MAE 5032 High Performance Computing: Methods and Practices

Lecture 10: Code version control

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

Motivation

- Codes evolve over time
  - sometimes bugs creep in
  - sometimes the old way was right
  - sometimes it is nice to look back at the evolution

- How can you get back to an old version?
  - keep a copy of very version

    paper_v1, paper_v2, …, paper_2021_aprial_v29, …. paper_final, paper_final_v2, …, paper_final_2022_v3, …. paper_revision_2022, …
  - use a tool optimized for this task
    - version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences
    - version contol helps team work by tracking every individual change and prevent concurrent work from conflicting
    - version control is an essential part of the every day of the modern software team's professional practices

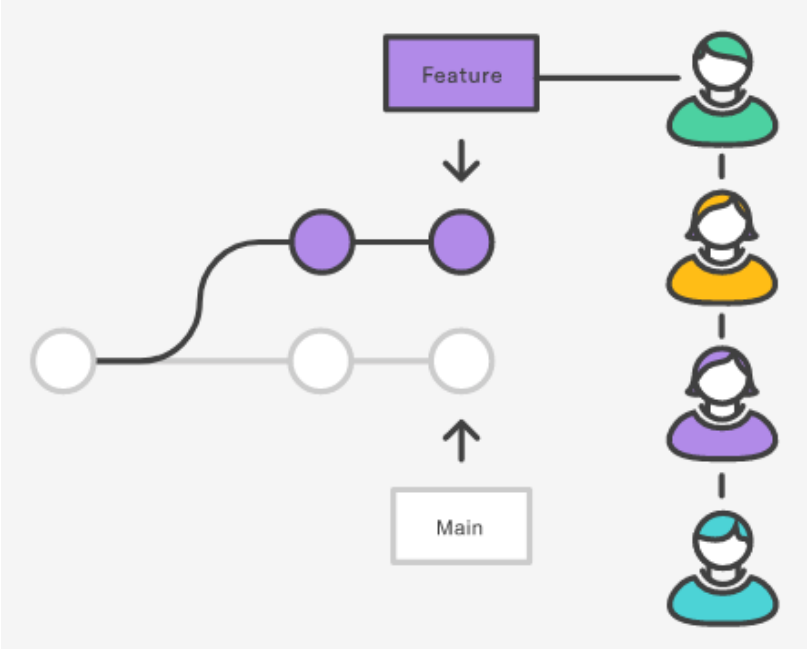
Motivation

REAL Time Machine

VCS and Git

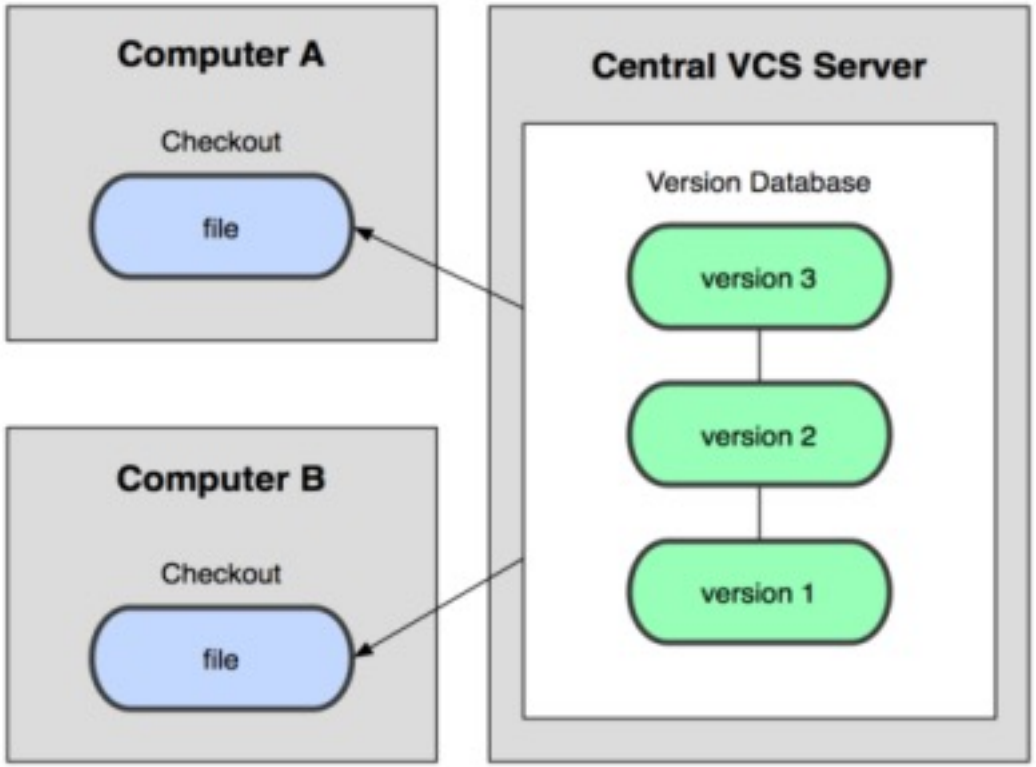
- Git is a **version control system (VCS)** designed to make it easier to have multiple versions of a code base, sometimes across multiple developers or teams
  - it is mature, actively maintained, open-sourced, developed by Linus Torvalds.
  - it is distributed and thus quite efficient and stable.
  - it is secure with a cryptographically secure hasing algorithm with the goal of protecting the code and the change history against both accidental and malicious change.
  - it is flexible in that it support various kinds of nonlinear development workflows.

- Git is good!
- Git is a de facto standard.
- Git can be difficult to learn.



