GIT

INTERVIEW QUESTIONS & ANSWERS

PART 2

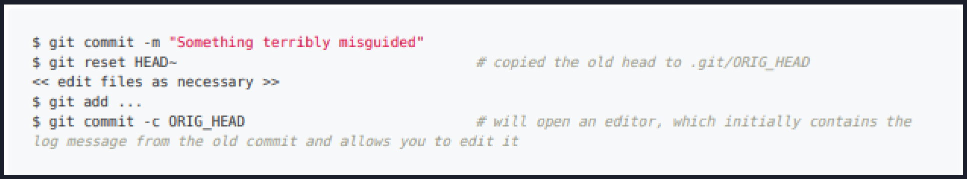
FAQ

1) How to undo the most recent commits in 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?

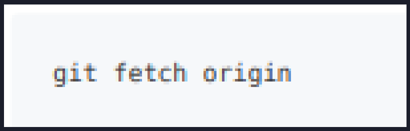
Solution:



```
$ 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
```

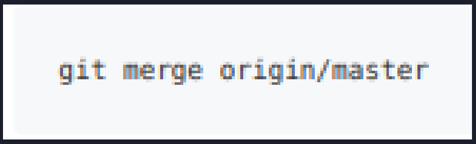
2) You need to update your local repos. What git commands will you use?

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



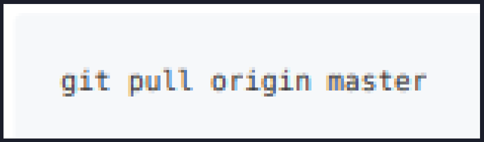
```
git fetch origin
```

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 .

FAQ

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

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
```

4) Write down a sequence of git commands for a "Rebase Workflow"

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
```

FAQ

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.

5) How to remove a file from git without removing it from your file system?

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 >> .gitignore # add it to .gitignore to avoid re-adding it
```

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

FAQ

6) When would you use "git clone --bare" over "git clone --mirror" ?

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.

FAQ

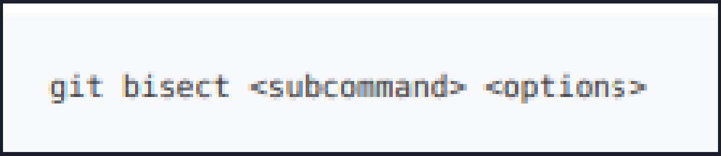
7) What is the difference between "git clone", "git clone --bare" and "git clone --mirror"?

- 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.

8) What is "git bisect"? How can you use it to determine the source of a (regression) bug?

- 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.

FAQ

9) How can you use "git bisect" to determine the source of a (regression) bug?

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

- 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
```

FAQ

10) What are the type of git hooks?

Git hooks can be categorized into two main types:

- Client-Side Hooks
- Server-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.

FAQ

11) Do you know how to easily undo a git rebase?

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
```

12) How to amend older Git commit?

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.

13) What are "git hooks"?

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

FAQ

14) Write down a git command to check difference between two commits

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>
```

15) What git command do you need to use to know who changed certain lines in a specific file?

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>
```

16)When do you use "git rebase" instead of "git merge"?

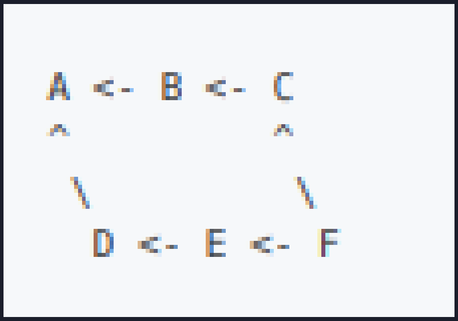
Both of these commands are designed to integrate changes from one branch into another branch – they just do it in very different ways.

Consider before merge/rebase:

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

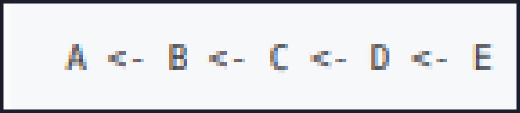
FAQ

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.