1. Git, Git Features, .gitignore, SCM, Repository

---------------------------------------------------

Git is a distributed version control system that allows multiple developers to collaborate on a project by tracking changes in files.

It enables branching, merging, and reverting, making it easier to handle versioning.

Git's key features include easy branching and merging, speed, and efficient handling of large repositories.

The .gitignore file is used to tell Git which files or directories to ignore, like build artifacts or system files.

SCM (Source Code Management) tools like Git track code changes, manage versions, and facilitate collaboration.

A repository is a directory containing your project and its history, with all commits, branches, and tags stored in a hidden .git directory.

2. git restore, --staged, --staged --worktree

-----------------------------------------------------

The git restore command is used to discard changes in the working directory or the staging area.

Using git restore --staged <file> removes a file from the staging area but keeps it in the working directory, essentially undoing the git add.

When using git restore --staged --worktree <file>, it discards changes both from the staging area and the working directory, reverting the file to its last committed state.

This can be helpful if you want to completely discard local modifications and start fresh from the latest commit.

3. git merge, its Types (Fast Forward, Three-Way, with Conflict, without Conflict Merge)

---------------------------------------------------------------------------------------------------------

The git merge command is used to combine changes from one branch into another. There are different types of merges:

- Fast-Forward Merge: Happens when the current branch is directly ahead of the branch being merged, so no new commit is created.

- Three-Way Merge: Occurs when there are divergent changes between two branches, requiring Git to create a new commit combining the histories.

- Merge with Conflict: When changes in both branches affect the same part of a file, Git can't automatically merge them. You'll need to manually resolve the conflict.

- Merge without Conflict: Happens when changes in both branches are in different parts of the file, allowing Git to automatically merge without manual intervention.

4. Revert

-------------------

The git revert command is used to undo changes by creating a new commit that reverses the effect of a previous commit.

This is useful when you want to preserve the commit history but need to undo a specific change.

Unlike git reset, which can modify history, git revert is a safe way to undo changes, especially in a shared or public repository.

To revert a commit, you use the command git revert <commit-hash>, which creates a new commit that undoes the specified commit's changes.

5. Rebase

------------------------

The git rebase command is used to integrate changes from one branch into another by moving the entire branch to a new base commit.

It’s often used to create a cleaner, linear history by "replaying" commits on top of another branch.

While rebase avoids the creation of merge commits (unlike git merge), it can rewrite commit history, which can be dangerous in shared branches.

Rebase is commonly used in workflows that involve updating feature branches with the latest changes from the main branch.

The basic command is git rebase <branch>, and you can resolve any conflicts that arise during the process.

6. Restore

-------------------

The git restore command is designed to revert changes in your working directory or index (staging area).

It allows you to discard local modifications or to restore a file to a specific commit state.

You can restore individual files, all files, or specific parts of files.

For instance, using git restore <file> will revert the file back to its state in the most recent commit.

You can also use git restore --staged <file> to unstage a file, effectively removing it from the staging area without affecting the working directory.

7. git rm

----------------------

The git rm command is used to remove files from both the working directory and the Git index (staging area).

This command ensures that the file is not only deleted from the working directory but also staged for deletion in the next commit.

If you want to keep the file in your local directory but remove it from version control, use the --cached flag, like so: git rm --cached <file>.

This is commonly used when files should be removed from Git tracking (for example, large files that were mistakenly added).

8. git log

---------------------

The git log command shows the commit history of a Git repository.

By default, it lists all commits in reverse chronological order, showing the commit hash, author, date, and commit message.

You can customize the log output using various options such as git log --oneline for a simplified view, or git log --graph to see a visual representation of the commit history.

Additionally, you can search for commits containing specific changes using git log --grep=<search-term> or filter commits by date and author.

9. git commit -m, --amend, Related

-------------------------------------------------

The git commit -m command is used to create a commit with a message that describes the changes made.

The message should be concise but informative about the work performed in that commit.

The --amend option allows you to modify the last commit, which is useful when you want to change the commit message or add changes to it.

For example, git commit --amend can be used to update a commit message after committing.

However, using --amend rewrites history, so it should be used with caution in shared repositories.

