FullStack.Cafe - Kill Your Tech Interview

Q1: What is Git? ☆

Topics: Git

Answer:

Git is a **Distributed Version Control system (DVCS)**. It can track changes to a file and allows you to revert back to any particular change.

Its distributed architecture provides many advantages over other Version Control Systems (VCS) like SVN one major advantage is that it does not rely on a central server to store all the versions of a project's files.

Q2: What is difference between Git vs SVN? ☆

Topics: Git

Answer:

The main point in Git vs SVN debate boils down to this: Git is a distributed version control system (DVCS), whereas SVN is a centralized version control system.

Q3: What is the command to write a commit message in Git? ☆

Topics: Git

Answer:

Use:

```
git commit -a
```

-a on the command line instructs git to commit the new content of all tracked files that have been modified. You can use

```
git add <file>
```

or

```
git add -A
```

before git commit -a if new files need to be committed for the first time.

Q4: What is Git fork? What is difference between fork, branch and clone? ☆☆

Topics: Git

Answer:

- A **fork** is a remote, server-side copy of a repository, distinct from the original. A fork isn't a Git concept really, it's more a political/social idea.
- A **clone** is not a fork; a clone is a local copy of some remote repository. When you clone, you are actually copying the entire source repository, including all the history and branches.
- A **branch** is a mechanism to handle the changes within a single repository in order to eventually merge them with the rest of code. A branch is something that is within a repository. Conceptually, it represents a thread of development.

Q5: What is the difference between `git pull` and `git fetch`? ☆☆

Topics: Git

Answer:

In the simplest terms, `git pull` does a `git fetch` followed by a `git merge`.

- When you use `pull`, Git tries to automatically do your work for you. **It is context sensitive**, so Git will merge any pulled commits into the branch you are currently working in. `pull` **automatically merges the commits without letting you review them first**. If you don't closely manage your branches, you may run into frequent conflicts.

- When you `fetch`, Git gathers any commits from the target branch that do not exist in your current branch and **stores them in your local repository**. However, **it does not merge them with your current branch**. This is particularly useful if you need to keep your repository up to date, but are working on something that might break if you update your files. To integrate the commits into your master branch, you use `merge`.

Q6: What's the difference between a `pull request` and a `branch`? ☆☆

Topics: Git

Answer:

- A **branch** is just a separate version of the code.

- A **pull request** is when someone take the repository, makes their own branch, does some changes, then tries to merge that branch in (put their changes in the other person's code repository).

Q7: How does the Centralized Workflow work? ☆☆

Topics: Git

Answer:

The **Centralized Workflow** uses a central repository to serve as the single point-of-entry for all changes to the project. The default development branch is called master and all changes are committed into this branch.

Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes. These new commits are stored locally.

To publish changes to the official project, developers *push* their local master branch to the central repository. Before the developer can publish their feature, they need to *fetch* the updated central commits and rebase their changes on top of them.

Compared to other workflows, the Centralized Workflow has no defined pull request or forking patterns.

Q8: What is `git cherry-pick`? ☆☆☆

Topics: Git

Answer:

The command git *cherry-pick* is typically used to introduce particular commits from one branch within a repository onto a different branch. A common use is to forward- or back-port commits from a maintenance branch to a development branch.

This is in contrast with other ways such as merge and rebase which normally apply many commits onto another branch.

Consider:

```
git cherry-pick <commit-hash>
```

Q9: How to revert previous commit in git? ☆☆☆

Topics: Git

Answer:

Say you have this, where C is your HEAD and (F) is the state of your files.

```
   (F)
A-B-C
    ↑
  master
```

1. To nuke changes in the commit:

   ```
   git reset --hard HEAD~1
   ```

   Now B is the HEAD. Because you used --hard, your files are reset to their state at commit B.
2. To undo the commit but keep your changes:

   ```
   git reset HEAD~1
   ```

   Now we tell Git to move the HEAD pointer back one commit (B) and leave the files as they are and `git status` shows the changes you had checked into C.
3. To undo your commit but leave your files and your index

```
git reset --soft HEAD~1
```

When you do `git status`, you'll see that the same files are in the index as before.

