|  |
| --- |
| **DON'Ts**   * Don't delete the '.git' directory. That's your repository * Don't do git operations as root or you will not own your files. 'root' will. And then we will not know who did the modifications. * Don't do 'git push --force'. * Don't do 'git push' without any args. Always specify which repo (origin or testbuild) and branch (local:remote). * Fetching from the remote repository   + Don't do 'git pull' because it is a fetch and merge in that order. That would put the remote commits after your local commits.   + For our workflow it is best to do the a fetch and rebase procedure described below. * Committed something and lost it? "git lost-found" will list dangling objects. e.g.: If you have a second branch off a first branch and rebase the first branch, the commits on the second branch will be dangling. * Sometimes after a failed push it appears that the 'origin' for my repository gets reset to be at HEAD. This is of course wrong because origin should point to the HEAD of the remote repository at all times. To fix this, update your branch using the fetch and rebase procedure described below. * Keep your git tree clean (no trash files, no other git repositories cloned inside the tree). That way, when you do 'git status' you will not get a huge message that you will end up ignoring.   **Read the Git User's Manual! It's very friendly**   * The git user manual is at: <http://schacon.github.com/git/user-manual.html> * Also there is the git community book: [http://book.git-scm.com/index.html​](http://book.git-scm.com/index.html)**​**   **Brief Outline of our Git Workflow**   * These are the steps we follow for each patch that we create. Each of these steps are described in **more detail in the sections below.** * Each ticket will result in one commit being pushed into the repository. * The ticket is created and assigned in JIRA. * Do your development. Make modifications for JIRA and commit your changes to your cloned repository. * Do a **fetch and rebase** to rebase your branch to have all the latest changes that have been pushed to the main repository. * If you need to squash several commits into a single commit, do an **interactive rebase**. * Push patch(es) to main repository:   + If other commits have been pushed, you will need to fetch and rebase first. Git will let you know by complaining that the push is "not fast forward".   + When you push, the server will run the 'hooks' scripts that do various checks on log, DOS LFs, etc. * Finally, you should resolve the ticket in JIRA.   **Some basic git concepts**  At this point I want to say that I really like git a lot. I use it constantly and do tons of git operations regularly. I have found it to streamline my handling of code during development. I promise it will be worth getting beyond the learning curve.   * git was originally written by Linus for doing Linux kernel development. So it was optimized to work on Linus's not-so-fast laptop. * git is distributed. Each git repository has all the information as the repository it was cloned from.   + This also means that you can completely trash your local git repository without affecting anybody else. Unless you push.   + So you can feel free to experiment. * git tracks changes.   + You can call them commits or patches. Same thing.   + The Linux kernel community emails lots of patches to each other and git is written to handle taking patches from email well.   + One caveat is that your email reader has to be running under Linux for this to work. Outlook reformats patches so git can't take them.   + The way around that is to send patches as attachments, not inline. * Git has the following concepts that you will need:   + repositories = when you clone, you have a local repository   + branches   + tags   + commit = your changes are written to your local repository   + push = your changes are pushed to the remote repository   + sha1 sums = each commit has a unique 40 digit hex sha1 sum   + the index   + the HEAD   + the tree = a set of files that are checked out from the repository * Commits vs. Pushes   + If you commit changes to a git repository, the changes are only local.   + When you push your commits to a remote repository, then that affects the group. * So the order is:   + Do your changes   + Use 'git add' to stage your changes for a commit.   + Use 'git commit' to commit your changes locally.   + Use 'git push' to push your commits to the remote repository.