I'm not a robot

I had a small Git issue today, which used to be quite scary when I was first learning Git but is actually easy to fix as long as you are working locally. We've all done it - sometimes you forget you haven't created a branch yet, or worse yet you're on the wrong feature branch, and you start adding some code and make a commit. Then you realise you've committed code to the wrong feature branch or straight to master, and now it's in the wrong place. As long as you haven't pushed your changes up to origin, this is very easy to undo. Let's say we have made a commit a1b2c3d on the branch feature-a, and we haven't yet made a feature-b branch. So first of all we want to get our commit onto the right branch, so let's take a note of the hash want to move, and start at master. Optionally (if we don't have the branch yet) we make a new branch feature-b to put it on and check it out: git checkout -b feature-b Then make sure you are on the right branch feature-b (may not be necessary if you just created it), and cherry pick this commit into that branch to add just that commit to feature-b: git checkout feature-b git cherry-pick a1b2c3d. And finally let's reset feature-a branch back to the previous commit hash (say z1b2c3d). Using git reset --hard will remove all the commit referencing the changes, and all the changes themselves, from feature-a branch, while leaving that commit on feature-b: git checkout feature-a git reset --hard z1b2c3d. You can do this with multiple commits too, just cherry pick several, then reset back to the last commit you want to keep. The process is the same if you have committed to local master by mistake - just cherry-pick to a branch, then reset master. Only ever do this if you haven't pushed the commits to origin. When working with Git, there might be times when you need to move commits from one branch to another. This guide will walk you through the process of moving commits using different methods such as git cherry-pick. Moving the last commit to another branchStep 1: Create and switch to the target branchFirst, create the new branch (if it doesn't already exist) and switch to it:git checkout -b Replace with the name of the branch you want to create and move your commit to.Step 2: Cherry-pick the last commitNext, you can use git cherry-pick to apply the commit from the source branch to the target branch. If you're moving the last commit, you can run:git cherry-pick . To get the commit hash of the last commit, run:Copy the hash and use it in the cherry-pick command.Step 3: Remove the commit from the original branchSwitch back to the original branch:git checkout Then, reset the branch to remove the last commit:This command will reset the branch to the state before the last commit, effectively removing it. Moving multiple commits to another branchStep 1: Identify the commits to moveUse git log to identify the commit hashes you want to move:Step 2: Create and switch to the target branchCreate the new branch and switch to it:git checkout -b Step 3: Cherry-pick the commitsCherry-pick the commits one by one or use a range:git cherry-pick ...Or for a range of commits:git cherry-pick ^.. Step 4: Remove the commits from the original branchSwitch back to the original branch:git checkout Then, use interactive rebase to remove the specific commits:git rebase -i ^ In the interactive rebase screen, change the command for the commits you want to drop from pick to drop.Step 1: Create and switch to the target branch. Begin by establishing a new branch and switching to it with the command git checkout -b . Step 2: Rebase the source branch onto this new branch. Rebase the source branch onto this new branch by switching back and using git rebase . Step 3: Force push the changes (if necessary). If you are working on a remote repository and need to update the branches, you may need to force push the changes. Be cautious with force pushing as it can overwrite history. For more information see this guide on cherry-picking in Git. Using Git Sometimes you commit to an incorrect branch and now you want to move the commit to the correct branch. Here's how to handle the situation. Sh(g)it happens. I mean it is usual to clone the main branch, create or switch to a dev branch and then commit the changes to this dev branch which is merged to the main later. Imagine you follow the same only you forget to switch to the dev branch and you made the commit to the main branch. But before pushing, you want to move this commit to the dev branch instead. You should also remove the commit from the main branch. Let me help you by showing the steps for: Moving the commit to the correct branch. Reverting the commit from the incorrect branch. Moving commit to another branch. First, let's address the issue that I encountered: While working with three branches, I was supposed to make one commit to the header branch and another to the footer branch. The first commit to the header branch was correct but unfortunately, I made the second commit to the header branch instead of the footer branch. When I checked the git log, it was pretty clear to me that I made a commit to the wrong branch. Now, let's take a look at the steps to move the commit to another branch. Step 1: Find the hash of the commit. To find the hash of the commit you want to move, you can use the git log in the branch where you made a wrong commit. I made a wrong commit in the head branch so I'll be using git log there: git log. Once you find the hash, copy the hash. Step 2: Switch to the target branch. Next, switch the branch in which you want to move the commit. For that purpose, you can use the git checkout command: git checkout . In my case, I want to move to commit to the footer branch, so I'll be using the following: git checkout footer. Step 3: Move the commit to the target branch. Once you switch to the target branch, use the git cherry-pick command along with the hash you copied from the first step: git cherry-pick . To verify if the commit was moved or not, you can check the git log: git log. There you go! Revert the incorrect commit. When you use the cherry-pick command, it does not move the commit but copies the commit to the current branch. So you are still left with the incorrect commit on the first branch. The solution is to revert the incorrect commit. For that, first switch to the branch in which you made the incorrect commit: git checkout . In my case, the branch name was header so I will be using the following: git checkout header. Now, if you check the git log, you will still find the incorrect commit which you recently moved using the git cherry-pick command. To revert this commit, you append the hash of the