10. git branch, git checkout, git checkout -b, git branch -d <branch>

---------------------------------------------------------------

The git branch command is used to list, create, or delete branches in a Git repository. You can create a new branch using `git branch <branch-name>`. The `git checkout` command is used to switch between branches or restore working directory files. For creating a new branch and switching to it in a single command, you can use `git checkout -b <branch-name>`. To delete a branch, you can use `git branch -d <branch-name>`, which deletes the branch locally. If the branch contains unmerged changes, Git will prompt you to either merge or force delete the branch.

11. git show

---------------------------

The `git show` command is used to display detailed information about any Git object such as commits, tags, and files. It can be used to view the commit history with more detailed output. The most common usage is `git show <commit-id>`, which displays the changes introduced in a specific commit, including the commit message, author, date, and diffs. It can also be used to show information about a specific tag, branch, or even files.

12. git tag -a <tag> <commitId> -m <message>

----------------------------------------------------------

The `git tag` command is used to create a tag for a specific commit, usually for marking release points like v1.0, v2.0, etc. The `-a` flag creates an annotated tag, which stores additional information such as the tag message and the author's name. The `-m` flag allows you to add a message to the tag. The command `git tag -a <tag-name> <commit-id> -m <message>` creates a new tag at the specified commit, with an optional message attached.

13. git reset soft, mixed, hard

------------------------------------------------

The `git reset` command is used to undo local changes and move the HEAD pointer to a specific commit. There are three main types of resets:

- `git reset --soft <commit>`: This moves the HEAD to the specified commit but leaves the changes staged in the staging area (index).

- `git reset --mixed <commit>`: This resets the HEAD and the staging area but leaves the working directory files unchanged.

- `git reset --hard <commit>`: This resets the HEAD, the staging area, and the working directory. This is a destructive reset that discards all changes in the working directory and staging area.

14. Diff between restore, rebase, revert, reset, merge

------------------------------------------------------

- \*\*git restore\*\*: The `git restore` command is used to discard changes in your working directory or staging area.

If you have made modifications that you don't want to keep, you can use `git restore <file>` to revert the file to its state in the last commit.

You can also use `git restore --staged <file>` to unstage a file, removing it from the staging area but keeping the changes in the working directory.

It’s a safer alternative to `git checkout` for discarding changes.

- \*\*git rebase\*\*: `git rebase` is used to apply commits from one branch onto another, effectively changing the base of the current branch.

Rebasing rewrites history by creating new commits on top of an existing branch.

It’s particularly useful for integrating feature branches into the main branch while maintaining a cleaner, linear commit history.

However, be cautious when rebasing shared branches because it can alter commit hashes and cause issues for collaborators.

- \*\*git revert\*\*: The `git revert` command is used to undo the changes introduced by a previous commit by creating a new commit that reverses those changes. This is a safe way to undo changes in a public repository because it doesn’t alter the commit history.

For example, `git revert <commit-id>` creates a new commit that undoes the effects of the specified commit, without affecting any subsequent commits.

- \*\*git reset\*\*: The `git reset` command is used to reset your current HEAD to a specific state, with three main options:

- `--soft`: Moves the HEAD to the specified commit but leaves changes staged.

- `--mixed`: Resets the HEAD and the staging area, but keeps the working directory unchanged.

- `--hard`: Resets the HEAD, staging area, and working directory, discarding all changes. This is a destructive operation, and the changes are lost unless you have a backup.

`git reset` is powerful for undoing changes but can be dangerous if used incorrectly, especially with `--hard`.

- \*\*git merge\*\*: The `git merge` command is used to combine changes from two different branches.

When you merge one branch into another, Git tries to automatically combine the changes.

If there are no conflicting changes, a merge commit is created to record the merge. If conflicts exist, Git will prompt you to resolve them manually before completing the merge. Merging is a non-destructive operation and preserves the history of both branches.

15. git init

---------------------

The `git init` command is used to initialize a new Git repository in a directory. It creates a `.git` folder in the current directory, marking it as a Git repository. Once initialized, you can begin tracking files, making commits, and performing other Git operations.

16. git 3 stages: working dir, staging area, local repo

----------------------------------------------------------------------

Git uses three main stages to track changes:

- \*\*Working Directory\*\*: The actual files in your project, where you make changes.

