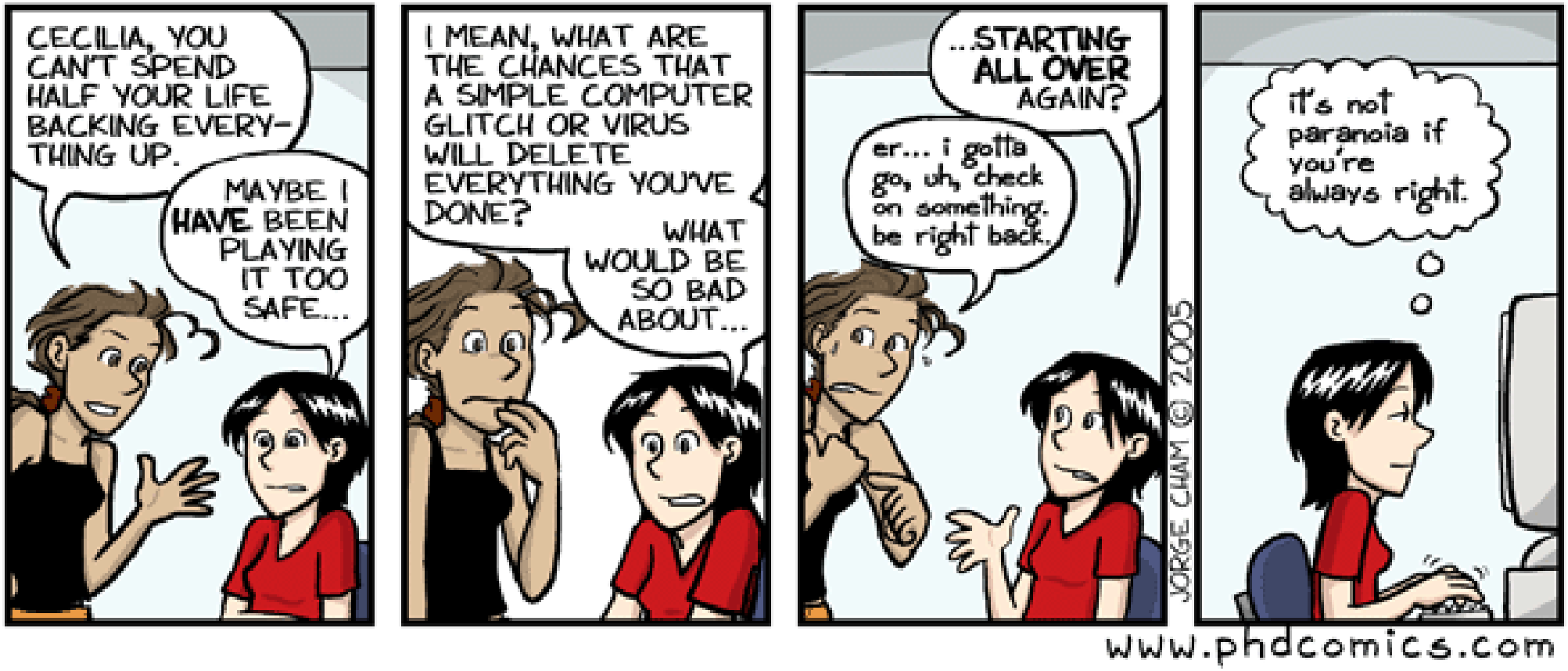
Git and Github

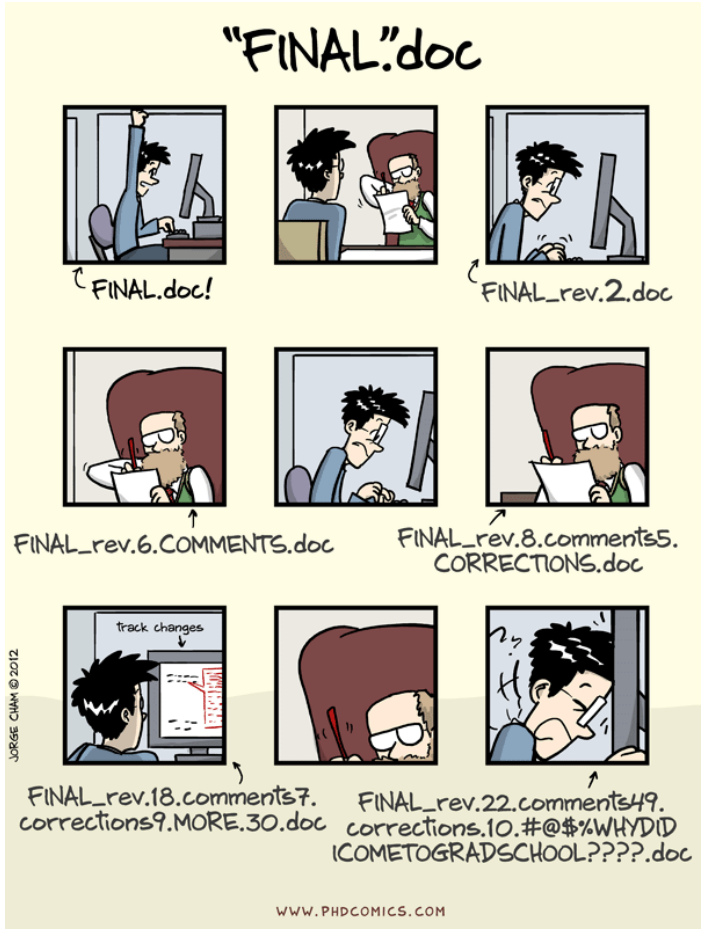
Versioning for beginners with some deep shit

