOCAL $REMOTE -o $MERGED'

Now you will be able to resolve your conflicts the command line via

git merge

if merge conflicts comes, execute below command to open vimdiff for conflicted files

git mergetool

Whats the size of local repsotory ? is it same as centralized repository?

To reduce git repo size :

git gc

git gc --aggressive

git prune

That seemed to have did the trick. I started with around 10.5MB and now its little more than 980KBs.

By default removing files from git doesn't actually remove them from git, it just commits that they aren't there anymore. If you want to actually remove the historical references (i.e. you have a committed a password), you need to do this:

git filter-branch --prune-empty --tree-filter 'rm -rf <name-of-folder>' HEAD

After that you can check that your file or folder no longer shows up in the git history at all

git log -- <name-of-folder> # should show nothing