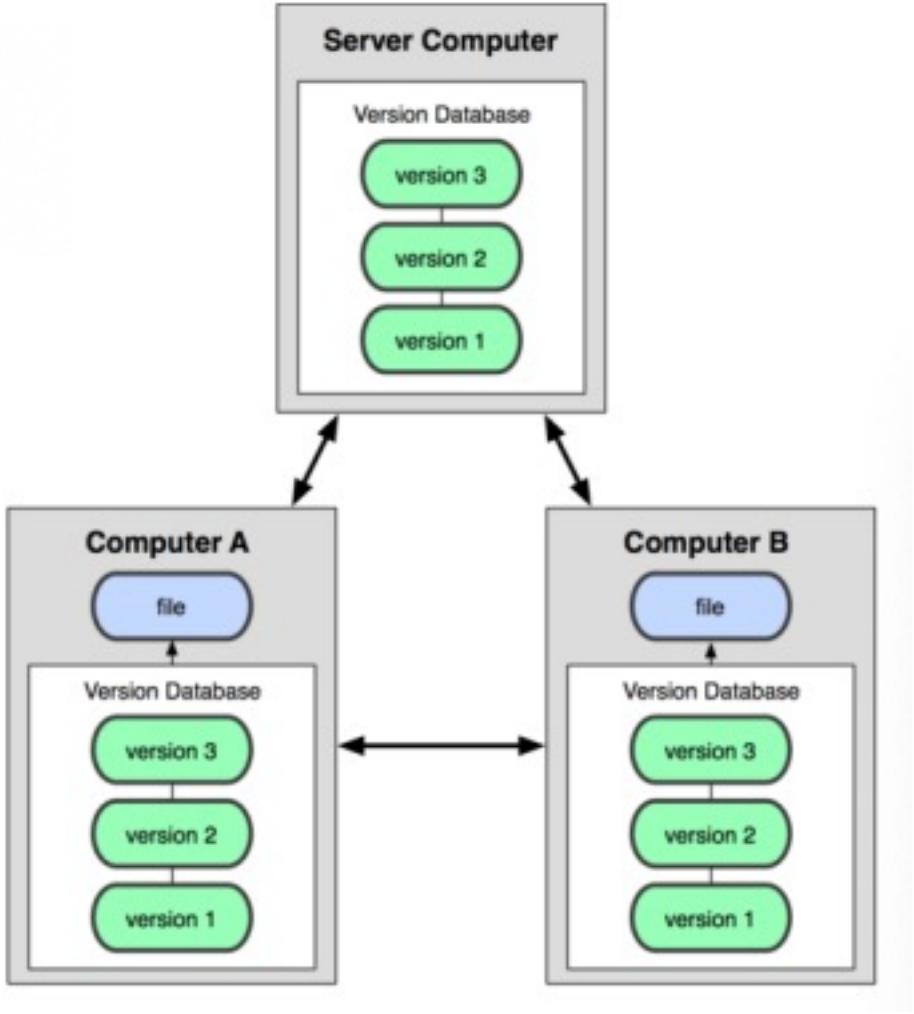
Centralized VCS

- In SVN or CVS, they use a central server repository (repo) to hold the **official** copy of the code
  - the server maintains the sole version history of the repo
- You make checkouts of it to your local copy
  - you make local modifications
  - your changes are not versioned
- When you are done, you check in to the server
  - your checkin increments the repo's version



Distributed VCS

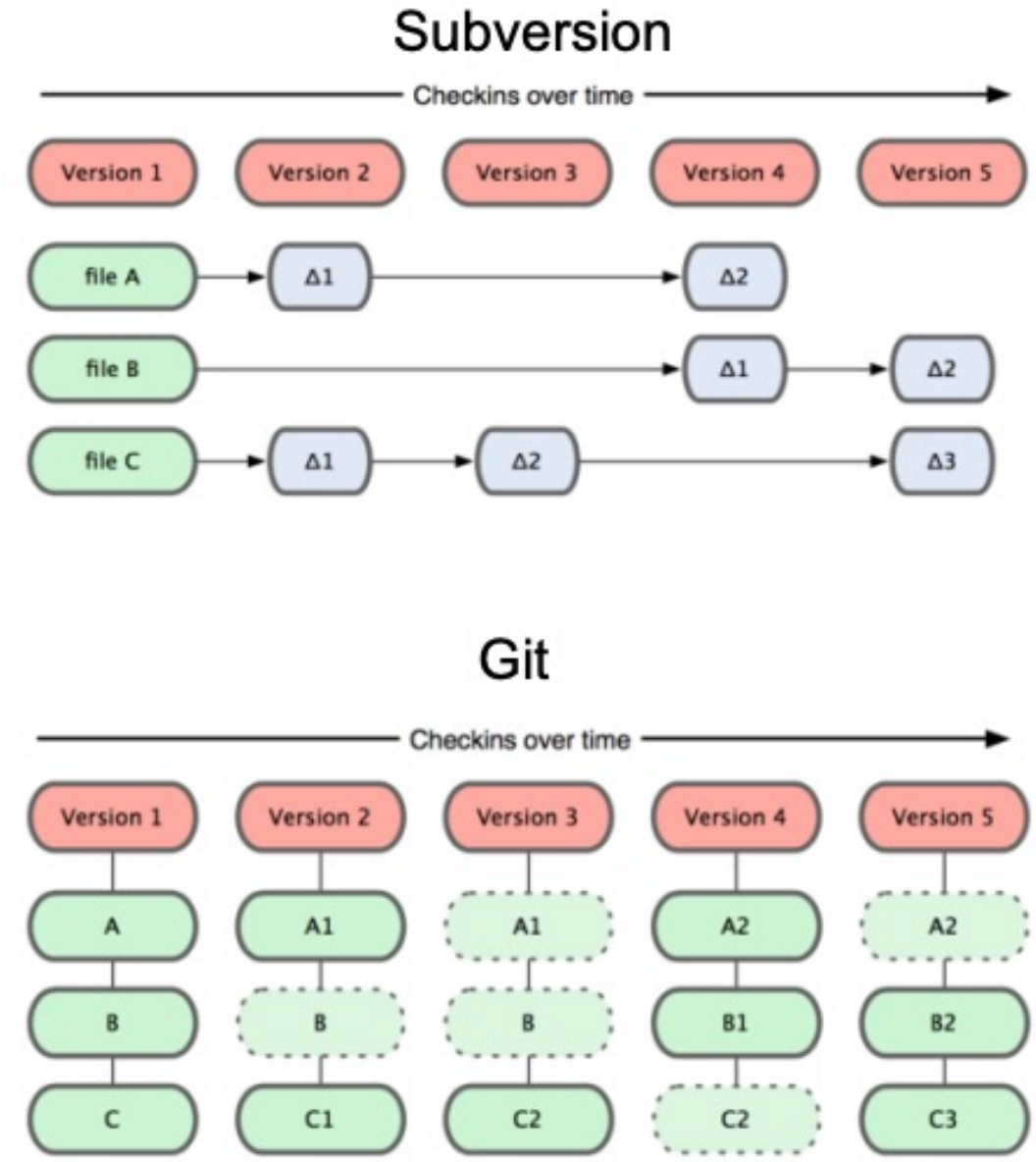
- In git, mercurial, etc., you do not "checkout" from a central repo
  - you clone it and pull changes from it
- Your local repo is a complete copy of everything on the remote server
  - yours is just as good as theirs
- Many operations are local
  - check in/out from local repo
  - commit changes to local repo
  - local repo keeps version history
- When you are ready, you can push changes back to the server
  - back files and facilitate collaboration