RAVI GARG

Not backing up?



Manual copies



What and Why Version Control System?

A system that keeps track of your changes and document it

Allows to know who made the changes, how and when

Allows to revert any changes and go back to a safe/running state

Many types

- Local version control system
- Central version control system
- Distributed version control system

Local, Central and Distributed VCS

Local VCS

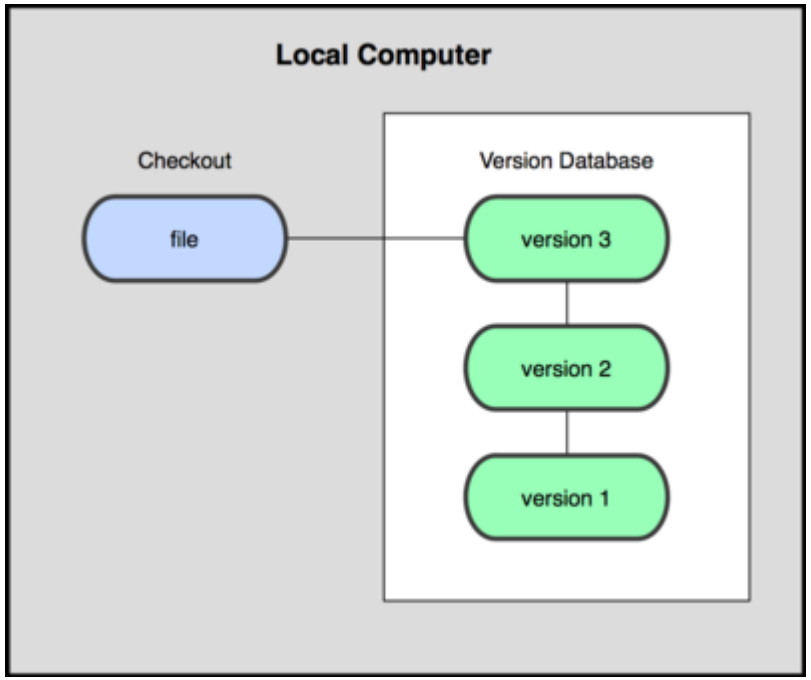
- Simple database that keeps all the changes to files under revision control system
- Example RCS (still on Mac). Keeps differences between files/patch in a special format on disk