target commit to the git revert command as shown here: git revert . It will open the text editor telling you it is reverting the commit. It creates another commit without those changes resulting in the removal of the specified commit. Close the text editor and that's it. Once you close the text editor, you will see an output telling you that the commit has been deleted. There you have it! Wrapping Up In this tutorial, I went through how you can move your commit to a different branch and also explained how you can remove thegit log -n 10 --oneline ========== Git log --stat Commits with file change stats Identifying impact of commits git log –patch Full diff patch for each commit Analyzing commit code changes git log –graph Branch and merge history Understanding branching You can combine --oneline with the -n flag to only show the last n commits. For example: git log --oneline -n 5 This would just print the id/messages of the 5 most recent commits, helping you quickly spot the commits you care about. Some other useful options for searching and filtering Git commit history include: --author="Name": Only show commits by a certain author --after="1 week ago": Only commits more recent than a date --grep="fix bug": Only commits with a matching message I recommend experimenting with git log options like these to slice and dice your commit history in different ways. Once you identify the target commit you want to move, make a note of the commit ID hash – you'll need it later. Checking Out the Destination Branch Now that you've found the commit you want to move, the next step is switching over to the target destination branch where yougit movement explained ========== When committing changes to multiple branches, you might need to move a commit from one branch to another. For instance, let's say you have a main branch and a feature branch. You accidentally committed a bug fix on the feature branch that should actually be on the main branch. To start the process, check out the main branch using git checkout main. One way to avoid damaging your production branches is by creating a new throwaway branch before making any changes: git checkout -b test-branch. This ensures you don't accidentally destroy your main production branches. You can always delete the test branch later. To move the commit, revert the commit on the source branch first. Instead of modifying the commit history, it's recommended to use the git revert command followed by the commit ID. This will create a new commit that reverses or undoes the changes from your target commit. Once you've cleaned up the original branch, cherry pick the commit onto the destination branch main. Cherry-picking takes the changes introduced in an existing commit and re-commits them as a new commit on top of your current branch. This can be done using the git cherry-pick command followed by the commit ID. However, if you're rewriting commit history by adding or moving commits, there's a chance your changes may conflict with work that's happened on the destination branch since those commits occurred.In automatic merge, conflicts often arise; resolve by manually editing main.py to reconcile changes from the picked commit and the latest branch versions. Begin by checking status with git status, then identify files with merge conflicts using a text editor of choice. Edit the file to remove conflict markers and restore all changes properly. Stage the file with: git add main.py Repeat for any other conflicted files, then commit with: git commit -m "Picked commit abc123 from feature" Resolving conflicts requires manual work; however, it's a crucial skill for integrating commits between diverged branches. Be patient, check status regularly, and utilize visual diff/merge tools if needed. To verify the commit was successfully moved, confirm that: The commit no longer exists on the original feature branch The commit now appears on your destination main branch Check this by running: git log --oneline feature git log --oneline main You should see the commit ID only appears on main. If needed, use git diff or check file hashes before/after to guarantee code changes were also moved. If the commit appears on both branches, double-check that the revert worked and you didn't accidentally merge instead of cherry-picking. Being diligent ensures you don't end up with duplicated commits across branches. Alternative methods for moving Git commits include soft reset, interactive rebase, or simply reverting and then cherry-picking. While these alternatives may offer more flexibility, cherry-pick is generally the safest and simplest option. To move a commit between two branches safely: Identify the target commit's ID hash with git log Checkout destination branch with git checkout Revert commit on source branch with git revert Cherry-pick commit on destination branch with git cherry-pick Resolve any merge conflicts before finalizing cherry-pick Verify commit history to confirm successful move Following this process ensures you safely port commits between branches without losing work or muddying up history. While it takes a few steps, committing to the wrong branch happens to all Git users. Therefore, knowing how to cleanly move commits to the right place is an essential skill in your version control toolbox. For further improving your Git skills, review these tutorials, training guides, and handy reference docs: Internalizing these Git skills will enable you to confidently manage repositories, collaborate with your team, and recover from just about any scenario with work and commit history intact.To efficiently manage your codebase in Git, sometimes it's necessary to relocate your most recent commit(s) to a new branch. This can be done for several reasons such as correcting mistakes or isolating changes. Moving commits to a new branch is useful for organizing and maintaining a clean workflow, especially in collaborative environments. It enables each branch to represent a specific task or feature, promoting better organization and testing. You can use various Git commands to accomplish this task. The HEAD pointer points to the most recent commit and reflects the currently checked out commit, which is also known as the working tree. You can specify commits by their hashes using relative references such as ^ for moving up one commit or ~ for moving up a specified number of times. The git reset command moves the current head to a specific commit. The --hard option resets the files in the index or staging area, losing any changes made to those files. Using Git branch commands, you can create new branches and switch between them using `git checkout `. To move recent commits to a new