Git snapshots

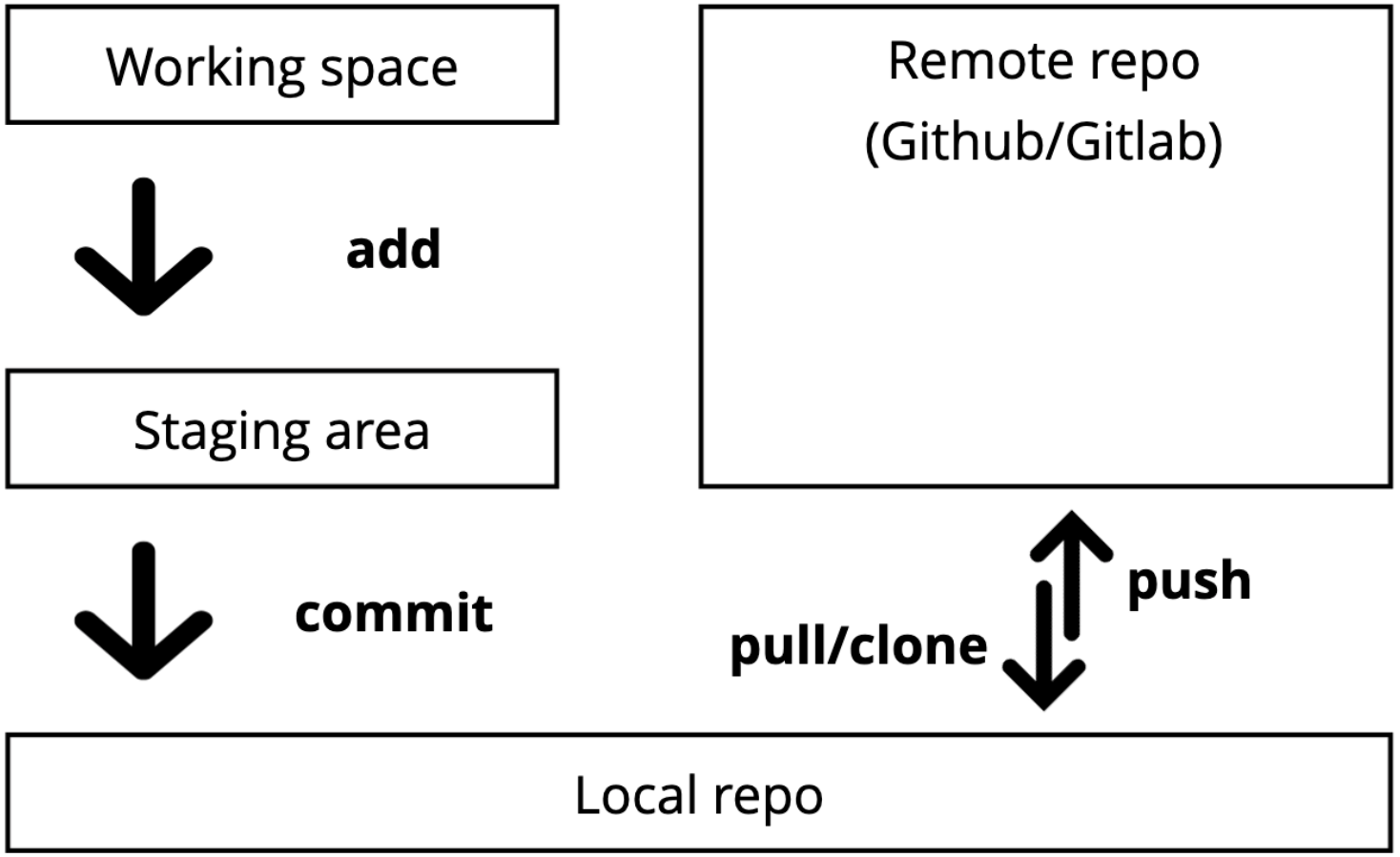
- Centralized VCS like SVN track version data on each individual file

- Git keeps snapshots of the entrie state of the project
  - ➢ each checkin version of the overall code has a copy of each file in it
  - ➢ some files change on a given checkin, some do not
  - ➢ more redundancy, but faster



Local git areas

- In your local copy of git, files can be:
  - In your local repo (committed)
  - checked out and modified, but not yet committed (working copy)
  - or in-between, in a staging area
    - staged files are ready to be comitted
    - a commit saves a snapshot of all staged state.



Install

- Install git is easy

1. From your shell, install git using apt-get:
   ```
   sudo apt-get update
   sudo apt-get install git
   ```

2. Verify the installation was successful by
   ```
   git –version
   ```

3. Configure your git username and email by the following commands. These will be associated with commits that you created
   ```
   git config –global user.name “Ju Liu”
   git config –global user.email “liuj36@sustech.edu.cn”
   ```

Configuration

- Git configuration options are stored in three separate files:
  - local: .git/config – repository specific settings
  - global: /.gitconfig – user specific settings
  - system: /etc/gitconfig – system-wide settings

- You may also create shortcut for a git command:

```
git config –global alias.ct commit
```

- You may define the text editor for use

```
git config –global core.editor “vim”
```

- You may enable colored output for rapid reading

```
git config –global color.ui true
```