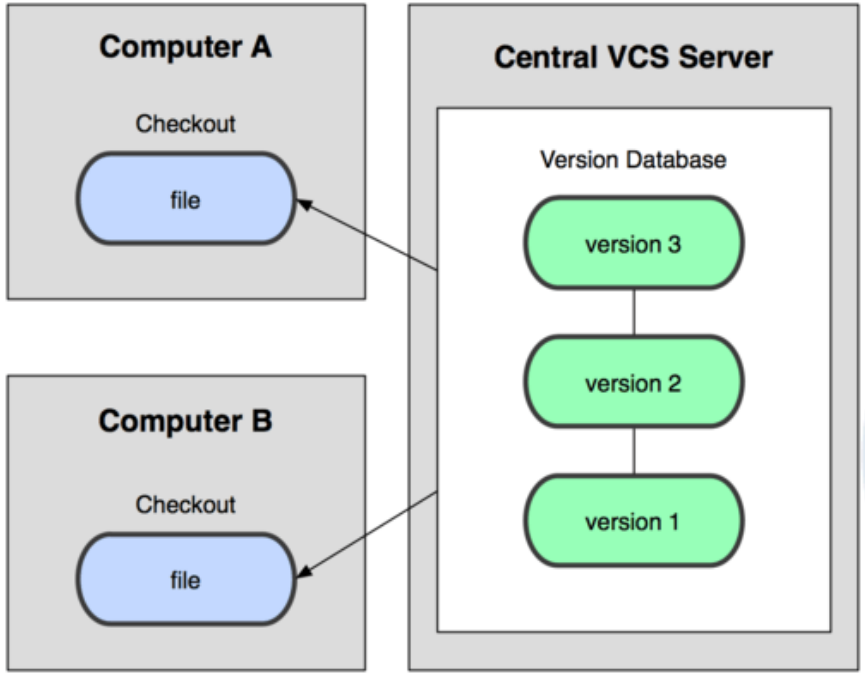
Central VCS

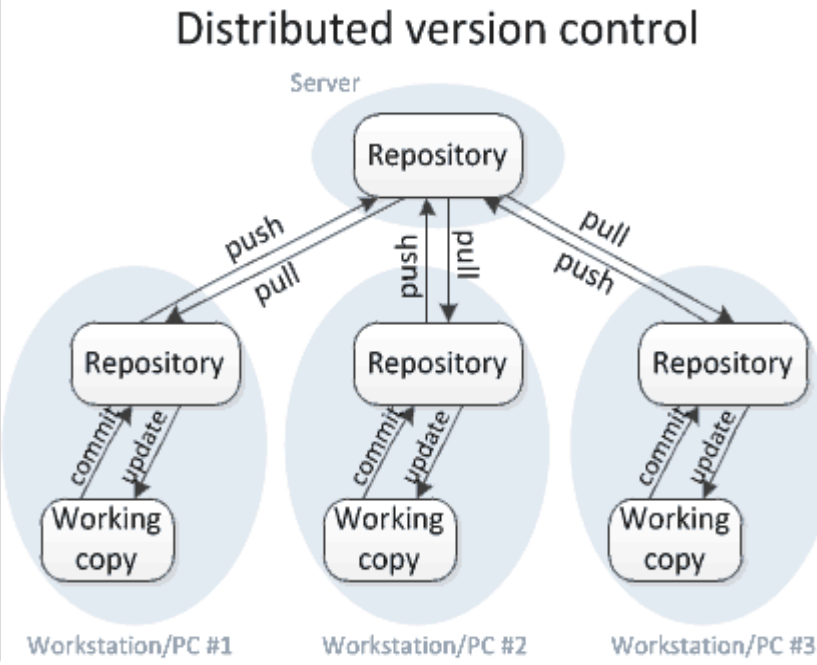
- To collaborate with other developers, Central VCS was developed.
- Single server that contain all the version files, clients can check out files from central place.
- Advantages : Networking, Team work, Centralized management and administrative control.
- Disadvantages : Single point of failure (Server down, disk corrupted)
- Example : Subversion (svn), Perforce etc.

Distributed VCS

Local, Central and Distributed VSC







What is Git?

GIT

- Distributed version control system
- Started in 2015 by Linus Torvalds to aid in Linux Kernel development

Advantages

- High performance : Committing new changes, branching, merging and comparing past versions are all optimized for performance.
- Highly secure : The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1.
- Highly flexible : Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.

Github

Social platform or repository hosting service for Git to share knowledge and work

Allows for code collaboration with anyone online

Add extra functionalities : UI, Documentation, Bug tracking, Feature requests, pull requests.

Installation

PLATFORM SPECIFIC

Preliminaries

Version

- git --version

Setting config values

- git config --global user.name “Ravi Garg”
- git config --global user.email ravigarg27@gmail.com
- git config --list

Need Help

- git help <verb> example : git help config
- git <verb> --help example git config --help

Getting started – 2 scenarios

1st Scenario

- Local project you want to track changes. You are the only one working on it

2nd Scenario

- Global project with many contributors. For example you want to contribute to an open source project.