Q10: Tell me the difference between HEAD, working tree and index, in Git? ☆☆☆

Topics: Git

Answer:

- The **working tree/working directory/workspace** is the directory tree of (source) files that you see and edit.
- The **index/staging area** is a single, large, binary file in /.git/index, which lists all files in the current branch, their sha1 checksums, time stamps and the file name - it is not another directory with a copy of files in it.
- **HEAD** is a reference to the last commit in the currently checked-out branch.

Q11: When should I use `git stash`? ☆☆☆

Topics: Git

Answer:

The `git stash` command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

Consider:

```
$ 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: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage
$ git status
On branch master
nothing to commit, working tree clean
```

The one place we could use stashing is if we discover we forgot something in our last commit and have already started working on the next one in the same branch:

```
# Assume the latest commit was already done
# start working on the next patch, and discovered I was missing something

# stash away the current mess I made
$ git stash save

# some changes in the working dir

# and now add them to the last commit:
$ git add -u
$ git commit --ammend
```

```
# back to work!
$ git stash pop
```

Q12: Could you explain the Gitflow workflow? ☆☆☆

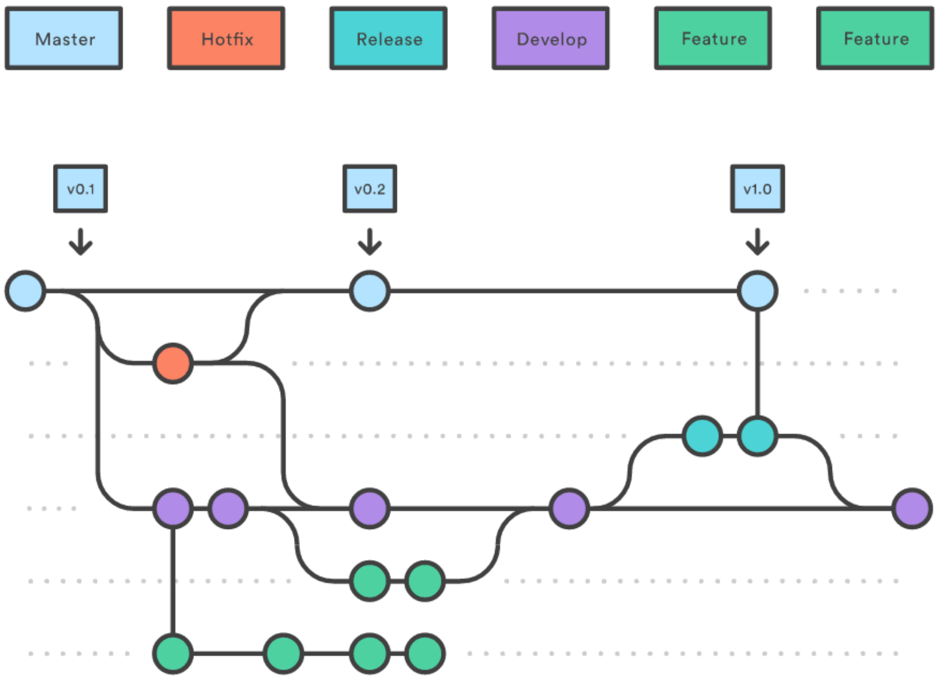
Topics: Git

Answer:

Gitflow workflow employs two parallel *long-running* branches to record the history of the project, `master` and `develop`:

- **Master** - is always ready to be released on LIVE, with everything fully tested and approved (production-ready).
- **Hotfix** - Maintenance or "hotfix" branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on `master` instead of `develop`.