Setup a repository

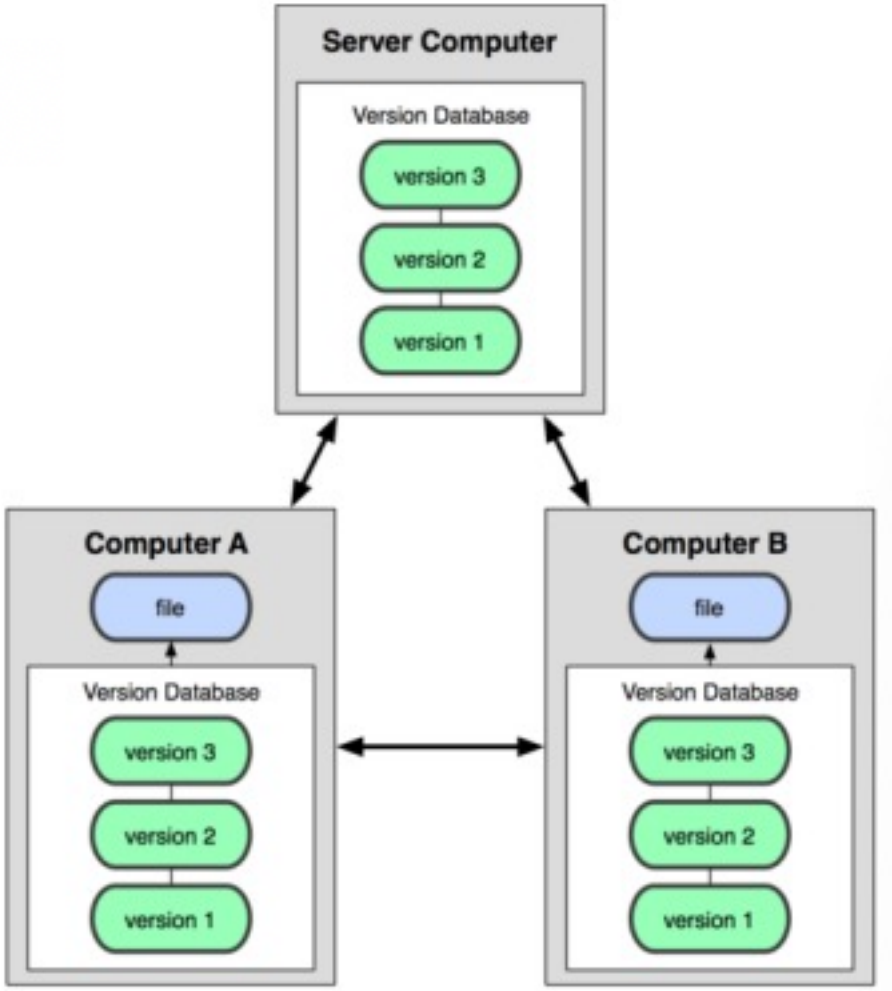
- A git repository (repo) is the .git/ folder inside a project. It tracks all changes made to files in your project.

- To create a repo, cd into your project folder and run
  git init

- If a project has already been set up in a central repository, you may obtain it by
  git clone <repo url>

  git clone git@HOSTNAME:USERNAME/REPO-NAME.git

Once executed, the latest version of the remote repo files on the main branch
will be pulled down and added to a new folder.



Saving changes to the repository

- You can add a file to the staging area by

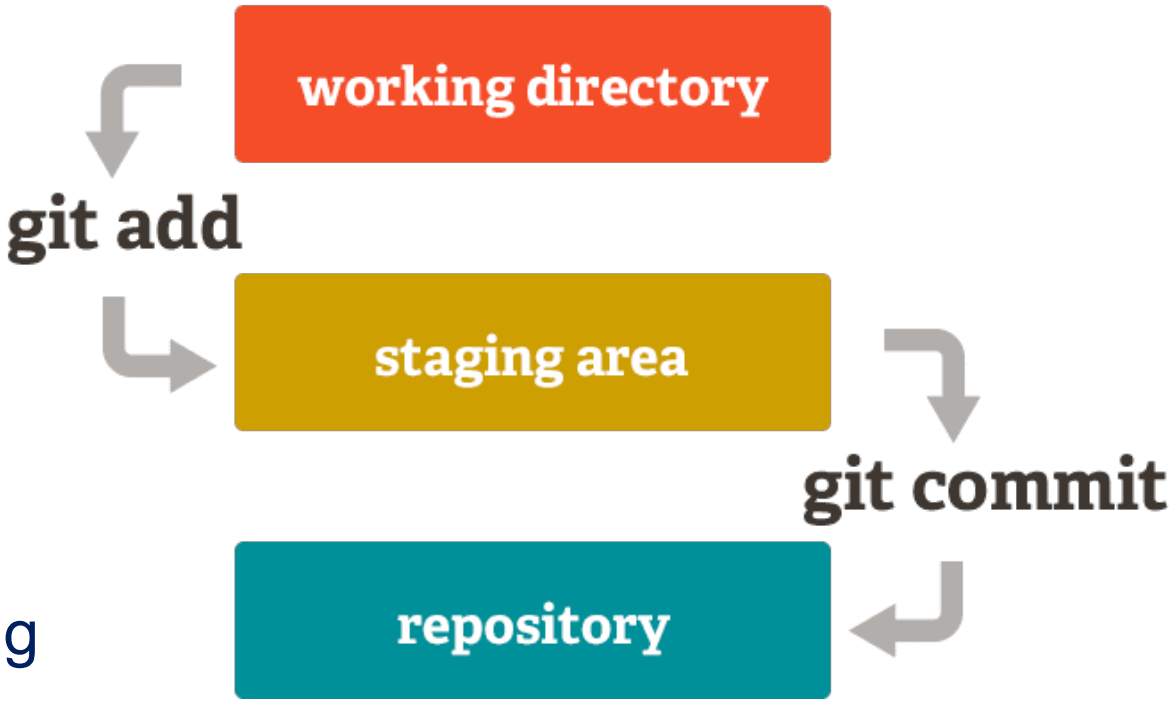
```
git add filename
git add *.txt
git add .
```

- You can send the staged files to the repository

```
git commit –m “your log message”
```

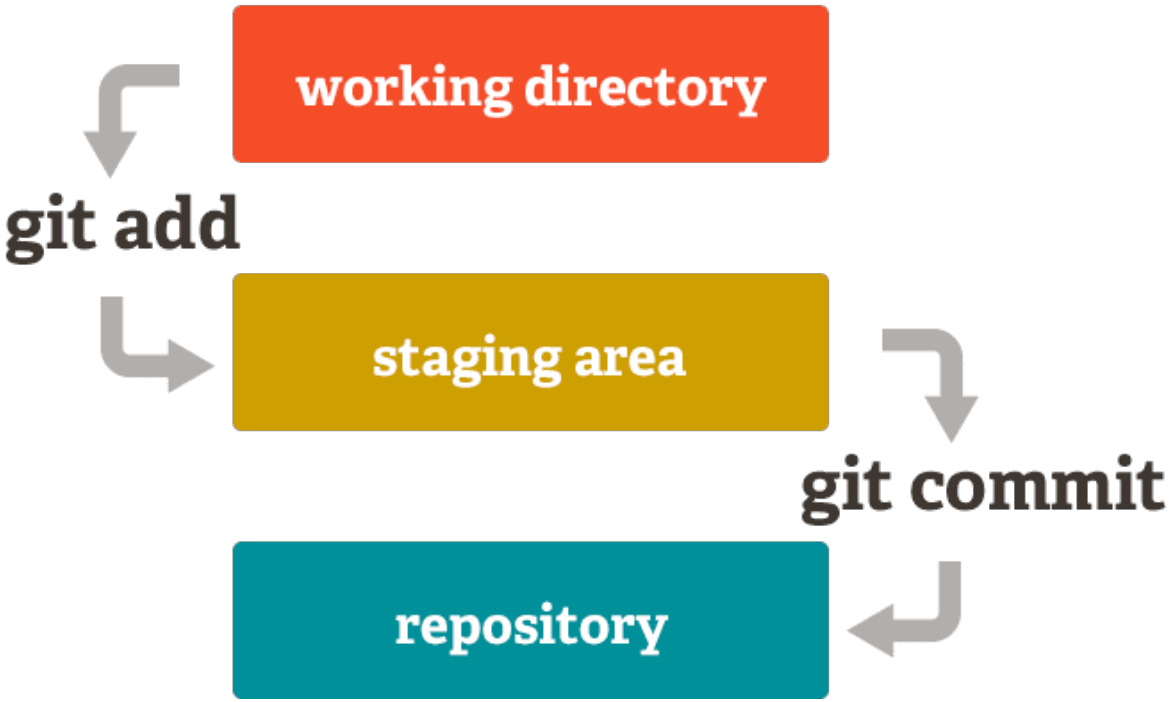
- You can send a file directly from the working directory to the repo by

```
git commit –a –m “your log message”
git commit –am “your log message”
```



