Git in a Nutshell

For Normal People (tm)

Jonas Jusélius

<jonas.juselius@chem.uit.no>
Centre for Theoretical and Computational Chemistry
University of Tromsø
N-9037 University of Tromsø, Norway

Rev 0.5.5:
`git clone http://git.jonas.iki.fi/git_guides.git`
Corrections and improvements are most welcome via e-mail using the `git format-patch` *facility.*

Contents

About this document

The intent of this document is to give an overview of git and to explain some of the aspects of working with git that I feel are somewhat poorly explained elsewhere. What I feel is missing is a manual which explains what the commands are for, how the pieces fit together and how to actually work with git on a daily basis. This guide has been written with an audience consisting of the typical academic programmer in mind, and relies implicitly on a repository setup as outlined in the "Git to CVS Migration Guide".

All git commands have excellent man pages explaining in excruciating detail the particular command. As you read through this manual, it might be a good idea to have a quick look at the corresponding manual pages of the commands, just to give you an idea of more advanced features lurking under the surface. For example, get all available information on the `git checkout` command, run

```
$ man git-checkout
```

Don't get overwhelmed by the amount of detail in the man pages. As you gain experience and confidence working with git you will learn to appreciate the finer details. Don't panic.

Finally a disclaimer. I'm not an expert on git, nor do I have a huge experience working with git. All comments, corrections and suggestions are most welcome!

1 Introduction to git

Git is a very powerful, easy to use and flexible revision control system. Although git has many advanced and flexible features, most basic day-to-day operations are very simple to use.

Git has been designed in very UNIX-like fashion; it's built from a set of small, efficient programs which do one thing, and do it well. By combining these programs, new high-level commands can be created to do more or less any desired task. This structure is reflected in how git commands are executed. There are usually two equivalent ways of executing a command, either by executing the command directly

```
$ git-command args ...
```

or by using the `git` command which wraps the most common commands for a more CVS-like interface

```
$ git command args ...
```

In this guide I'll use the second form throughout, mostly because modern shells like `zsh` and `bash` have command line completion features which interact very nicely with the `git` wrapper.

1.1 Creating a repository

Git makes it very easy to put projects (code, manuscripts, or whatever) under revision control. Suppose you have a directory which contains a number of files you want to have under revision control. The first step is to remove all files that should not be under revision control, i.e. files that can be generated from source (.o, .pdf, ...). When you have a “clean” directory tree, simply in the top level project directory run

```
$ git init
$ git add .
$ git commit
# edit the commit message, save and quit.
```

That’s it! If you don’t want to clean your project directory you can instead specify the files to include by explicitly giving the file names to `git add` instead of the current directory (’.’)

Running `git init` in a directory sets up the repository and the necessary files in a directory named `.git/` in the current directory. Since everything is contained in the project directory, no special permissions or groups are needed.

1.2 Cloning a repository

Ok, so you have a repository on a server somewhere, and you want to get your own working copy and dig in! Git supports many different protocols (http, ssh, git...), but for simplicity I’ll assume you have ssh access to the server. To get your own repository, simply run

```
$ git clone me@myserver:/path/to/myrepo .
```

This creates a directory called `myrepo`, with a copy of the remote repository in the current directory.

Git is quite different from CVS in most respects. When you clone a repository you get the WHOLE repository, everything, not just a working copy like in CVS! `git clone` is in principle almost the same as doing a[1]

```
$ scp -r me@myserver:/path/to/myrepo .
```

Within this personal repository you can do *whatever* you like, i.e. create branches, delete branches, tags, and of course commit as much as you like. It’s only when you push to the master that others can see your changes. In fact, your (cloned) repository can act as a master for someone else! The division into master and client is really quite blurred and artificial in git. When you clone a repository, the new copy will contain administrative information in `.git/config` about where it was cloned from, and has a slightly modified branch structure (as you will see), but apart from that it’s identical to its parent. And where we humans can’t choose nor change who our parents are, git has no problem in changing or removing the parent(s).

1.3 Working in a repository

With your repository in place, working with git is very easy. Work and edit your files, and whenever you have completed something to the point you think it could be worthy of a save, simply run

```
$ git commit -a
# edit the commit message, and save
```

[1] Actually git clone is a bit more selective, and it also sets up various administrative information, etc.