- **Develop** - is the branch to which all feature branches are merged and where all tests are performed. Only when everything's been thoroughly checked and fixed it can be merged to the `master`.
- **Feature** - Each new feature should reside in its own branch, which can be pushed to the `develop` branch as their parent one.



Q13: Explain the advantages of Forking Workflow ☆☆☆

Topics: Git

Answer:

The **Forking Workflow** is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the "central" codebase, it gives every developer their own server-side repository. The Forking Workflow is most often seen in public open source projects.

The *main advantage* of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository that leads to a clean project history. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository.

When developers are ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated.

Q14: What is a "bare git" repository? ☆☆☆

Topics: Git

Answer:

- It is a repository that is created without a *working tree*.
- To initializes the bare repository.

  ```
  git init --bare .
  ```

- It is typically used as a *remote repository* that is sharing a repository among several different people.
- It only contains the *repository data*(refs, commits) and nothing else.

Q15: What is difference between `git stash pop` and `git stash apply`? ☆☆☆☆

Topics: Git

Answer:

`git stash pop` **throws away** the (topmost, by default) stash after applying it, whereas `git stash apply` **leaves it in the stash list** for possible later reuse (or you can then `git stash drop` it).

This happens unless there are conflicts after `git stash pop` (say your stashed changes conflict with other changes that you've made since you first created the stash), in this case, it will not remove the stash, behaving exactly like `git stash apply`.

Another way to look at it: `git stash pop` is `git stash apply && git stash drop`.

Q16: What is the `HEAD` in Git? ☆☆☆☆

Topics: Git

Answer:

`HEAD` is a ref (reference) to the currently checked out commit.

In normal states, it's actually a symbolic ref to the branch you have checked out - if you look at the contents of .git/HEAD you'll see something like "ref: refs/heads/master". The branch itself is a reference to the commit at the tip of the branch. Therefore, in the normal state, HEAD effectively refers to the commit at the tip of the current branch.

It's also possible to have a `detached HEAD`. This happens when you check out something besides a (local) branch, like a remote branch, a specific commit, or a tag. The most common place to see this is during an interactive rebase, when you choose to edit a commit. In detached HEAD state, your HEAD is a direct reference to a commit - the contents of .git/HEAD will be a SHA1 hash.

Generally speaking, HEAD is just a convenient name to mean "what you have checked out" and you don't really have to worry much about it. Just be aware of what you have checked out, and remember that you probably don't want to commit if you're not on a branch (detached HEAD state) unless you know what you're doing (e.g. are in an interactive rebase).

Q17: Can you explain what `git reset` does in plain English? ☆☆☆☆

Topics: Git

Answer:

In general, `git reset` function is to take the current branch and reset it to point somewhere else, and possibly bring the index and work tree along.

```
- A - B - C (HEAD, master)
# after git reset B (--mixed by default)
- A - B (HEAD, master)      # - C is still here (working tree didn't change state), but there's no branch
pointing to it anymore
```

Remeber that in git you have:

- the HEAD pointer, which tells you what commit you're working on
- the working tree, which represents the state of the files on your system
- the staging area (also called the index), which "stages" changes so that they can later be committed together

So consider:

- `git reset --soft` moves HEAD but doesn't touch the staging area or the working tree.
- `git reset --mixed` moves HEAD and updates the staging area, but not the working tree.
- `git reset --merge` moves HEAD, resets the staging area, and tries to move all the changes in your working tree into the new working tree.
- `git reset --hard` moves HEAD and adjusts your staging area and working tree to the new HEAD, throwing away everything.

Use cases:

- Use `--soft` when you want to move to another commit and patch things up without "losing your place". It's pretty rare that you need this.
- Use `--mixed` (which is the default) when you want to see what things look like at another commit, but you don't want to lose any changes you already have.
- Use `--merge` when you want to move to a new spot but incorporate the changes you already have into that the working tree.
- Use `--hard` to wipe everything out and start a fresh slate at the new commit.