Common Workflow – Git stages

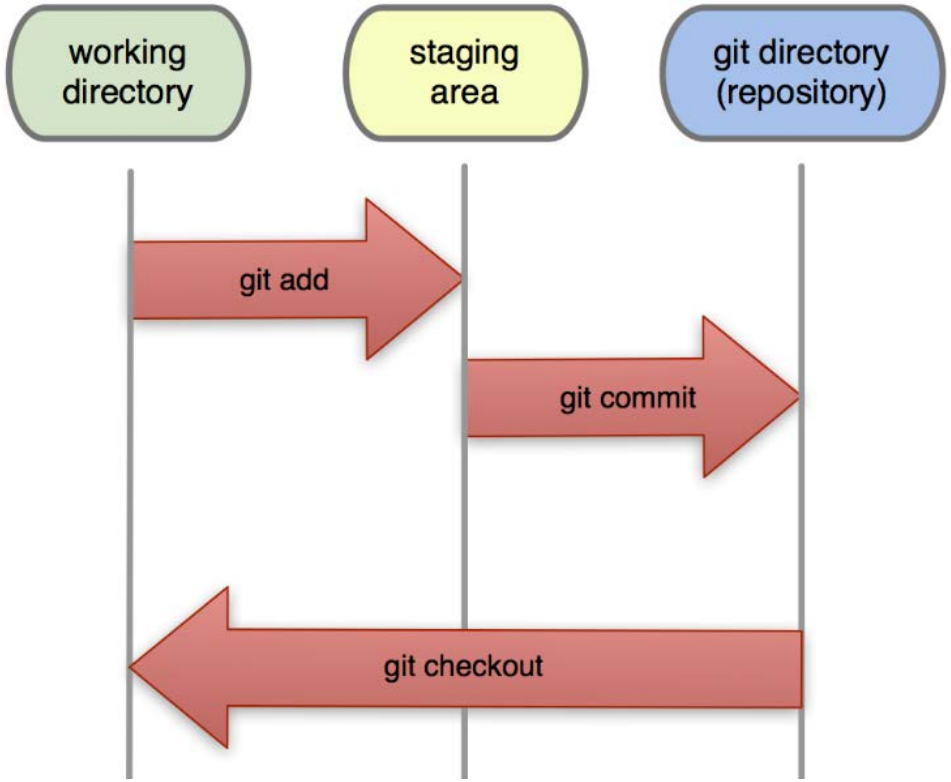
Three stages files can reside in

- Modified
- Staged
- Committed

Modified means you have edited that file but not added or committed that file to repo

Staged means you have tagged the edited file in its current version for committing to repo

Committed means the file is safely stored in local repo



1st Scenario – Local project

Initialize a repo from existing code

- git init

Checking the status of the repo

- git status : *check which files have been modified and/or staged since the last commit*

Adding files to staging area

- git add <filename> (or git add -A) : *if untracked, start tracking a file or directory; if tracked and modified, stage it for committing*

Detour .gitignore :

Removing files from staging area

- git reset <filename> : *unstage a changed file*
- git status

1st Scenario - continued

Commit

- git commit –m “Relevant commit message”
- git status

Log (to get the history of commits)

- git log

Pushing to Github

- git remote add origin <url of the github repo>
- git push origin master

Get rid of git / Fed up of tracking changes

- remove the .git directory (rm –rf .git)

2nd Scenario Common workflow

Step 0 : Fork the Git repository

Step 1 : Clone the Git repository as a working copy.

Step 2 : Modify the working copy by adding/editing files / BRANCHING .

Step 3 : If necessary, also update the working copy by taking other developer's changes.

Step 4 : Review the changes before commit / Resolve merge conflicts.

Step 5 : Commit changes. If everything is fine, then push the changes to the repository.

Step 6 : After committing, if something is wrong, then correct the last commit and push the changes to the repository.

2nd Scenario – Step 0,1 – Forking/Cloning

Create or Fork a remote repo

- On Github

Cloning a remote repo

- cd <directory where you want to clone>
- git clone <url of the remote directory> : copies your remote repo to your local machine (in a subdirectory with the repo's name), and automatically creates an "origin" handle
- git remote add upstream <forked repo url> : adds an "upstream" handle for the repo you forked

Viewing information about the remote repository

- Listing branches : git branch –a
- Listing aliases/handles : git remote -v

2nd Scenario – Step 2 – Making changes

Say you have modified some files

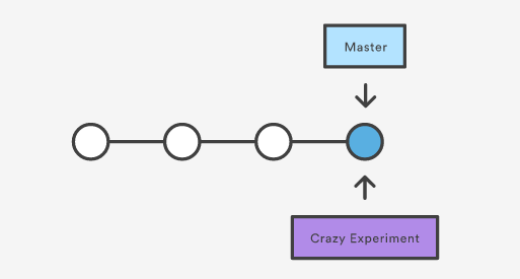
To check what files have been modified

- git diff : *shows the diff for files that are modified but not staged*
  - *--staged : shows the diff for files that are staged but not committed*
- git status

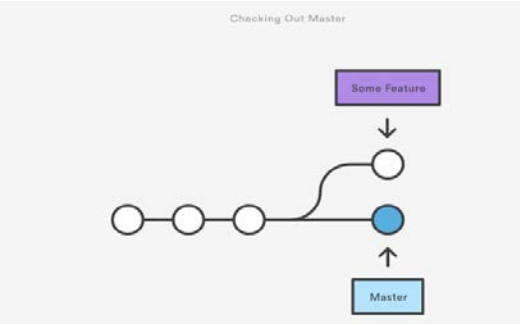
Detour - Branching

Branching

- git branch -a : Lists all the branches in the repository
- git branch <branch> : creates a new branch <branch>. It doesn't checkout the branch. You are still on the old branch. You get a new pointer the current commit.
- git checkout <branch> : Check out the specified branch



- git branch -d <branch> : Deletes the specified branch. This is a "safe" operation in that Git prevents you from deleting the branch if it has unmerged changes.
- git branch -D <branch> : Force delete the specified branch, even if it has unmerged changes.



