Git Tips and Tricks

Overview.

This page gives some helpful hints and tricks on using git, along with a suggested work-flow, with an explanation of its motivation and details.

Some pertinent Git details.

It is assumed at this point that you have a passing familiarity with CVS and/or Subversion; and that you have at least obtained a local copy (cloned) of a remote repository.

- Unlike CVS and SVN, every local working area is also a repository.
- Unlike CVS and SVN, repositories are *not* sub-divisible: in the former systems, one can easily check out only a subsection of a package; with git, it's all-or-nothing.
- A repository may have zero or more remote repositories to which items may be pushed (or from which they may be retrieved): Git is a truly distributed system.
- Branches are easy to create, merge and destroy.
- The, "unit of change" is a commit, labeled by its SHA1 hash. A tree's, "state" is a collection of commits. Merging branches multiple times is therefore trivial because it consists of comparing commit lists.
- Git commit trees do **not** record history: they record changes. A simple change has one parent. Creating a branch creates another child of the parent commit; and a non-trivial merge joins two branches -- the resulting commit has two parents.

Git (very) basics.

Getting help.

- `man git`
- `git help <command>`
- `man git-<command>`
- The git community book
- Pragmatic Guide to Git
- Google

Glossary.

- **Add**: put a file (or particular changes thereto) into the index ready for a commit operation. Optional for modifications to tracked files; mandatory for hitherto un-tracked files.
- **Alias** shorthand for a git (or external) command, stored in a `.gitconfig` file.
- **Branch**: a divergent change tree (eg a patch branch) which can me merged either wholesale or piecemeal with the master tree.
- **Commit**: save the current state of the index and/or other specified files to the local repository.
- **Commit object**: an object which contains the information about a particular revision, such as parents, committer, author, date and the tree object which corresponds to the top directory of the stored revision.
- **Dirty**: a working area that contains uncommitted changes.
- **Fast-forward**: an update operation consisting only of the application of a linear part of the change tree in sequence.
- **Fetch**: update your local repository database (**not** your working area) with the latest changes from a remote.
- **HEAD**: the latest state of the current branch.
- **Index**: a collection of files with stat information, whose contents are stored as objects. The index is a stored version of your working tree. Files may be staged to an index prior to committing.
- **Master**: the main branch: known as the trunk in other SCM systems.
- **Merge**: join two trees. A commit is made if this is not a fast-forward operations (or one is requested explicitly.
- **Object**: the unit of storage in git. It is uniquely identified by the SHA1 hash of its contents. Consequently, an object can not be changed.
- **Origin**: the default remote, usually the source for the clone operation that created the local repository.
- **Plumbing**: the low level core of git.
- **Porcelain**: higher level, user-visible interfaces to the plumbing.
- **Pull**: shorthand for a fetch followed by a merge (or rebase if `--rebase` option is used).
- **Push**: transfer the state of the current branch to a remote tracking branch. This must be a fast-forward operation (see merge).
- **Rebase**: a merge-like operation in which the change tree is rewritten (see Rebasing below). Used to turn non-trivial merges into fast-forward operations.
- **Ref**: a 40-byte hex representation of a SHA1 or a name that denotes a particular object.
- **Remote**: another repository known to this one. If the local repository was created with "clone" then there is at least one remote, usually called, "origin."
- **Stage**: to add a file or selected changes therefrom to the index in preparation for a commit.
- **Stash**: a stack onto which the current set of uncommitted changes can be put (eg in order to switch to or synchronize with another branch) as a patch for retrieval later. Also the act of putting changes onto this stack.
- **Tag**: human-readable label for a particular state of the tree. Tags may be simple (in which case they are actually branches) or annotated (analogous to a CVS tag), with an associated SHA1 hash and message. Annotated tags are preferable in general.
- **Tracking branch**: a branch on a remote which is the default source / sink for pull / push operations respectively for the current branch. For instance, origin/master is the tracking branch for the local master in a local repository.
- **Tree-ish**: a ref pointing to either a commit object, a tree object, or a tag object pointing to a tag or commit or tree object.
- **Un-tracked**: not known currently to git.