- \*\*Staging Area\*\*: The area where you place changes that are ready to be committed. Files are added here using `git add`.

- \*\*Local Repository\*\*: The Git database where your commits are stored. This is managed by Git and contains the history of your project.

17. git pull

-----------------

The `git pull` command is used to fetch and integrate changes from a remote repository into your local branch. It is equivalent to running `git fetch` followed by `git merge`. This command is commonly used to synchronize your local repository with changes made by others in the remote repository.

18. git fetch

-----------------

The `git fetch` command is used to download changes from a remote repository without applying them to the local working directory. This updates the remote-tracking branches in your local repository. You would use `git fetch` to see changes made by others before deciding to merge or rebase those changes into your branch.

19. Diff between pull and fetch

-------------------------------------------

- \*\*git pull\*\*: Fetches changes from the remote repository and automatically merges them into the current branch. It is a combination of `git fetch` and `git merge`.

- \*\*git fetch\*\*: Fetches changes from the remote repository but does not automatically merge them. It updates the remote-tracking branches but requires manual merging or rebasing.

20. What is a Pull Request

-------------------------------------

A pull request (PR) is a feature provided by platforms like GitHub and GitLab to propose changes from one branch to another. It's used to review and discuss changes before merging them into the main codebase. A PR allows team members to comment, suggest modifications, and approve changes before they are finalized.

21. Merge Conflict

--------------------------

A merge conflict occurs when two branches being merged have changes in the same part of a file. Git cannot automatically decide which changes to keep, so it marks the file as conflicted. The user needs to manually resolve the conflict, choosing which changes to keep or combining both sets of changes.

22. git clean

--------------------

The `git clean` command is used to remove untracked files and directories from the working directory. It helps to clean up files that are not being tracked by Git, such as temporary files, build artifacts, or files added to `.gitignore`. Use `git clean -n` to preview which files will be removed, and `git clean -f` to actually remove them.

23. git add ., -A, <file>

--------------------------------------

- `git add .`: Stages all changes (new, modified, deleted) in the current directory and its subdirectories.

- `git add -A`: Stages all changes, including files that have been deleted, modified, or added in the entire repository.

- `git add <file>`: Stages a specific file for committing, allowing you to commit specific changes rather than all changes.

24. How does git primarily store data

------------------------------------------------

Git primarily stores data as snapshots of the filesystem. Each commit in Git represents a snapshot of your entire project at a given point in time. These snapshots are stored in a compressed and efficient way in the `.git` directory. Git uses objects like blobs, trees, and commits to store the project's history.

25. Origin

----------------------

"Origin" is the default name for a remote repository in Git. It is a shorthand reference to the URL of the repository from which you cloned a project or the repository you’re collaborating with. It is used in commands like `git push origin <branch>` to push local changes to the remote repository.

26. Different strategies to reduce merge conflict

-------------------------------------------------------------------

To reduce merge conflicts:

- Use feature branches for individual tasks.

- Regularly pull the latest changes from the main branch to stay updated.

- Communicate with team members to avoid working on the same lines of code.

- Use smaller, focused commits to minimize conflict potential.

- Resolve conflicts early by rebasing or merging frequently.

27. Purpose of remote repository

---------------------------------------------------

A remote repository is used to store code that is accessible to multiple developers or teams. It serves as a centralized location to manage and share changes. Remote repositories are commonly hosted on platforms like GitHub, GitLab, or Bitbucket. They allow teams to collaborate, share code, and maintain a central version of the project.

28. git push

---------------------

The `git push` command is used to upload local repository content to a remote repository. It sends the committed changes in your local branch to the corresponding branch on the remote repository. The basic syntax is `git push <remote> <branch>`, where `<remote>` is usually "origin", and `<branch>` is the name of the branch you want to push.

29. Why do we use SSH keys in Git

----------------------------------------------

SSH keys are used to securely authenticate with remote Git repositories without needing to provide your username and password every time. SSH keys provide a more secure and convenient way to connect to remote repositories compared to using HTTPS. They are used in platforms like GitHub and GitLab to authenticate access and ensure data transmission is encrypted.