2nd Scenario – Step 3 – Sync your repo

- git fetch upstream: *fetch the upstream and store its master branch in "upstream/master“*
- *git checkout master*
- *git merge upstream/master*

- git pull upstream : *combining the above two commands*

2nd Scenario – Step 4 – Resolving merge conflicts

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

Must see reference : https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/

Diff tools for mac : https://www.git-tower.com/blog/diff-tools-mac

Diff tools for windows : https://www.git-tower.com/blog/diff-tools-windows/

Smart people on Linux don’t need diff tools

2nd Scenario – Step 5 – Add and Commit

Tag/Move the files to staging area

- git add –A
- git status

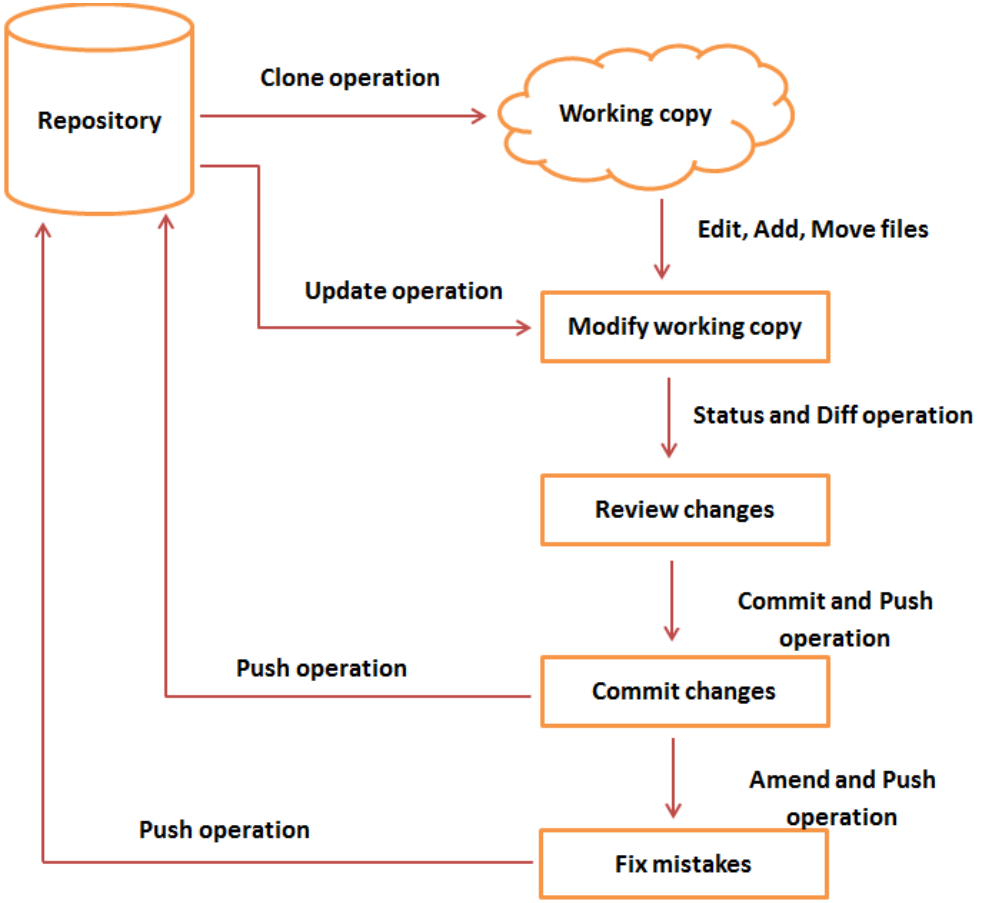
Commit your changes to local repo

- git commit –m “Some good message”

Push the changes

- git push origin <branch> (or git push <handle where you want to push> <working branch>)
- Create a pull request ☺

To Recap


Repository
Clone operation
Working copy
Edit, Add, Move files
Update operation
Modify working copy
Status and Diff operation
Review changes
Commit and Push operation
Push operation
Commit changes
Amend and Push operation
Push operation
Fix mistakes