​ * Lots of git commands refer to a single commit. There are many ways to refer to the same commit. For example to look at the log of only the top commit you would do 'git log -1 <commit specifier> * # the commit's 40 digit sha1 sum, such as * $ git log -1 2165c7d130179f0cd67f64af62e5e94c2a6b2f88 * # HEAD refers to the commit that is at the head of the currently checked out branch. * $ git log -1 HEAD * # HEAD^ refers to the commit before that * # HEAD^^ refers to two commite before the head. So on. * # HEAD~1 is the same as HEAD^. So you can do HEAD~39 to specify the 39th commit before the HEAD. * $ git log -1 HEAD~39 * # You can use the tag name. * $ git log -1 rel\_imx\_2.6.31\_10.05.02 * # You can use a local branch name * $ git log -1 my\_branch * # or a remote branch * $ git log -1 origin/imx\_2.6.35   # You can use ^ or ~ on tag names and branch names also.   * branches   + you can check out one branch at a time.   + branches are cheap. You can create them or distroy your local branches easily.   + you can switch easily between topic branches.   + your branch can track changes on a remote branch (branch in a remote repository).   **How to clone and set up remotes**  # Create a clone of our main kernel git repository:  git clone ssh://sw-git.freescale.net/git/sw\_git/repos/mcu-sdk.git  # Example to add a remote to a cloned repository  # Then add the mcu-sdk\_rio repository as another remote in the same repository  cd mcu-sdk  git remote add mcu-sdk\_rio ssh://10.192.242.9/srv/git-repos/mcu-sdk\_rio.git  git fetch mcu-sdk\_rio   * Now your repository has two remotes (remote repositories):   + origin = mcu-sdk.git     - This is the mainline repository we release from.   + mcu-sdk\_rio = mcu-sdk\_rio.git     - We use this repository for local usage, etc. * To see the remotes: * $ git remote -v * To see the remote branches * $ git branch -a   + The remote branches are either origin/... or mcu-sdk\_rio/... depending on which remote repository they came from.   **Repo Status**   * Get status of uncommitted changes and working tree. Shows staged modified files, unstaged modified files, and untracked files * git status * Get commit history: * git log * List branches: * git branch * List all branches including remote tracking branches: * git branch -a * Diff of index (staged) changes: * git diff --cached * Diff of working dir (unstaged) changes: * git diff * Diff of all uncommitted changes: * git diff HEAD * Diff of all unpushed changes: * git diff origin/dev\_kpsdk\_1.0\_ga * Note that if you add --stat to any of the git diff commands it will give a nice readable diffstat instead of the diffs.​   **Managing remote repositories**  Your local repository can fetch from several remote repositories.   * To see what remote repositories your local repository is aware of do: * git remote -v * To add a remote, do: * git remote add [remote name] [repository url] * # i.e. * git remote add testbuild ssh://sw-git01-tx30.am.freescale.net/git/sw\_git/repos/linux-2.6-testbuild.git * # Then fetch the changes from that remote * $ git fetch [remote name]   **Creating/Checking out a branch**   * To see all branches do * $ git branch -a   + You will see branches listed as remote/branch name, such as origin/imx\_2.6.35 * To check out a remote branch as a local branch: * $ git checkout -b [LOCAL BRANCH NAME] [REMOTE BRANCH NAME] * # i.e. * $ git checkout -b dev\_kpsdk\_1.0\_ga origin/dev\_kpsdk\_1.0\_ga * To check out a local branch that you have already created: * $ git checkout [LOCAL BRANCH NAME] * To delete a local branch named 'mybranch' (note that you can't have the branch you are deleting checked out) * $ git branch -D mybranch * to cherry-pick a commit from another branch: * $ git cherry-pick <sha>   **git add = Staging changes for commit**  When you want to make a commit, first you **add** the changes so that they are staged for commit. Then you can **commit** the changes.   * To only add changes to files that were already being tracked: * $ git add -u * 'git add' is also used to add a new file to repository and to be committed or to add some changes that have been made to a file to be committed: * $ git add platform/drivers/can/src/flsl\_flexcan\_driver.c * Note that 'git add' is VERY DIFFERENT from 'cvs add' * When you do 'git add', git saves off the changes to be committed. * If you keep editing the file after doing the 'git add', the new changes are not saved off for committing. So you have to do 'git add' again. * So it is always essential to do 'git status' after doing 'git add' to see if all your changes got added AND no extra files got added. * It is a common problem to run 'git add .' to add all files, and then find that some files that were created by the editor as backups (file name ends with a tilda~) were added.   + So don't do 'git add .' or 'git add --all' * To remove a modified file from staged (to be commited) list (note that this also causes the file to no longer be tracked in git):   git rm --cached <file>   * To remove a file from working tree and index: * git rm <file> * To restore a deleted file: * git reset HEAD <file> * git checkout <file> * Add and remove can work with wildcards or recursively.   **git commit**   * Before doing the commit, you need to use 'git add' to add your changes. * do 'git status' to make sure that you are committing only what you want to commit. * Commit staged changes: * git commit * You can use '--amend' to commit your staged changes to the commit that is at HEAD: * git commit --amend * Also, you get an opportunity to edit the commit log. To add a signoff line for the commit that is at HEAD: * $ git commit --amend -s * You can use 'git reset --hard' to remove the previous commit. **It also removes the changes from the working tree.** * git reset --hard HEAD^   **Fetch and Rebase**  To rebase your branch to have all the latest changes that have been pushed to the main repository:   * For example, if you are on the a branch that is tracking the remote origin/dev\_kpsdk\_1.0\_ga branch: * # First, fetch the changes from the remote repository: * $ git fetch origin * # Next, rebase your changes ON TOP of the remote branch. * $ git rebase origin/dev\_kpsdk\_1.0\_ga * The rebase operation will stop on merge conflicts. Fix the conflicts and run: * git rebase --continue * Then do a 'git status' and 'git log'. Make sure that you only have the commits you want to have. For example, if a commit was backed out on the git server, it may still be in your local tree. If that is the case, do an interactive rebase (see below) to remove it. * Note that we **do not do "git pull"**. That would grab the remote changes and put them AFTER your changes. We want your unpushed changes to be on top of the remote commits. * If you don't have any local commits, "git pull" is fine.   **Squashing commits**  If you made multiple commits, you need to squash them into one commit using an **interactive rebase**.   * For example, with a branch based on our dev\_kpsdk\_1.0\_ga kernel branch do: * git rebase -i origin/dev\_kpsdk\_1.0\_ga * An editor will pop up.   + In the editor, change "pick" to "squash" for all but the first commit.   + Note also that you can delete commits by deleting lines from the editor window.   + Or by shuffling the order of the lines, you can reorder the commits.   + The rebase will start when you save and close the editor window.   **Working with patches**   * Apply and commit all patches in a directory: * git am ../patches/\* * Apply a patch (but don't add or commit): * git apply -p<n> <patch file> * git apply --cached <patch file> * Email a patch: * git send-email --to lala@freescale.com --from bob@freescale.com <patch file> * Create a patch for the three commits that are at the top of the current branch: * git format-patch -3 * Generate one patch for the commit 8e12ea8fac0bc15e502c61ab9e2f57546bbe3091   git format-patch -1 8e12ea8fac0bc15e502c61ab9e2f57546bbe3091  **Cleaning up your tree**  To delete all untracked files:  git clean -fdx  **Cleaning up your branch**   * Sometimes a commit gets backed out on the main repository. Usually an email is sent out urging people to clean up their branches and don't accidentally re-push the commit.   ## save your work as a new branch called 'temp':  $ git branch temp  $ git fetch origin  $ git reset --hard origin/dev\_kpsdk\_1.0\_ga  # Now the branch is set to be the cleaned up remote branch without your changes.  # Cherry-pick your commits over. For example if you had 3 commits:  $ git cherry-pick temp^^  $ git cherry-pick temp^  $ git cherry-pick temp  # Delete your temp branch when you are done:  $ git branch -D temp   * Here is another method: * To clean up your branch, do the fetch/rebase procedure (also described elsewhere on this page), then use interactive rebase ('git rebase -i ...') to get rid of the commits. For example, assume that you are on branch dev\_kpsdk\_1.0\_ga and have one local commit and you need to clean up: * ## save your work as a new branch called 'temp' in case you need to start over: * $ git branch temp * $ git fetch origin * $ git status * On branch dev\_kpsdk\_1.0\_ga * Your branch and 'origin/dev\_kpsdk\_1.0\_ga' have diverged,  and have 7 and 4 different commit(s) each, respectively. * # Note here that git is telling you that two patches were backed out plus your one local commit. * $ git rebase origin/dev\_kpsdk\_1.0\_ga * First, rewinding head to replay your work on top of it... * Applying: AAAA: XXXXXXXXXXXXXXX * Applying: BBBB: XXXXXXXXX * Applying: CCCC: XXXXXXXXXXXX * # So 'git rebase' moved the backed out patches to the top of your branch with your local commit on top of them. * $ git status * On branch dev\_kpsdk\_1.0\_ga * Your branch is ahead of 'origin/dev\_kpsdk\_1.0\_ga' by 3 commits.  