branch, follow these steps: Step 1: Check the current status of your repository by running `git status` in your terminal or command prompt. This will help you identify any changes or conflicts in your repository. Step 2: Create a new branch where you want to move the recent commit(s) using the command `git checkout -b new-branch-name`. Replace `new-branch-name` with the desired name for your new branch. Step 3: Move the most recent commit(s) to the newly created branch. You can use the git reset command to achieve this, depending on whether you want to move just one commit or multiple recent commits.you can adjust the reset command to match the number of commits. For example, to move the last three commits: git checkout original-branch-namegit reset --hard HEAD~3git checkout new-branch-namegit cherry-pick original-branch-name..HEAD@{1}In dis command, HEAD~3 indicates de last three commits, and HEAD@{1} refers to de state of de branch before de reset. Step 4: Verify ChangesAfter movin de commits, ees essential to verify dat everyting is in order. Use de git log command on both branches to ensure de commits have been movin correctly: git logCheck de log for both de original and new branches to confirm de commits are where dey should be. Git helps organize changes with branches, but wrong commits can cause problems. To move changes, switch branches using de "git checkout " command. To correct committed mistakes, soft reset before switching so your changes aren't commited yet. While Git keeps track of your daily changes, it also features systems like branches that help you organize. If you're not careful, though, you can end up with problems like commits and changes made to de wrong branch dat can be difficult to solve without de right commands. Moving Changes (If You Haven't Committed Yet) Git watches over your whole folder, but changes you make to files are not tied to a specific Git branch until you commit dem. You can move branches, and bring those changes along with you. De simplest option is to simply switch branches, but dis will only work if de two branches you're targeting have a matching history: git checkout feature You can get around dis in a few ways. De first ist by makin a new branch, and den merging de diverging histories: git checkout -b tempfeaturegit checkout featuregit merge tempfeature You can also use git stash to store changes for later, and reapply dem on a new branch: git stashgit switch featuregit stash apply If you already commited, don't worry; you can always soft reset, so commits are not final until pushed to remote source control. If you did dat already, you can still fix de problem, but de record of your mistake will live on in your Git history, so ees best to do it locally before your coworkers see it. To undo commits, you can simply soft reset, usually just undoing de last commit made, but you can also pass in a reference to de commit ID: git reset HEAD~1 This will leave you at de "haven't commited yet" state, after which you can use de methods above to fix de problem. Alternatively, you can use git cherry-pick. Dis command copies commits from one branch to another, and is a nice way of picking out commits and movin dem to new branches in a clean manner. Run git log to find de ID of de commit you want to revert: git log Then checkout de feature branch, assuming your changes have been commited, and run cherry-pick: git switch featuregit cherry-pick After dat, dere will still be a duplicate commit on de main branch. You can reset dis and discard de changes if de feature branch is in proper order, or keep it and let Git sort it out once you merge. If you want to learn more about using Git, read about how you can always know what branch you're in, or see if you know all de basic Git commands. Branching is a central concept in de Git workflow, allowing multiple developers to work on a single project simultaneously in a distributed environment. Branching in Git helps developers work independently while maintaining stability. A developer can work on any feature of de software by creating different branches and finally merging all de branches to get de resulting software product. In dis article, we'll show you how to move a commit from one branch to anudder branch and have de commit's changesgit branchng and its usefullness branching is useful for many resons like bug fixin without changin the main version and also implementin new feautures ========== branching is ushualy done to move comit from one branche to anuvver but it can cause problum if you dont no which branche the comit was moved to so first you must creat a new branche then you can move the last comit of the master branch to this new branche ========== after that your master branch will be empty and only the new branche will have all the changesgit reset command can be used to move a set of commits or a single commit from one branch to another. ========== we have used the reset command with the number 2, as we have done our task of moving those commits to our required dummy branch and now we can remove those two commits. So the new commit history for the master branch looks like this: This means that a new dummy branch will contain the code for your changes. The master branch looks just like it did before you made this commit and pushed it to the master branch. if you only want to move a specific commit to a dummy branch and not move any other commits after it that exist in the master branch's sequence, you can use the cherry-pick command in git. where is a unique id associated with each commit and is automatically generated at the time of commit. A commit ID is an encrypted number generated using the Secure Hash Algorithm (also known as SHA). You can use the Git log command to view the history of commits for a particular repository. Branching in Git allows developers to work in an isolated manner and provides a different version of project history. You can move one or many commits from one branch to another branch if you want your changes on one branch to be shown on a different branch. You can move a commit to another branch, whether it's a new branch or an existing branch. git reset --hard HEAD~1 command is used to revert the branch to the state it was in before the last commit. git cherry-pick will select one commit specified by commit id from one branch and apply it to another branch you plan to move to. If you only want to move a specific commit to a new branch and not move any other commits after it that exist in the branch's sequence, then you can use the cherry-pick command in git. A commit_id is a unique id associated with each commit and is automatically generated at the time of commit.