



Advanced GIT



Advanced features

- git cherry-pick
- git stash
- git bisect
- Partial commits
- git filter-branch
- git rebase
- git rerere
- git pickaxe
- git submodule



stash

- Save changes to your working copy
WITHOUT committing them
 - `git stash`



stash

- There can be multiple stashes, each with a name.
 - `git stash list`
- You can apply a stash other than the last one.
 - `git stash apply <stash name>`
- Applying a stash does not delete it.
 - `git stash drop <stash name>`

git cherry-pick

- The ability to get a specific commit from a branch and merge *only that commit* on a different branch.
 - Also known as: “backport fix for bug ##### to an older branch.”
 - This is a merge operation. Conflicts may occur and will have to be resolved normally

git cherry-pick

■ Usage:

- `git cherry-pick <commit>`

- Will merge commit `<commit>` on the current branch.

- If you are merging from a public branch, add “-x”, i.e.:

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

- Reason: Will add a note to the commit message specifying the source of the cherry-pick

git pickaxe

- Also known as `git log -S` or `git log -G`
 - When did you change something?
 - Search the whole history of git to find out!
- `git log -S<string>`
 - Shows commits where the number of occurrences of `<string>` changed
- `git log -G<regex>`
 - Shows commit where there are additions/deletions of the current regex.




Partial commits

- Many modifications?
- Logically different?
- Split them into different commits!
- `git add -p <file>`
 - Allows to edit what exactly gets staged
- Example




git bisect

- Also known as “automated debug.”
- When and How did I break feature X?
- Suppose you introduced a bug somewhere in your code.
- If you can detect a commit which clearly HAS bug, and another which HAS NOT the bug, then git can tell you exactly when it was introduced.



git bisect -- usage

- `git bisect start`
- `git bisect bad <commit>`
 - This commit has the bug
- `git bisect good <commit2>`
 - This commit does not have the bug
- From now on, git starts a binary search on all commits included between the two, to discover where the bug happened



git bisect -- usage

- Git will have checked out a revision.
- Test it.
 - ☐ Does it have the bug?
 - YES: git bisect bad
 - NO: git bisect good
- Another checkout will be done. Iterate
- Eventually, you will have reached a single commit.
 - ☐ That commit will have introduced the bug

git bisect – automated usage

- Not very convenient – You have to manually test each one.
- However, if you can script the test somehow...
 - make a script
 - It should return:
 - 0 – no bug
 - 125 – cannot be tested
 - 1-124, 126-127 – bug present
 - The do:
 - `git bisect run <script>`
 - When this command returns, the commit that introduced the bug will be checked out.



git bisect – automated usage

- Note that builds failing may muddle this
 - If the commit which introduced the bug was a build failure, you would get the first commit after that that actually builds as a result
 - Hence why: Rebase usage case 1.



Reminder

- In git, the name of a “branch” is in reality an alias to the LAST commit on that “branch”
 - Follow the EXPERT version of this course to see why branch is between quotation marks.
 - There is no EXPERT version yet.

git rerere

■ Rerere

- REuse REcorded Resolutions

- Allows git to record how you solved a particular conflict, and to resolve it automatically from then on.

- But why should I be interested? I solved it, after all.



rerere – use cases

- You have several topic branches that you want to occasionally merge in a single one to test it.
 - It will have conflicts.
 - You will have to resolve them
 - But in case of failure, you want to blow away the merge commit(s)
 - Without rerere, this means that you will have to re-resolve the same conflicts the next time

rerere -- usecases

- Want to make sure that you can merge cleanly with another branch.
 - Without rerere:
 - Periodically do a 'git merge branch'
 - Resolve all the conflicts
 - There may be many many many many of them.
 - git commit
 - With rerere:
 - Periodically do a 'git merge branch'
 - Resolve few conflicts
 - Blow away the merge commit
 - Rerere will keep track of your resolutions, and reapply them in the next merge
 - At the end: do a 'git merge branch'
 - Resolve only the latest conflicts
 - git commit



rerere – hot to use it

- rerere usage is automatic when you do a merge
 - You just need to activate it
 - `git config --global rerere.enabled 1`



git rebase

- The main topic of this part
- git rebase allows you to rewrite your history.
 - It alters the repository so that the commits you can see before and after its usage are different.
 - It can change, merge, split, add, remove, modify commits



git rebase

- IT IS A DANGEROUS COMMAND!