nothing to commit (working directory clean) * $ git log * $ git rebase -i HEAD^^^ * # Do the interactive rebase. It will bring up an editor window. * # Delete the two lines that correspond to commits that aren't yours to rid * # of the backed out patches while keeping yours. * # Delete your temp branch when you are done:   $ git branch -D temp​  **git push**  After your patches have been approved by the reviewer, you can push to the main repository:  # The format of 'git push' is:  # git push <remote> <localbranch>:<remotebranch>  # Such as:  $ git push origin my\_local\_branch\_name:dev\_kpsdk\_1.0\_ga  # If your local branch is the same as the remote branch, you can just do:  $ git push origin dev\_kpsdk\_1.0\_ga  **How to separate out one commit into two commits**   * You start out with a patch already committed. You want to separate out your changes into two commits, moving the commits to the 'driver' directory to the second patch. The original patch was XXXXXXXX, the new patches will be XXXXXXXX-1 and XXXXXXXX-2 * You should do 'git status' between every step to see what git is doing. * Make sure you have no changes waiting to be committed:   + # git status     - should say 'nothing to commit'   + # git log -1 --stat     - should tell you what files you have changed in your commit. * Duplicate your work on branch 'temp'.   + # git branch temp   + After this step, if things get messed up, you can always get back by doing 'git reset --hard temp' * Back out the commit, but leave the changes added.   + # git reset --soft HEAD^ * Unstage the changes under drivers/. The changes will still be in the tree. The other changes are still added for commit.   + # git reset drivers/ * Commit, editing the cr to be XXXXXXXX-1   + # git commit -c temp * Add the drivers/... changes that are still in the tree. Commit, editing the cr to be XXXXXXXX-2   + # git add -u   + # git commit -c temp * Check to see that you have two commits now:   + # git log --stat * Check that your commits have the same content that you started with:   + # git diff temp * Read some man pages or something it understand how wonderful and powerful and benevolent git really is.   **Recommended Reading for git**   * Recommend reading the following in order: * The Git User's Manual is the best reference besides the man pages. <http://www.kernel.org/pub/software/scm/git/docs/user-manual.html> * Git tutorial:   + There is a really nice git tutorial in the man pages.   + man gittutorial * man pages. Note that you have to add a hyphen, i.e. 'man git-commit' to see the man page on 'git commit'. * <http://www.kernel.org/pub/software/scm/git/docs/> * Git tutorial: <http://www.kernel.org/pub/software/scm/git/docs/gitcore-tutorial.html> * Everyday git with 20 commands - <http://www.kernel.org/pub/software/scm/git/docs/everyday.html> * Git for CVS users: <http://www.kernel.org/pub/software/scm/git-core/docs/gitcvs-migration.html> * Git Homepage: <http://git-scm.com/> * More documentation links: <http://git.or.cz/gitwiki/GitDocumentation> * Linus talks about how he feels about subversion. He also talks about git a little: <http://www.youtube.com/watch?v=4XpnKHJAok8> * Another git video: <http://www.youtube.com/watch?v=8dhZ9BXQgc4>   **To request an Austin UNIX domain account for Git access:​**   1. Go to https://hclfslitsm.service-now.com/ess/services.do 2. Under the "Order Things" panel, click "Services" 3. Then click "Engineering Compute Service Request" 4. Then click "EC Accounts" 5. In the "Comments" field, enter "Need a CDE-TX UNIX account" 6. In the "Core ID" field, enter your Core ID 7. In the "Site" field, enter "TX30" because that is where the Git server is located 8. In the "Unix domain" field, enter "CDE-TX" because that the Git server's domain 9. In the "Unix group" field, request "Primary: sw\_dev; Secondary: none" 10. Click the "Order Now" button to submit the request   Your ticket will be assigned to someone in IT who will send you a form requesting additional (mostly redundant) information that you must complete and return. In addition, your direct Freescale manager will be contacted to approve that you need an account, and group owners will be contacted to approve your groups.  It can take a 2-3 business days for your account to be created. If it is taking longer, then maybe there is a problem. Add a comment to your ticket asking the status, or you can email the ticket ID to George McCullough and he will follow-up with the UNIX admin team.​​  This page was derived from Alan Tull's, refer to the original page at http://wiki.freescale.net/display/MADLinux/Git+Workflow |

false,false,1