Saving changes to the repository

- git commit takes snapshots of your project. Your log message shall explain the state of the commit.
- each commit is identified by a ID generated by SHA (secure hash algorithm) which is 40-digits long. Most of the time, git shows the first 7 digits of it.
- you may accumulate commits in your local directory and push them to remote repository at anytime later.



Analyzing the state of the repo

- Compare working directory and staging area `git diff filename`
- Compare the staging area and repo `git diff --cached filename`
- Compare working directory and repo `git diff head`
- You can inspect what files are staged, unstaged, and untracted by

```
git status
git status -s
```

- git log displays committed snapshots

```
git log
git log -oneline
git log -oneline
--reverse
```

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file3.txt
```

.gitignore

- Git sees every file in your working directory as one of three states:
  - tracked -- a file which has been previously staged or committed
  - untracked – a file which has not been staged or committed
  - ignored – a file which Git has been explicitly told to ignore

- There are files that we do not want to track:
  - compiled codes .o, .a, a.out, .so
  - files genereated at run time: .log
  - hidden system files: .DS_Store
  - build output directory /bin, /lib, etc.

- Ignored files can be specified in .gitignore at the root of your project foler.

- https://github.com/github/gitignore

Remove and rename files

- You can do rm and do git add to save the state:

- Or simply do git remove command

```
git rm filename
```

- You may use cached option to remove the file from repo, and it will remain in the working directory

```
git rm --cached filename
```

- Similarily, if you use Linux mv command to rename a file, git will understand it as it is removed and a new file created

- Or you can use git mv

```
git mv oldfilename newfilename
```

equals `git rm oldfilename; git add newfilename`

Undo changes

- git checkout will take you to a previous commit using its identifying hash.

```
git checkout SHA-ID
```

- git checkout will undo changes in working directory back to its state in repo

```
git checkout filename
git checkout .
```

- git clean will remove untrackted files

```
git clean -n # shows files to be removed
git clean -f # force the remove operation
```

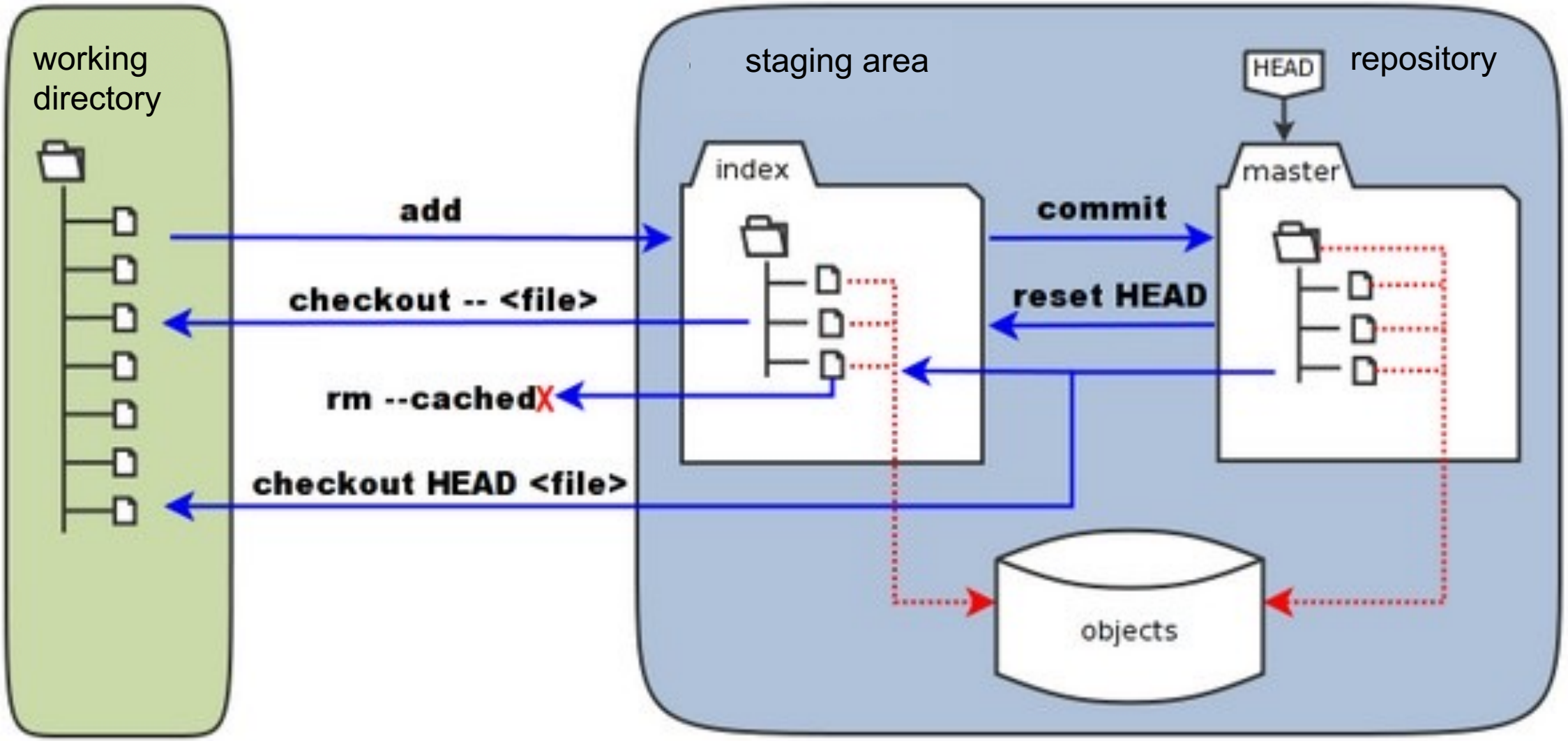
- git revert will record a new commit to reverse the effect of earlier commits

```
git revert --no-edit HEAD
```

- git reset can help the staging area to match the most recent commit, leaving the working directory unchanged.