- If you change history, you will break merges for EVERYONE that has already 'pulled' your branch.
- git tries to protect you from it
 - If it detects that pushing will probably cause it, it tries to stop you.
 - git push will fail
 - You will have to use a different syntax to go ahead anyway.
 - However, you should not rely on this

git rebase

- Simple ground rules:

- ☐ No commit should EVER be rebased if it is already public.
 - If you have already “pushed” it, or
 - If you have “pulled” or “fetchd” it, or
 - These include commits inherited from branches created from “pushed” or “pulled” ones.
- ☐ No commit should be rebase if:
 - You have merged it on a different branch
 - ☐ This latter will not break other users, but it **will ** break your local merges and rerere.

- A suggestion:

- ☐ Even if you can use it, do not go overboard.
- ☐ Only use it on private branches you have not merged anywhere.



git rebase

- Two main usages:
 - ☐ Batch
 - ☐ Interactive

git rebase – batch usage

- Takes a branch, and modify it to make it look like the branch never existed, and the dev. Was done directly off another branch.
- Example:
 - A-B-C-D master
 - \ -E-F-G topic
- Becomes:
 - A-B-C-D-E'-F'-G' master
- From 'topic' branch: 'git rebase master'

git rebase – batch usage

- git rebase does an actual merge
 - Merges may have conflict. You have three choices
 - Solve the conflicts:
 - git add the resolutions
 - git rebase –continue
 - Skip this commit
 - git rebase –skip
 - Abort the rebase
 - git rebase --abort

git rebase – batch usage

- You can also change the tree structure of your repository.
 - E.g: you have master, branch A forked from master, branch B forked from A
 - After: git rebase –onto master A B
 - Now, branch B is forked from the HEAD of master
 - | | | | | |
|-------|--------|----|---------|--------|
| a-b-c | master | | a-b-c | master |
| \-d-e | A | -> | \-f'-g' | B |
| \-f-g | B | | \-d-e | A |

git rebase – batch commands

- Finally, you can delete commits:
 - A-B-C-D-E-F master
- `git rebase –onto master~5 master~2`
master
 - A-E'-F' master

git rebase – interactive commands

- `git rebase -i <commit>`
 - `<commit>` should be the commit BEFORE the first one you wish to alter.
 - Will open your `$EDITOR` with the following buffer:

git rebase – interactive usage

```
pick d6a7c25 Added README file.  
pick dce0696 changed README file.  
pick 8e5ba04 Makefile  
pick b42cffd rgheguie  
pick 2306a37 new line.
```

```
# Rebase 68bcfea..2306a37 onto 68bcfea  
#  
# Commands:  
# p, pick = use commit  
# r, reword = use commit, but edit the commit message  
# e, edit = use commit, but stop for amending  
# s, squash = use commit, but meld into previous commit  
# f, fixup = like "squash", but discard this commit's log message  
# x, exec = run command (the rest of the line) using shell  
# d, drop = remove commit  
# These lines can be re-ordered; they are executed from top to bottom  
# If you remove a line here THAT COMMIT WILL BE LOST.  
# However, if you remove everything, the rebase will be aborted.  
# Note that empty commits are commented out
```



git rebase – interactive usage

- If you do nothing, or remove all the lines, nothing happens
- squash – This commit gets deleted, but its contents are added to the previous commit. Commit messages are merged.
- fixup – Like squash, but the commit message gets lost



git rebase – interactive usage

- reword – change the commit message
- edit – stop there to allow modifying the commit
- edit is the more interesting here:

git rebase – interactive usage

- When the rebasing arrives at an “edit” commit, you get the command line back.
- At that point, you can do whatever you wish:
 - Splitting the commit
 - `git reset HEAD^`
 - `git add file1 ; git commit ; git add file2 ; git commit`
 - Adding files
 - `Git add file3 ; git commit`
 - Etc...
- Afterwards:
 - `git rebase --continue`

git rebase -- reminder

- Simple ground rules:

- ☐ No commit should EVER be rebased if it is already public.
 - If you have already “pushed” it, or
 - If you have “pulled” or “fetchd” it, or
 - These include commits inherited from branches created from “pushed” or “pulled” ones.
- ☐ No commit should ever be rebased if:
 - You have merged it on a different branch
 - ☐ This latter will not break other users, but it *will* break your local merges and rerere.


- A suggestion:

- ☐ Even if you can use it, do not go overboard.
- ☐ Only use it on private branches you have not merged anywhere.




git rebase – why and when

- So, when and how should I do it?
 - I have found only two practical usages.



git rebase - usage 1

- On a topic branch:
 - Commit early and often.
 - Who cares if it does not build? It's private
 - Keep track of your work
 - Before merging on a release branch, rebase as to make sure that all commits at least build.
 - Why? Because otherwise 'git bisect' breaks.



git rebase - usage 2

- On a branch you wished 'pulled' by someone else
 - Do a git rebase to ensure clean pulling
 - DO NOT CHANGE COMMITS THAT ARE ALREADY PUBLIC!



git submodule

- Suppose your software has dependencies.
- And that those dependencies are also held in git repositories.
- You can add informations about them to your program.

git submodule

■ How to initialize it:

- `git submodule add git://repo/path/to/project.git`
 - This creates a directory called 'project'
- `cd project; git checkout <version>`
 - Check out the exact version you wish to use
- `git commit ; git push`

git submodule

- When cloning:

- git clone <main repo>

- As usual

- git submodule init ; git submodule update

- Updates the submodules to the correct checkout version as set by the origin repository.

- This **will** require connectivity to the submodule repositories.

- To DELETE a submodule:

- git submodule deinit <dir>

git submodule -- notes

- Inside the submodules, by default you do not get a branch.
 - I.e: commits in there are lost by default
 - You should explicitly switch to a branch,
 - Commits inside it can be “pushed” to the repo.
 - But this WILL not change what gets checked out
 - Need commit (and push) on the main repo
- Whenever you do a ‘git pull’ you should also do a ‘git submodule update’
- ‘git submodule’ commands MUST be given from outside the submodule




git submodule

■ Cons:

- ☐ Needs extra work after clone
- ☐ 'git submodule update' destroys uncommitted changes
- ☐ Local changes cannot be kept in master repository.
- ☐ Local changes cannot be seen by remotes unless pushed to dependency repo.
- ☐ Standard way to undo things does not work

■ Pros:

- ☐ Present in standard distribution
- ☐ Clean separation between main sources and dependencies



git filter-branch

- Applies an operation to all commits on a branch (or all commits on a repo)

Delete content from history

- You committed a file that should not be there.
 - E.g: 'passwords'
 - That file should be completely removed from history. How?
- > `git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch passwords' --prune-empty --tag-name-filter cat -- --all`
 - > `git for-each-ref --format='delete %(refname)' refs/original | git update-ref --stdin`
 - > `git reflog expire --expire=now --all`
 - > `git gc --prune=now`



Git signing

- Two types of signatures:

- Informative

- On commits

- Cryptographical

- On tags



Signing commits

- `git commit -s`
 - Adds the “signed-off by:” line to the commit messages
 - Usefulness:
 - In case of merge done by other developers, keeps track of who originally wrote the code.
 - Especially important if patches are exchanged via email. Not so important with push/pull

Signing tags

■ `git tag -s`

- ☐ Signs the tag with your GPG signing key.
- ☐ This does not just fix the tag, but to all the commits that precede it in the chain.
- ☐ GPG key and keyring setup left as exercises to the reader.



Questions?

?