Initializing a repository.

```
git clone <repository-spec> <local-dir>
```

or

```
mkdir <local-dir>
cd <local-dir>
git init
```

or (FNAL Redmine-specific)

```
rclone [-r <repo>] <project> <local-name>
```

where `rclone` is defined in cet-chg:export:unix-admin/profile.d/rclone.sh

Basic log information.

```
git log [<tree-ish>]
```

- **Important tip**: log messages have an optional structure, since many git commands only look at the first line. Get into the habit of putting only a short synopsis on the first line of a log message and putting more detailed information on subsequent lines. You can omit the `-m` option entirely and an editor (as specified with `VISUAL` or `EDITOR`) will be started.
- This command is *extremely* versatile. You may want to have a couple of aliases defined in your .gitconfig file (see attached gitconfig for ideas).

Working in your local repository.

- Obtain differences with

  ```
  git status
  ```

- Move files from one part of your directory tree to another:

  ```
  git mv <old-path> <new-path>
  ```

- Delete unwanted tracked files:

  ```
  git rm <path>
  ```

- Add un-tracked files:

  ```
  git add <un-tracked-file>
  ```

- Stage a modified file for commit:

  ```
  git add <file>
  ```

- Commit currently-staged files:

  ```
  git commit -m <log-message>
  ```

- Commit only specific files (regardless of what is staged):

  ```
  git commit -m <log-message> <files>...
  ```

- Commit all modified files:

  ```
  git commit -a -m <log-message>
  ```

- Un-stage a previously staged (but not yet committed) file:

  ```
  git reset HEAD <file>
  ```

- Examine a representation of your change tree with log files and patch descriptions:

  ```
  gitk
  ```

- Get differences with respect to the committed (or staged) version of a file:

  ```
  git diff <file>
  ```

- Get differences between local file and committed version:

  ```
  git diff --cached <file>
  ```

Basic interaction with local branches.

- Create (but do **not** switch to) a new local branch based on the current branch:

  ```
  git branch <new-branch>
  ```

- Create and switch to a local branch based on the current branch:

  ```
  git checkout -b <new-branch>
  ```

- Change to an existing local branch:

```
git checkout <branch>
```

- Examine the list of commits in the current branch **not** reflected in another branch:

```
git cherry -v <branch>
```

- Merge another branch into the current one:

```
git merge <branch>
```

- Delete a local branch (eg after merging):

```
git branch -d <branch>
```

  OR (if changes have not been completely merged but you're sure you want to delete anyway):

```
git branch -D <branch>
```

Basic interaction with a remote branch.

Assuming you created your local repository with `git clone`, there is already one configured remote *origin* and you will have a local branch for each remote branch that existed at the time of your last `pull` or `clone`.

- Get the current list of remotes (including URIs) with

```
git remote -v
```

- Get the current list of defined branches with

```
git branch -a
```

- Change to (creating if necessary) a local branch tracking an existing remote branch of the same name:

```
git checkout <branch>
```

- Update your local repository ref database without altering the current working area:

```
git fetch <remote>
```

- Update your current local branch with respect to your repository's current idea of a remote branch's status:

```
git merge <branch>
```

- Pull remote ref information from all remotes and merge local branches with their remote tracking branches (if applicable):

```
git pull
```

- Examine changes to the current local branch with respect to its tracking branch:

```
git cherry -v
```

- Push changes to the remote tracking branch:

```
git push
```

- Push all changes to all tracking branches:

```
git push --all
```

Some more advanced operations.

- **Important tip**: if you're going to do a git operation the outcome of which is *even remotely* uncertain to be the desired one: **make a copy of your repository**:

```
mkdir -p <path-to-safe-dir>
tar -cf - . | tar -xC <path-to-safe-dir>
```

  Disk space is cheap and `rm -rf` is easy. Note that you must copy the **entire** repository, since all the important information is in the `.git` directory tree at the top level.

Stashing.

This is a good way quickly to get a clean tree if you want to `merge` or `rebase` (see below) to import changes from a branch without having to commit your current work.