```
git reset
```

Summary of local repo operations



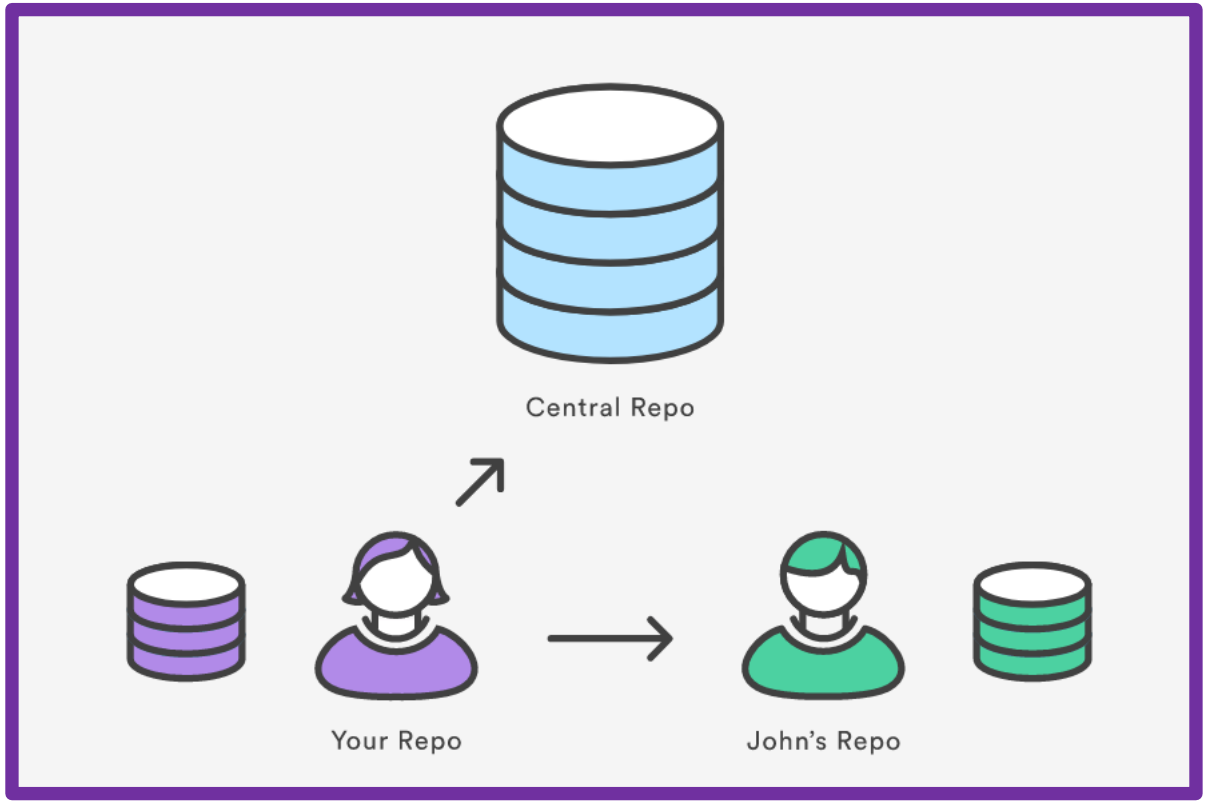
Git remote

git remote command lets you create, view, and delete connections to other repositories.

```
git remote add <name> <url>
```

```
git remote rm <name>
```

Git-based projects call their central repo origin.



Github and Gitee

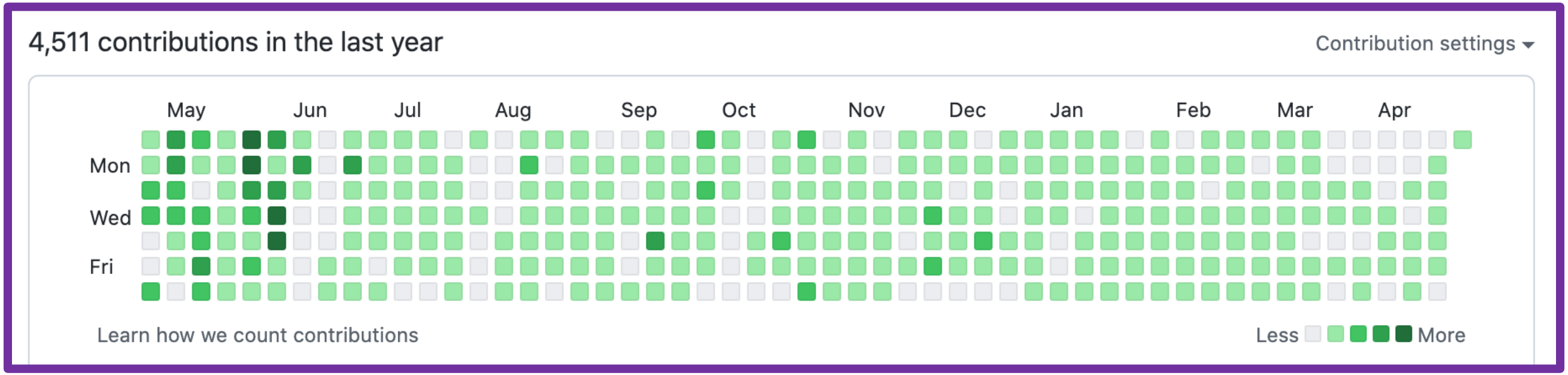
- Github.com is a site for online storage of git repositories
  - you can put your remote repo there and push code to it
  - many open source code use it, such as Linux kernel
  - you can get a free space for open source projects
  - alternatives include bitbucket, gitlab, etc.

- Gitee is a site similar to github, owned by China.
  - The company is in Shenzhen, Nanshan ☺

Github and Gitee

超过 200,000 家企业/机构的信任之选

Github and Gitee



My commit history in the past 12 months.

Extremely useful in job hunting!

Git fetch, pull, and push

- git fetch download contents from a remote repository, but it does NOT integrate any of this new data into your working files.