Q18: How do you make an existing repository bare? ☆☆☆☆

Topics: Git

Answer:

Run the command below:

```
git clone --mirror <repo_source_path>
```

The `--mirror` flag maps all *refs* (including remote-tracking branches, notes etc.) and sets up a refspec configuration such that all these refs are overwritten by a `git remote update` in the target repository.

Q19: What is the difference between `git clone`, `git clone --bare` and `git clone --mirror`? ☆☆☆☆

Topics: Git

Answer:

- `git clone origin-url` is primarily used to point to an existing repo and make a clone or copy of that repo at in a new directory. It has its own `history`, manages its own `files`, and is a completely `isolated` environment from the original repository.
- `git clone --base origin-url` makes a copy of the remote repository with an `omitted` working directory. Also the branch `heads` at the remote are copied directly to corresponding local branch heads, without `mapping`. Neither remote-tracking branches nor the related configuration variables are created.
- `git clone --mirror origin-url` will clone all the extended `refs` of the remote repository, and `maintain` remote branch tracking configuration. All `local references` (including `remote-tracking branches`, `notes` etc.) will be overwritten each time you fetch, so it will always be the same as the original repository.

Q20: When would you use `git clone` over `git clone --mirror`? ☆☆☆☆

Topics: Git

Answer:

Suppose origin has a few branches (`master (HEAD)`, `next`, `pu`, and `maint`), some tags (`v1`, `v2`, `v3`), some remote branches (`devA/master`, `devB/master`), and some other refs (`refs/foo/bar`, `refs/foo/baz`, which might be notes, stashes, other devs' namespaces, who knows).

- **`git clone` (non-bare):** You will get all of the tags copied, a local branch `master (HEAD)` tracking a remote branch `origin/master`, and remote branches `origin/next`, `origin/pu`, and `origin/maint`. The tracking branches are set up so that if you do something like `git fetch origin`, they'll be fetched as you expect. Any remote branches (in the cloned remote) and other refs are completely ignored.

- **`git clone --mirror`:** Every last one of those refs will be copied as-is. You'll get all the tags, local branches `master (HEAD)`, `next`, `pu`, and `maint`, remote branches `devA/master` and `devB/master`, other refs `refs/foo/bar` and `refs/foo/baz`. Everything is exactly as it was in the cloned remote. Remote tracking is set up so that if you run `git remote update` all refs will be overwritten from origin, as if you'd just deleted the mirror and recloned it. As the docs originally said, it's a mirror. It's supposed to be a functionally identical copy, interchangeable with the original.

To summarize,

- `git clone` is used when we need to point to an existing repo and make a clone or copy of that repo at in a new directory.
- `git clone --mirror` is used to clone all the extended `refs` of the remote repository, and `maintain` remote branch tracking configuration.

FullStack.Cafe - Kill Your Tech Interview

Q1: You need to update your local repos. What git commands will you use? ☆☆

Topics: Git

Answer:

It’s a two steps process. First you fetch the changes from a remote named origin:

```
git fetch origin
```

Then you merge a branch master to local:

```
git merge origin/master
```

Or simply:

```
git pull origin master
```

If origin is a default remote and ‘master’ is default branch, you can drop it eg. `git pull`.

Q2: How to undo the most recent commits in Git? ☆☆

Topics: Git

Problem:

You accidentally committed wrong files to Git, but haven't pushed the commit to the server yet. How can you undo those commits from the local repository?