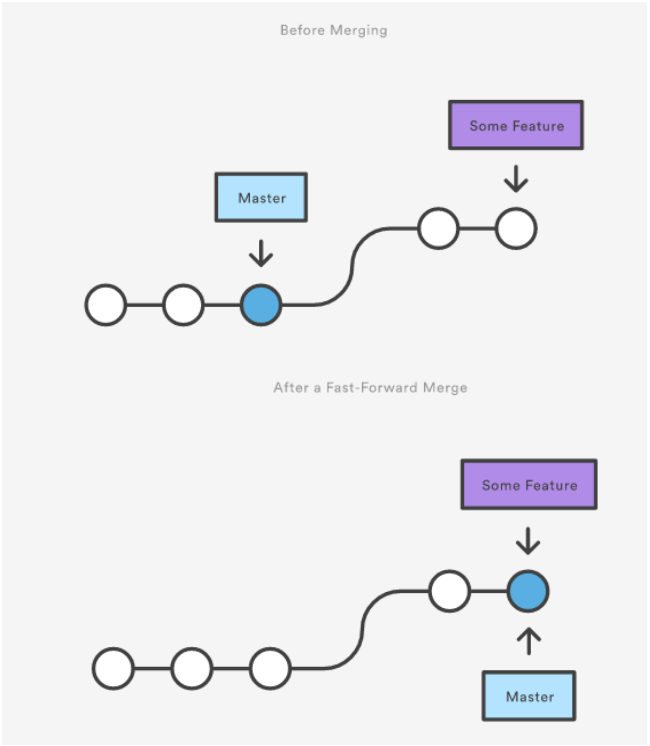
Some Advance Things

Merging

git merge <branch-to-merge> : Merge the specified branch into the current branch. Git will determine the merge algorithm automatically

Git has several distinct algorithms to accomplish merging : a fast-forward merge or a 3-way merge.

Fast forward merge : A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip.



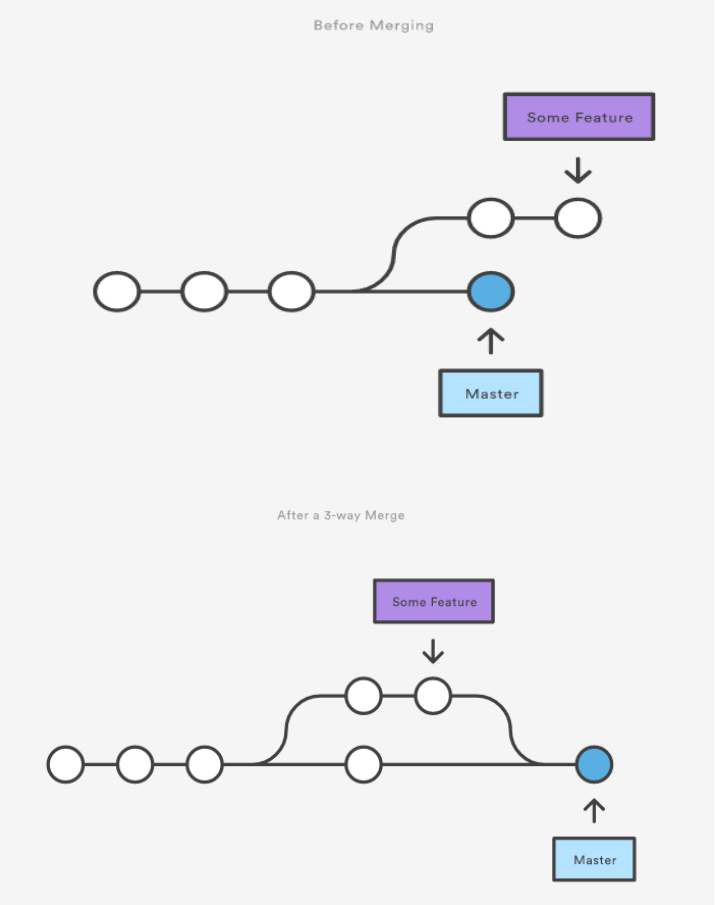
Fast forward example

```
# Start a new feature
git checkout -b new-feature master

# Edit some files
git add <file>
git commit -m "Start a feature"

# Edit some files
git add <file>
git commit -m "Finish a feature"

# Merge in the new-feature branch
git checkout master
git merge new-feature
git branch -d new-feature
```

Merging - 2

3 - way merge : If the branches have diverged fast word merging is not possible. There is no linear path to target branch. In that case, git combine them via 3-way merge.

The nomenclature comes from the fact that Git uses *three* commits to generate the merge commit: the two branch tips and their common ancestor.