```
git fetch <remote repo> <branch>
e.g. git fetch origin
```

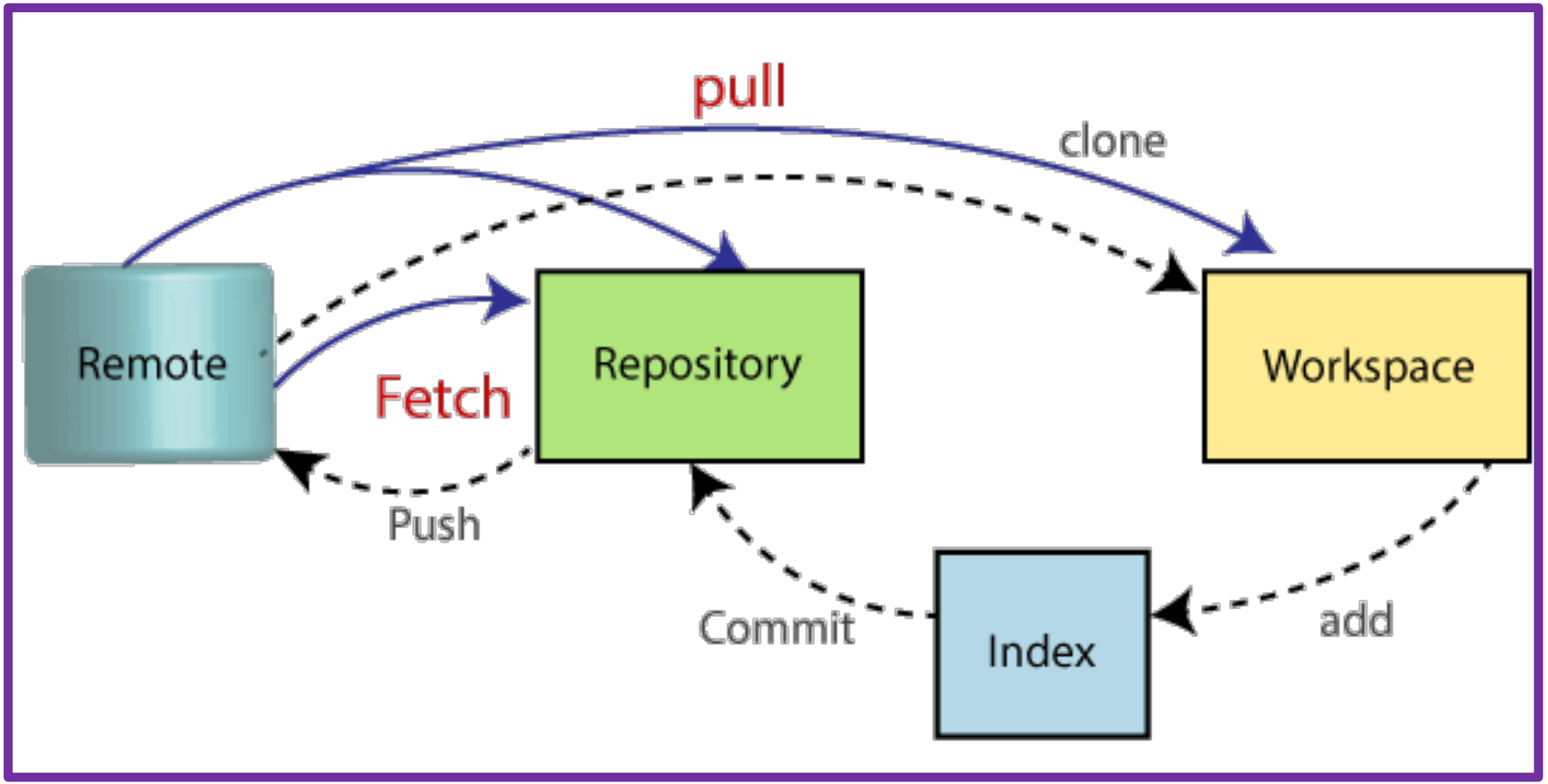
- git pull is used to update your current HEAD with the latest changes from the remote server.

```
git pull <remote repo> <branch>
e.g. git pull origin master
```

- git push is used to upload local repository content to a remote repository.