Solution:

```
$ git commit -m "Something terribly misguided"
$ git reset HEAD~                                  # copied the old head to .git/ORIG_HEAD
<< edit files as necessary >>
$ git add ...
$ git commit -c ORIG_HEAD                          # will open an editor, which initially contains the
log message from the old commit and allows you to edit it
```

Q3: You need to rollback to a previous commit and don't care about recent changes. What commands should you use? ☆☆☆

Topics: Git

Answer:

Let's say you have made mulitple commits, but the last few were bad and you want to rollback to a previous commit:

```
git log // lists the commits made in that repository in reverse chronological order
git reset --hard <commit-sha1> // resets the index and working tree
```

Q4: Write down a sequence of git commands for a "Rebase Workflow" ☆☆☆☆

Topics: Git

Answer:

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. Create a feature branch
   ```
   $ git checkout -b feature
   ```
2. Make changes on the feature branch
   ```
   $ echo "Bam!" >>foo.md
   $ git add foo.md
   $ git commit -m 'Added awesome comment'
   ```
3. Fetch upstream repository
   ```
   $ git fetch upstream
   ```
4. Rebase changes from feature branch onto upstream/master
   ```
   $ git rebase upstream/master
   ```
5. Rebase local master onto feature branch
   ```
   $ git checkout master
   $ git rebase feature
   ```
6. Push local master to upstream
   ```
   $ git push upstream master
   ```

Rebasing gives you a clean linear commit history and creates non-obvious benefits to your project if used diligently. Think of it as taking a line of work and pretending it always started at the very latest revision.

Q5: How to remove a file from git without removing it from your file system? ☆☆☆☆

Topics: Git

Answer:

If you are not careful during a `git add`, you may end up adding files that you didn’t want to commit. However, `git rm` will remove it from both your staging area (index), as well as your file system (working tree), which may not be what you want.

Instead use `git reset`:

```
git reset filename          # or
echo filename >> .gitingore # add it to .gitignore to avoid re-adding it
```

This means that `git reset <paths>` is the opposite of `git add <paths>`.

Q6: When would you use `git clone` over `git clone --bare`? ☆☆☆☆

Topics: Git

Answer:

Suppose origin has a few branches (`master (HEAD)`, `next`, `pu`, and `maint`), some tags (`v1`, `v2`, `v3`), some remote branches (`devA/master`, `devB/master`), and some other refs (`refs/foo/bar`, `refs/foo/baz`, which might be notes, stashes, other devs' namespaces, who knows).

- **`git clone` (non-bare):** You will get all of the tags copied, a local branch `master (HEAD)` tracking a remote branch `origin/master`, and remote branches `origin/next`, `origin/pu`, and `origin/maint`. The tracking branches are set up so that if you do something like `git fetch origin`, they'll be fetched as you expect. Any remote branches (in the cloned remote) and other refs are completely ignored.

- **`git clone --bare`:** You will get all of the tags copied, local branches `master (HEAD)`, `next`, `pu`, and `maint`, no remote tracking branches. That is, all branches are copied as is, and it's set up completely independent, with no expectation of fetching again. Any remote branches (in the cloned remote) and other refs are completely ignored.

To summarize,

- `git clone` is used when we need to point to an existing repo and make a clone or copy of that repo at in a new directory.
- `git clone --bare` is used to make a copy of the remote repository with an *omitted working directory*.

Q7: When would you use `git clone --bare` over `git clone --mirror`? ☆☆☆☆

Topics: Git

Answer:

Suppose origin has a few branches (`master (HEAD)`, `next`, `pu`, and `maint`), some tags (`v1`, `v2`, `v3`), some remote branches (`devA/master`, `devB/master`), and some other refs (`refs/foo/bar`, `refs/foo/baz`, which might be notes, stashes, other devs' namespaces, who knows).

- **`git clone --bare`:** You will get all of the tags copied, local branches `master (HEAD)`, `next`, `pu`, and `maint`, no remote tracking branches. That is, all branches are copied as is, and it's set up completely independent, with no expectation of fetching again. Any remote branches (in the cloned remote) and other refs are completely ignored.

- **`git clone --mirror`:** Every last one of those refs will be copied as-is. You'll get all the tags, local branches `master (HEAD)`, `next`, `pu`, and `maint`, remote branches `devA/master` and `devB/master`, other refs `refs/foo/bar` and `refs/foo/baz`. Everything is exactly as it was in the cloned remote. Remote tracking is set up so that if you run `git remote update` all refs will be overwritten from origin, as if you'd just deleted the mirror and recloned it. As the docs originally said, it's a mirror. It's supposed to be a functionally identical copy, interchangeable with the original.