- Save uncommitted changes to the current working area to the stash (**not** a commit operation):

```
git stash
```

- Apply previously-saved stash:

  ```
  git stash pop
  ```

  (pops off the changes and applies them to the current working area) or

  ```
  git stash apply
  ```

  which applies the changes but retains them on the stack.
- Examine the current state of the stash:

  ```
  git stash list
  ```

- Clear the entire stash:

  ```
  git stash clear
  ```

Rebasing.

Rebasing is **changing history**, if you think that git stores history. As mentioned above, it doesn't: it saves objects with parent, child and other (eg date, author, etc) information. In a truly distributed environment, the actual history will be different for every repository depending exactly how and when changes were fetched, merged or pushed.

Rebasing is a good way to do a couple of things:

1. "Squash" related commits in your local repository prior to a push (eg, "Implement feature X," "Tests for feature X" and, "Fix bugs found while testing feature X").
2. Simplify merging branches and keeping up-to-date with remote changes during long periods between pushes.

- **Important tip**: do not attempt to `rebase` anything that has already been pushed to a remote repository. Your next push will almost certainly fail (and quite right too).

Squashing related commits:

- Squash some of the last few commits in your current branch:

  ```
  git rebase -i HEAD~5
  ```

  Your configured editor (`VISUAL` or `EDITOR`) will be started and contain a list of your last five commits (most recent at the bottom) along with instructions on what to do. Commits can have their log messages reworded; commits can be removed entirely, combined with other commits or re-ordered. If you specified any rewording or squashing, you will be taken to an edit session for the commit message(s) after saving and exiting the current edit session.
- Squash, re-order or reword commits since divergence from <branch>:

  ```
  git rebase -i <branch>
  ```

Keep up to date with remote branches without merging.

```
git pull --rebase
```

or

```
git fetch <remote>
git rebase <remote>/<branch>
```

Resolving conflicts.

Any `pull`, `merge`, or `rebase` operation can result in a conflict during the application of a particular change from the remote branch. Follow the on-screen instructions to resolve problems. This will usually consist of doing a `git status` to list conflicts, editing the files and using `git add` to mark each conflict resolved. The process **must** either be allowed to continue by issuing a `git rebase --continue` or `git merge --continue` command, or the operation can be reverted with `--abort` instead of `--continue`. If in doubt, **copy your repository**.

Making a new remote branch.

- Create a new **local** branch based on an existing one:

  ```
  git checkout -b <branch>
  ```

- Do stuff.
- Push the branch to the remote:

  ```
  git push <remote> <local-branch-name>[:<new-remote-branch-name>]
  ```

Tagging.

- Tag the current state of a branch (eg for release):

```
git tag -am <message> <version>.
```

Note that the `-a` creates an `annotated` tag, which is itself a commit with a hash and a commit message. This is the closest analogue to the CVS tag command. Omitting the `-a` option will create a, "simple tag" which is actually a branch. In general, you will probably prefer annotated tags with version-branches created explicitly as desired.

- Push the tag to the remote:

```
git push --tags
```

Special notes on mis-tagging.

There are several things that can go wrong with tagging:

1. One can omit an intended `-a` option;
2. One can misspell the tag; or
3. One can omit or (horror!) fix a file and wish to update the tag.

If you have not pushed tags yet (See above) then the fix is trivial: for the first two cases, remove the erroneous tag with `git tag -d <tag>`; for the third, re-tag with `git tag -am <mesasge> [<tree-ish>]`. **However**, if you have already pushed tags, there are wider consequences. For this reason, altering pushed tags is *emphatically* discouraged: create a new tag. However, since you're going to ignore me and do it anyway, here's how to do what you want without getting into too much of a mess:

1. To remove an erroneous tag, someone with the manager rôle on the project must log into `cdcvs` directly as the repository user (*e.g.* `p-art`), `cd` to the bare repository with `cd /cvs/projects/<project>` and then remove the tag with `git tag -d <tag>`.
2. Back in your working directory, tag correctly and then push tags.
3. Now, you **must** alert all your developers that, if they have pulled the erroneous tag to their local repository, they will need to remove the tag from their local repository with `git -d <tag>` and then re-pull from the repository. Otherwise, deleted tags will keep re-appearing in the remote repository and/or users will be unable to pull or push to the remote.