30. What is forking a repository means

-------------------------------------------------

Forking a repository means creating a personal copy of someone else’s repository on GitHub or GitLab. This allows you to experiment with changes without affecting the original project. Forking is commonly used in open-source contributions, where you can fork a repo, make changes, and then submit a pull request to the original repository for review and potential merging.

31. git config

----------------------

The `git config` command is used to set configuration options in Git. These settings can be applied globally or locally. You can configure user information (name and email), editor, merge tool, and many other preferences. The global configuration applies to all repositories, while local settings apply to a specific repository. Common examples include `git config --global user.name "Your Name"` and `git config --global user.email "your.email@example.com"` to set your global identity.

32. git push -u origin <branch>

-----------------------------------------

The `git push -u origin <branch>` command is used to push a local branch to a remote repository (typically named "origin") and set the upstream tracking reference. The `-u` flag allows Git to remember the remote and branch so that in the future, you can simply use `git push` or `git pull` without needing to specify the remote or branch every time.

33. Git Hooks

-----------------------

Git hooks are scripts that run automatically at specific points in the Git workflow, allowing developers to automate processes, enforce standards, or integrate with other tools. They are located in the `.git/hooks` directory of a repository and can be customized to meet project requirements. Git hooks can be divided into two categories: client-side hooks (which run on the local machine) and server-side hooks (which run on the Git server).

- \*\*pre-commit\*\*: This hook runs before a commit is made. It's typically used for tasks like checking the commit message format or ensuring that there are no spelling errors in the commit message. For example, it can prevent commits if the message does not follow a specific template or standard.

- \*\*pre-receive\*\*: This server-side hook runs before a push is received by the remote repository. It's commonly used to enforce project coding standards, such as ensuring that all commits pass unit tests, comply with a style guide, or do not introduce security vulnerabilities.

- \*\*post-commit\*\*: This hook runs after a commit has been created. It's often used for tasks like sending notifications (via email or SMS) to team members to alert them about new commits, ensuring that they stay updated on changes in the project.

- \*\*post-receive\*\*: This server-side hook runs after the push has been received by the server. It is often used for pushing code to production environments, triggering deployment scripts, or initiating continuous integration (CI) processes once the code is pushed to the repository.

\*\*How to Use Git Hooks\*\*: To use Git hooks, you need to create or modify the relevant script in the `.git/hooks` directory. Each hook is represented by a sample file (e.g., `pre-commit.sample`), which can be renamed and edited to implement the desired functionality. You can also install third-party hook managers like `Husky` to simplify configuration and management.

34. What is GitHub Administration? Repository permissions, team permissions, organization permissions

-------------------------------------------------------------------

GitHub administration involves managing repositories, users, and teams within an organization.

Repository permissions define who can read, write, or administer a repository.

Team permissions define the level of access a team has within a repository, such as read, write, or admin rights.

Organization permissions manage access to all repositories in the organization.

Admins can grant or revoke these permissions to ensure proper access control and collaboration.

Best practices include using least privilege and managing access carefully to avoid security risks.

35. git diff

---------------------

The `git diff` command shows the differences between various versions of a file or between the working directory and the staging area. By default, `git diff` compares changes that have been made but not yet staged. You can use `git diff <commit-id>` to compare a specific commit to the current state or between two commits. It's a helpful command to review changes before staging or committing them.

36. git logging and auditing

------------------------------------------

Git logging and auditing refer to the process of reviewing the commit history of a repository to understand who made changes and why.

The `git log` command is used to view the commit history, and options like `--oneline`, `--graph`, or `--author=<name>` can filter or format the output.

Git auditing involves checking for specific changes, tracking commits, and reviewing metadata like authorship and commit messages. This can be useful for security or compliance auditing.

37. git lfs, installation, select files, lock unlock, including/excluding

--------------------------------------------------------------

Git LFS (Large File Storage) is an extension for Git that allows you to manage large files more efficiently by replacing them with pointers in the repository while storing the actual files outside of the repository.

To install Git LFS, you can run `git lfs install`.

To track specific files with Git LFS, use `git lfs track <file-pattern>`.

You can include or exclude files by modifying the `.gitattributes` file.

Git LFS also provides commands to lock (`git lfs lock <file>`) and unlock (`git lfs unlock <file>`) files, preventing others from modifying the files while you are working on them.

Git Practical->

  
  