To summarize,

- `git clone --bare` is used to make a copy of the remote repository with an *omitted working directory*.
- `git clone --mirror` is used to clone all the extended `refs` of the remote repository, and `maintain` remote branch tracking configuration.

Q8: When do you use `git rebase` instead of `git merge`? ☆☆☆☆☆

Topics: Git

Answer:

Both of these commands are designed to integrate changes from one branch into another branch - they just do it in very different ways.

Consider before merge/rebase:

```
A <- B <- C    [master]
^
 \
  D <- E       [branch]
```

after `git merge master`:

```
A <- B <- C
^         ^
 \         \
  D <- E <- F
```

after `git rebase master`:

```
A <- B <- C <- D <- E
```

With rebase you say to use another branch as the new base for your work.

When to use:

1. If you have any doubt, use merge.
2. The choice for rebase or merge based on what you want your history to look like.

More factors to consider:

1. **Is the branch you are getting changes from shared with other developers outside your team (e.g. open source, public)?** If so, don't rebase. Rebase destroys the branch and those developers will have

broken/inconsistent repositories unless they use `git pull --rebase`.
2. **How skilled is your development team?** Rebase is a destructive operation. That means, if you do not apply it correctly, you could lose committed work and/or break the consistency of other developer's repositories.
3. **Does the branch itself represent useful information?** Some teams use the *branch-per-feature* model where each branch represents a feature (or bugfix, or sub-feature, etc.) In this model the branch helps identify sets of related commits. In case of *branch-per-developer* model the branch itself doesn't convey any additional information (the commit already has the author). There would be no harm in rebasing.
4. **Might you want to revert the merge for any reason?** Reverting (as in undoing) a rebase is considerably difficult and/or impossible (if the rebase had conflicts) compared to reverting a merge. If you think there is a chance you will want to revert then use merge.

Q9: What git command do you need to use to know who changed certain lines in a specific file? ☆☆☆☆☆

Topics: Git

Answer:

Use `git blame` - a little feature in git that allows you to see who wrote what in the repository. If you want to know who changed certain lines, you can use the -L flag to figure out who changed those lines. You can use the command:

```
git blame -L <line-number>,<ending-linenumber> <filename>
```

Q10: Write down a git command to check difference between two commits ☆☆☆☆☆

Topics: Git

Answer:

`git diff` allows you to check the differences between the branches or commits. If you type it out automatically, you can checkout the differences between your last commit and the current changes that you have.

```
git diff <branch_or_commit_name> <second_branch_or_commit>
```

Q11: How to amend older Git commit? ☆☆☆☆☆

Topics: Git

Problem:

You have made 3 git commits, but have not been pushed. How can you amend the older one (ddc6859af44) and (47175e84c) which is not the most recent one?

```
$git log
commit f4074f289b8a49250b15a4f25ca4b46017454781
Date:   Tue Jan 10 10:57:27 2012 -0800

commit ddc6859af448b8fd2e86dd0437c47b6014380a7f
```

```
Date:   Mon Jan 9 16:29:30 2012 -0800

commit 47175e84c2cb7e47520f7dde824718eae3624550
Date:   Mon Jan 9 13:13:22 2012 -0800
```

Solution:

```
git rebase -i HEAD^^^
```

Now mark the ones you want to amend with `edit` or `e` (replace `pick`). Now save and exit.

Now make your changes, then:

```
git add -A
git commit --amend --no-edit
git rebase --continue
```

If you want to add an extra delete remove the options from the commit command. If you want to adjust the message, omit just the `--no-edit` option.