3 - way example

```
# Start a new feature
git checkout -b new-feature master

# Edit some files
git add <file>
git commit -m "Start a feature"

# Edit some files
git add <file>
git commit -m "Finish a feature"

# Develop the master branch
git checkout master

# Edit some files
git add <file>
git commit -m "Make some super-stable changes to master"

# Merge in the new-feature branch
git merge new-feature
git branch -d new-feature
```

Stashing

Temporarily shelves (or *stashes*) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

Stashing your work.

After stashing you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

```
$ git status
On branch master
Changes to be committed:

 new file: style.css

Changes not staged for commit:

 modified: index.html

$ git stash
Saved working directory and index state WIP on master: 500
HEAD is now at 5002d47 our new homepage

$ git status
On branch master
nothing to commit, working tree clean
```

Stashing 2 - Re-applying stashed changes

Use git stash pop

Popping your stash removes the changes from your stash and reapplies them to your working copy.

```
$ git status
On branch master
nothing to commit, working tree clean
$ git stash pop
On branch master
Changes to be committed:

 new file: style.css

Changes not staged for commit:

 modified: index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e3
```

Alternatively, you can reapply the changes to your working copy *and* keep them in your stash using git stash apply

```
$ git stash apply
On branch master
Changes to be committed:

 new file: style.css

Changes not staged for commit:

 modified: index.html
```

Stashing 3 - Stash untracked changes

By default, Git *WON'T* stash changes made to untracked or ignored files

Adding -u option tells git to also stash untracked files

```
$ script.js

$ git status
On branch master
Changes to be committed:

 new file: style.css

Changes not staged for commit:

 modified: index.html

Untracked files:

 script.js

$ git stash
Saved working directory and index state WIP on master: 500
HEAD is now at 5002d47 our new homepage

$ git status
On branch master
Untracked files:

 script.js
```

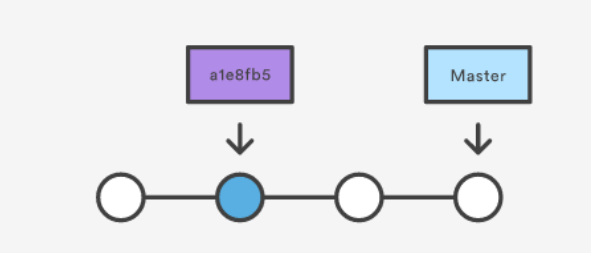
```
$ git status
On branch master
Changes to be committed:

 new file: style.css

Changes not staged for commit:

 modified: index.html

Untracked files:

 script.js

$ git stash -u
Saved working directory and index state WIP on master: 500
HEAD is now at 5002d47 our new homepage

$ git status
On branch master
nothing to commit, working tree clean
```

Undoing changes : Checkout

Checkout serves three functions : checking out files, checking out commits, checking out branch

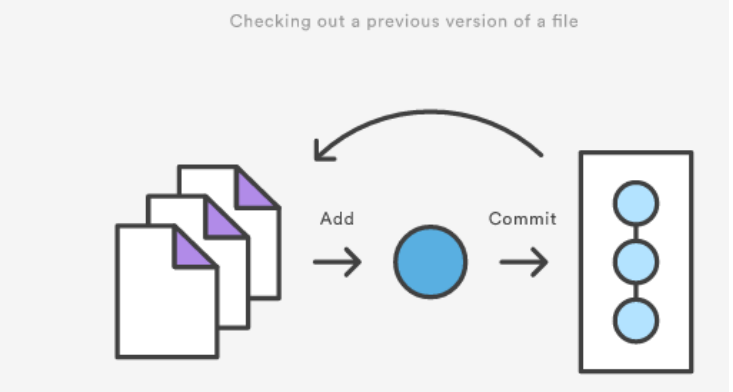
Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way.

- git checkout <commit>
- You can use either a commit hash or a tag. This will put you in a detached HEAD state.
- Checking out an old commit is a read-only operation.



Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

- git checkout <commit> <filename>
- Checking out an old file does affect the current state of your repository.
- You can re-commit the old version in a new snapshot as you would any other file.
- So, in effect, this serves as a way to revert back to an old version of an individual file.
- If you decide you don't want to keep the old version, you can check out the most recent

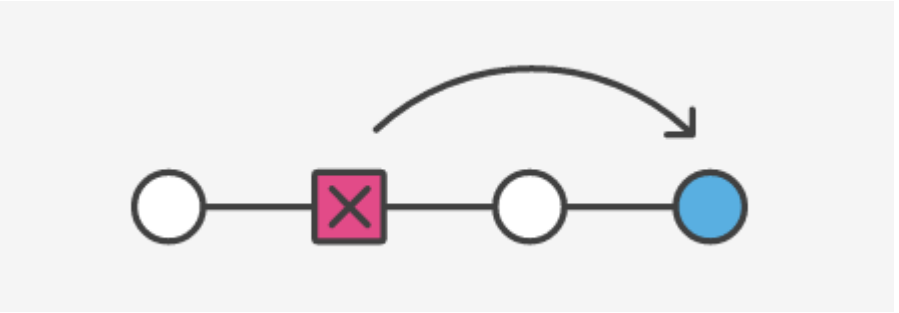


Undoing changes : Revert

git revert <commit> : undoes a committed snapshot. Generate a new commit that undoes all of the changes introduced in <commit> and apply it current branch

Instead of removing the commit from the project history, Git figures out how to undo the changes introduced by the commit and appends a *new* commit with the resulting content.

If you're tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use git revert to automatically do this.



Undoing changes : Reset

git reset : Undo the changes. Dangerous. It’s one of the Git commands that has the potential to lose your work. It not possible to get back.

Versatile

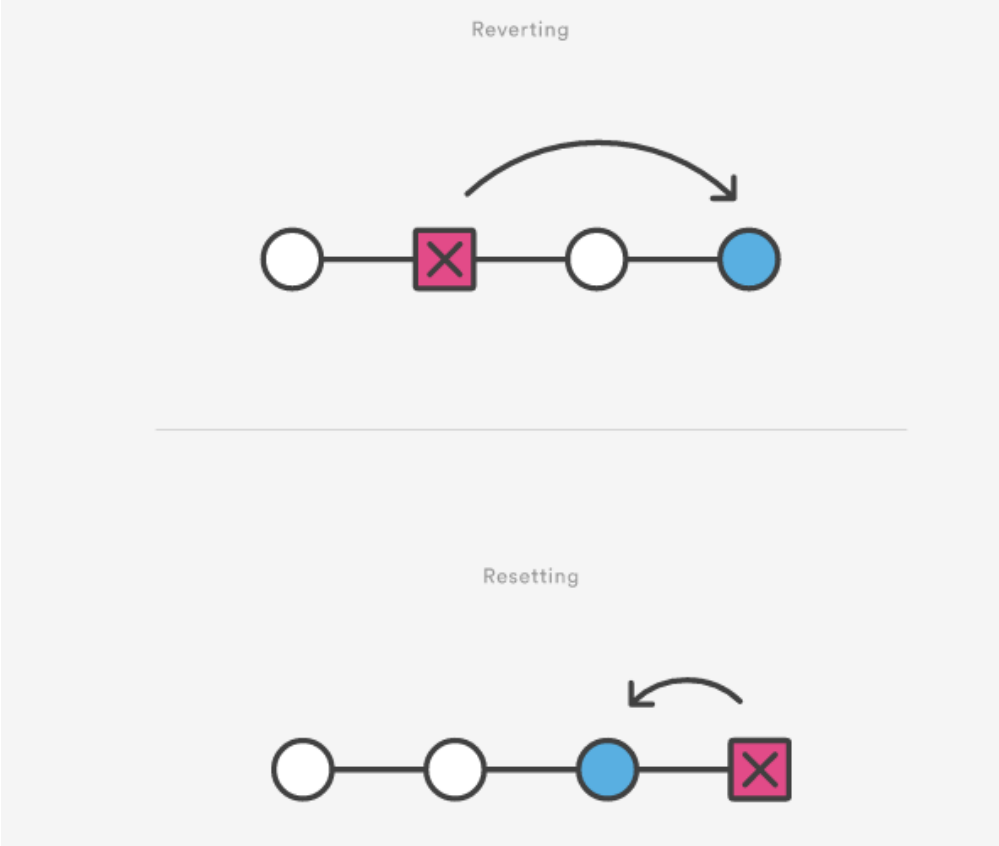
- git reset <file> : Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
- git reset : Reset the staging area to match the most recent commit, but leave the working directory unchanged.
- git reset --hard : Reset the staging area and the working directory to match the most recent commit.
- git reset <commit> : Move the current branch tip backward to <commit> reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.
- git reset --hard <commit> : Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match.

Revert vs Reset

Reverting has two advantages over resetting.

First, it doesn't change the project history, which makes it a "safe" operation for commits that have already been published to a shared repository.

Second, git reset is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backwards from the current commit. For example, if you wanted to undo an old commit with git reset you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits.



Not covered

Rebasing

Cherry-picking

Thank you


GIT PUSH
GO HOME!
memegenerator.net


USED GIT ONCE
GIT MASTER
quickmeme.com