Undo the last commit:

- Undo the commit:

```
git reset --soft HEAD^
```

- Do stuff.
- Recommit:

```
git commit -a -m <message> -c ORIG_HEAD
```

Note that the `-c ORIG_HEAD` clause causes git to use the meta-data from the previous HEAD (author, etc) with the exception of the commit message. Changing the `-c` to `-C` and omitting the `-m` option will cause git to reuse the commit message too.

Recover deleted (committed files).

- Get a list of all commits with deleted files:

```
git log --diff-filter=D --summary | less
```

- Find your file and note the SHA1 hash for that commit.
- Recover it:

```
git checkout <commit>^ -- file
```

Stage selected changes within a file.

-

```
git add --patch <file>
```

- Follow the on-screen directions.

The `.gitconfig` file

This file contains global (`~/.gitconfig`) or repository-local configuration settings. You can (eg):

- Set user and email information to label commits usefully:

```
git config --global user.name "Chris Green"
git config --global user.email <email-address>
```

- Set colors for various types of command output.
- Set which local branches track which remote branches.
- Set pull behavior for branches to be rebase rather than merge.
- Define aliases as shortcuts for internal or external commands.

See the attached .gitconfig example. Have fun!

A suggested work flow for distributed projects: NoSY

Overview.

Building on the tips and other points explained above, this workflow has the following advantages relative to always working on the master branch:

- It is easy to keep track of upstream changes even when working on a protracted task.
- The change tree remains simple, easy to understand at a glance and even (mostly) linear (revision trees with multiple developers can quickly start looking like a train switch yard)
- Unsightly "merge with branch" commits are minimized.
- It is easy to keep separate unrelated tasks upon which you may be working simultaneously.
- Commits related to each other can be kept together or merged for increased clarity.

For the purposes of having an easy-to-remember label, I will refer to this workflow as the "No Switch Yard" (**NoSY**) workflow.

Details of NoSY.

For each **specific, well-defined task**:

1. Create a local branch and switch to it:

   ```
   git checkout -b <local-branch>
   ```

2. Work on the branch, both committing regularly and keeping up-to-date with the remote (eg):

   ```
   git fetch origin; git rebase origin/master
   ```

3. When ready to push back to the main remote, squash related commits (see above).
4. Change back to your master branch:

   ```
   git checkout master
   ```

5. Make sure your master is up-to-date:

   ```
   git pull
   ```

6. Merge with the branch:

   ```
   git merge --ff-only <local-branch>
   ```

   If this operation fails:
   - Swap back to the other branch

     ```
     git checkout <local-branch>
     ```

   - Rebase again (upstream must have changed since your last rebase):

     ```
     git rebase origin/master
     ```

   - Go back to step 4.

7. Push changes to the master:

   ```
   git push
   ```

8. Delete the branch:

   ```
   git branch -d <local-branch>
   ```

Converting to NoSY half-way through a given set of changes.

Imagine that you have been making some commits to your local repository on the master branch, and you realize (perhaps because your project is turning out to be a bit more involved than you thought, or because a slew of changes have just appeared upstream) that you might have been better using **NoSY**. It's actually quite easy to swap to using **NoSY** without any disruption to your already-committed changes. Starting from your current position on the master branch:

1. Stash your current changes if appropriate:

   ```
   git stash
   ```

2. Create (but do not switch to) a branch which will contain all your local commits up to this point:

   ```
   git branch <local-branch>
   ```

3. Download the latest metadata from the remote:

   ```
   git fetch origin
   ```

4. Now, reset your local master branch directly to the current state of `origin/master`:

   ```
   git reset --hard origin/master
   ```

Note that you have **not** lost your local commits: they are on your local branch already.

5. Switch to your local branch:

```
git checkout <local-branch>
```

6. Do an initial sync between your local branch and the remote, resolving conflicts if necessary:

```
git rebase origin/master
```

7. Apply your stash, again, resolving conflicts if necessary:

```
git stash pop
```

8. Pick up at step 2 of the **NoSY** workflow above.

gitconfig - Sample ~/.gitconfig (1.071 KB) Christopher Green, 28/07/2011 12:10 PM