Q12: Do you know how to easily undo a git rebase? ☆☆☆☆☆

Topics: Git

Answer:

The easiest way would be to find the head commit of the branch as it was immediately before the rebase started in the `reflog`

```
git reflog
```

and to reset the current branch to it (with the usual caveats about being absolutely sure before reseting with the `--hard` option).

Suppose the old commit was `HEAD@{5}` in the ref log:

```
git reset --hard HEAD@{5}
```

Also rebase saves your starting point to `ORIG_HEAD` so this is usually as simple as:

```
git reset --hard ORIG_HEAD
```

Q13: What are "git hooks"? ☆☆☆☆☆

Topics: Git

Answer:

- **Git hooks** are scripts that Git executes before or after events such as: *commit*, *push*, and *receive*.
- By default the hooks directory is `.git/hooks`, but that can be changed via the `core.hooksPath` configuration variable.
- Any scripting language that can be run as an executable can be used to make hooks.
- Hooks are local to any given Git repository, and they are not copied over to the new repository when `git clone` is run.
- Some of the hooks are:
  - pre-commit
  - post-commit
  - post-checkout
  - pre-push
  - update

Q14: What are the type of git hooks? ☆☆☆☆☆

Topics: Git

Answer:

Git hooks can be categorized into *two* main types:

- Client-Side Hooks
- Server-Side Hooks

Client-Side Hooks:

- These are hooks installed and maintained on the developers local repository and are executed when events on the local repository are triggered.
- They are also known as **local hooks**.
- They cannot be used as a way to enforce universal commit policies on a remote repository as each developer can alter their hooks.
- Some client-side hooks are:
  i. pre-commit
  ii. prepare-commit-msg
  iii. commit-msg
  iv. post-commit
  v. post-checkout
  vi. pre-rebase
- From the list above, the first 4 hooks are executed from *top* to *down hierarchy*. The last 2 hooks allows to perform some extra actions or safety checks for `git checkout` and `git rebase` commands.
- All of the `pre-` hooks let you alter the action that’s about to take place, while the `post-` hooks are used only for notifications.

Server-Side Hooks:

- These are hooks that are executed in a remote repository on the triggering of certain events.
- These hooks can act as a system administrator to enforce nearly any kind of policy for a project like rejecting a commit based on some rule.
- The server-side hooks are listed below based on the execution order:
  i. pre-receive
  ii. update
  iii. post-receive
- The *output* from server-side hooks are *piped* to the client’s console, so it’s very easy to send messages back to the developer.

Q15: What is `git bisect`? How can you use it to determine the source of a (regression) bug? ☆☆☆☆☆

Topics: Git

Answer:

- `git bisect` command is used to find the commit that has introduced a bug by using binary search.
- The command is:

  ```
  git bisect <subcommand> <options>
  ```

- This command uses a binary search algorithm to find which commit in your project's history introduced a bug.
- You use it by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then `git bisect` picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the change.

Q16: How can you use `git bisect` to determine the source of a (regression) bug? ☆☆☆☆☆

Topics: Git

Answer:

Suppose you are trying to find the commit that broke a feature that was known to work in previous versions. You start a **bisect** session as follows:

- First, *commit* or *stash* you unfinished work.
- Initialize the bisect with:

  ```
  git bisect start
  ```

- Next, mark your commits with *good* or *bad* labels by either specifying the commit SHA or checking out to the commit.

```
git checkout 1234567
git bisect good
```

OR

```
git bisect good 1234567
```

- Then it will respond with

  ```
  Bisecting: 337 revisions left to test after this (roughly 9 steps)
  ```

- Keep repeating the process: compile the tree, test it, and depending on whether it is good or bad run `git bisect good` or `git bisect bad` to ask for the next commit that needs testing.

- Eventually there will be no more revisions left to inspect, and the command will print out a description of the first bad commit. The reference `refs/bisect/bad` will be left pointing at that commit.
- After a bisect session, to clean up the bisection state and return to the original HEAD, issue the following command:

  ```
  git bisect reset
  ```