```
git push <remote repo> <branch>
```

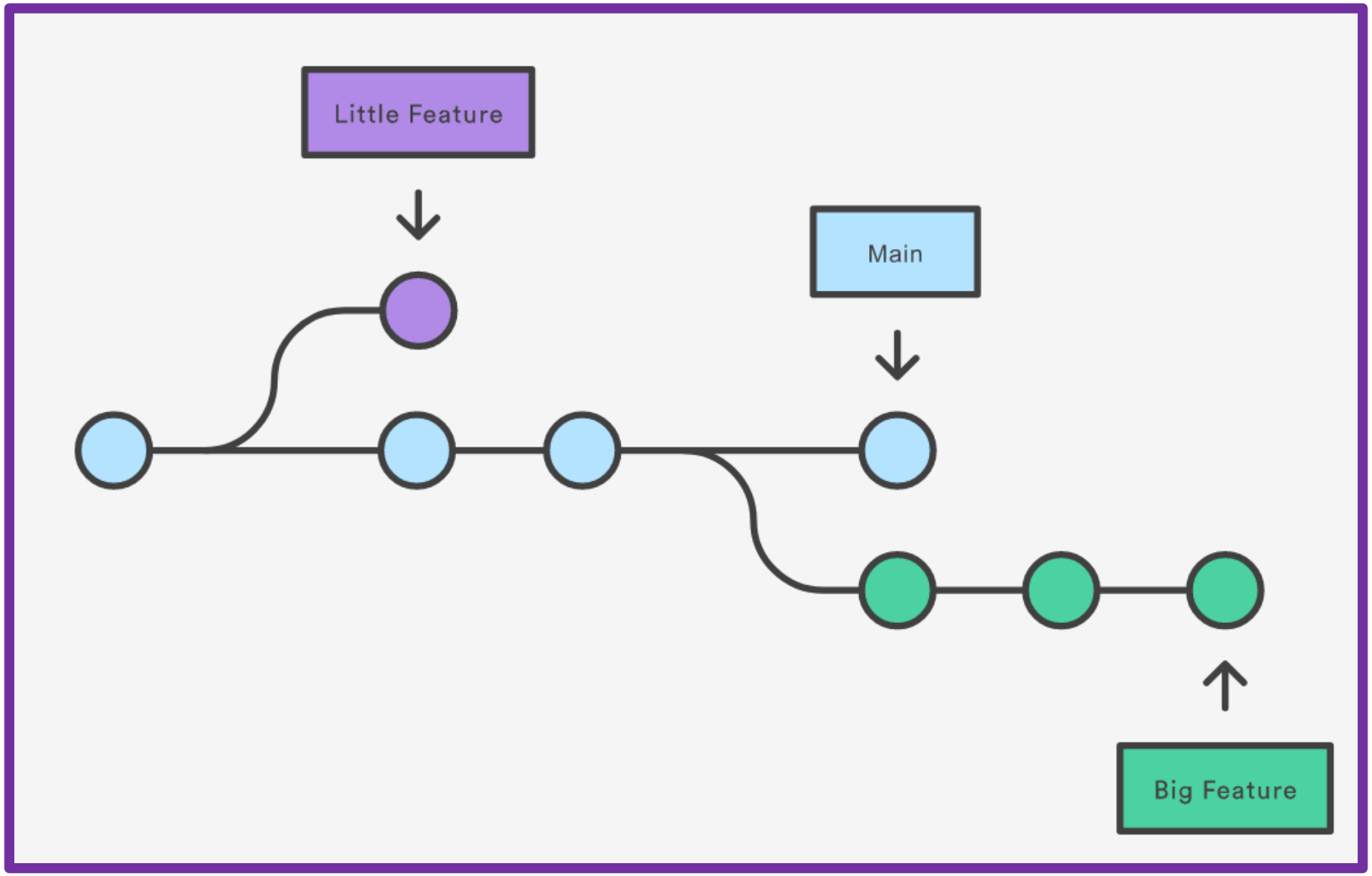
Git fetch, pull, and push



Git branch

branch represents an independent line of development;

branch head is the tip of a series of commits



Git branch and checkout

- git branch lists all branches in your repository
- git branch <branch-name> creates a new branch with name branch-name
- git branch –d <branch-name> deltes a specified branch

git checkout is useful for switching between different versions of a target entity (files, **commits**, and branches).

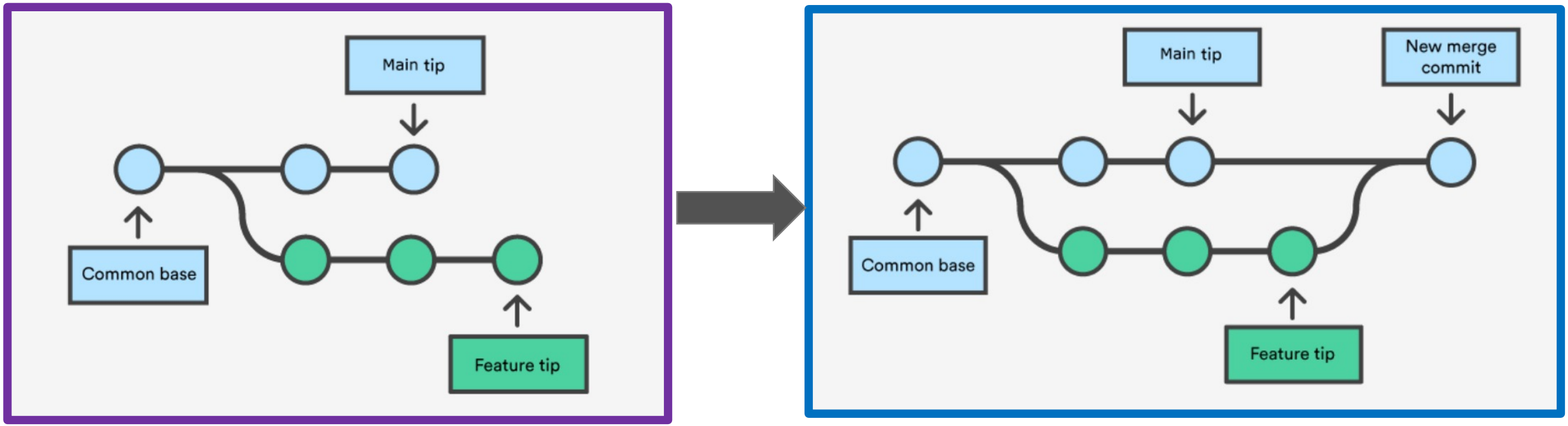
git use HEAD to point to the current snapshot. checkout simply updates the HEAD to the specified snapshot.

- git checkout –b <new-branch> creates and checks out a branch named new-branch
- git checkout <branch-name> switch to the HEAD tip of the branch with name branch-name

Git merge

- Git merge will combine multiple sequences of commits into one unified history. Git will try to do the merge automatically.

- If a piece of file is modified in both branches, git will be unable to merge for that file. This is a version control conflict. User intervention becomes necessary.



Git merge steps

- Do a checkout to make the HEAD pointing to the correct merge-receiving branch
- fetch the updated remote changes
- pull to make the receiving branch updated
- git merge <branch-name>, branch-name is the name of the branch to be merged into the receiving-branch
- if the branch-name is no more needed, delete it by git branch -d branch-name

Git merge steps

- Git will notify you if there are conflicts:

```
Auto-merging file2.txt
CONFLICT (content): Merge conflict in file2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- run status to see details

```
-> git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Git merge steps

- Git will edit the conlifcted file with visual indicators: <<<<<<<< and >>>>>>>>>
- It is easy to search these indicators in text files
- The content before ====== marker is the receiving branch and the part after is the merging branch
- Fix it by hand and do a normal git commit
- Git pull = git fetch + git merge

so you may encounter conflict when do pulling

```
<<<<<<< HEAD
=======
hello!
>>>>>>> new-feature
```

Best practices

- Do not make small or big commits
  - commits are cheap and is a snapshot that the code base can be reverted to if needed.
  - you do not want to make it too small (e.g. single file per fommit)
  - Ensure your committed code can be compiled at very least
- Ensure you are working from latest version
  - It's easy to have a local copy of the codebase fall behind the global copy. Use git pull to avoid conflicts.
- Make detailed notes
  - Commit log messages are like your code comments. Help track changes for future contributors.
- Review changes before committing
  - There is a staging area. It can be used to collect a group of edits before writing them to a commit. Using the staging area to review the changes before commiting.
- Use branches
  - Branching allows developers to create separate lines of development. These lines are generally different product features. When development is complete on a branch, it is then merged into the main line of development.