A commit does not make your changes public. The commits are local to your repository, and unless you let somebody clone or pull changes from you, they will remain hidden until you decide to publish them by pushing them to a common server.

2 Basic git

The most basic operation under a revision control system is saving the state of a project as a new revision. When you create new files these also need to be placed under revision control. This is done with

```
$ git add file(s)
```

This marks the files to be added to the repository next time you run `git commit`.

In fact, you can run `git add` on files already under revision control in the repository, to selectively mark them for inclusion in the next commit. This is known as *staging* files for a commit. We thus have a number of ways of marking files for a commit, either by directly specifying the files to `git commit` or by first adding them with `git add` and then running `git commit` without any files.

2.1 Branches

The use of branches is where git probably differs most from CVS to most users. Under git branches are used extensively, and they are essentially free. It costs nearly nothing to create branches, both in terms of time and disk space. Also, there is no need to tag first and branch second, like in CVS.

It's very easy to create a new branch under git

```
$ git branch foobar
```

This creates a branch named foobar which will be an exact copy of the latest revision of the current branch. To start working under this new branch you much first do a checkout

```
$ git checkout foobar
```

This command switches to the new branch, and updates the files in the source tree to the HEAD of the branch (HEAD is a symbolic tag representing the current branch). For more detailed information on how to specify revisions see section 6.3, p. 14.

The checkout command is thus very different from the `cvs checkout` command. Since branches are so cheap and easy to use, it's often convenient to create a new branch for a specific task. Such branches are known as topic branches. Branches are also good for experimentation; suppose you have a wild idea you want to try out, just create a branch, and discard it if it doesn't work out. Suppose we want to base the experiment on a branch named 'work', here is how to create and switch branch in one go:

```
$ git checkout -b wildidea work
# work, test, commit, work... realise it was a bad idea
$ git checkout work
# delete the wildidea branch
$ git branch -d wildidea
```

Deleting the branch 'wildidea' not only deletes the branch, but actually the whole commit history and all changes on the branch, so you will never be able to get back that information. So be sure that you are not throwing away something you might want to save.

2.2 Merging

Obviously branches would be very cumbersome to work with unless it would be nearly trivial to incorporate changes in one branch into another. Git provides two main ways[2] to do this; merging and cherry picking (for more information on cherry picking see section 6.4, p. 15).

Merging is the main mechanism for incorporating changes from one branch into another. Merging is trivial if you work in a slightly disciplined way, avoiding making changes to the same files in different branches without synchronising first. Of course, it's not a problem modifying the same files, but then you will have to select by hand which changes to retain. For more information on how to resolve conflicts see section 6.2, p. 14.

Suppose that the branch 'wildidea' in the previous section worked out, and that you want to merge the changes back into the 'work' branch:

```
$ git checkout work
$ git merge wildidea
```

If the work branch has not been changed since wildidea was created from it, this brings the work branch into the exact same state as the wildidea branch. If, on the other hand, the work branch has changed two things can happen: The changes do not overlap and the changes in wildidea are cleanly incorporated. If the same file in both branches have changed you will have a conflict which needs to be resolved (see section 6.2, p. 14).

Branch names don't have to be simple strings. In fact you can create branches with sub-branches exactly like a directory tree. This can be useful if there are many developers sharing a master repository. Every developer has his/her own branch tree, e.g.

```
$ git branch $USER/work
$ git branch $USER/crazy_stuff
...
```

As a last point, it's actually possible to merge many branches in one merge operation. Such a merge is called an octopus merge, see the `git-merge` man page for more info.

2.3 Tags

Tagging is a way of specifying a symbolic name to a specific revision (state) of the source tree. This makes it much easier to access that particular revision later.

Tags should not be checked out directly, rather used to create branches starting at the specified tag. For example, it's a good idea to tag every release

[2]Actually, there is a third way called rebasing, but that method is for experts only. See the `git rebase` man page for more info.

of a project. Later if someone finds a bug, it's easy to either go back to that state or to create a bugfix branch.

```
$ git tag version-1.12
...
$ git branch bugfix-1.12 version-1.12
$ git checkout bugfix-1.12
```

Tags are not specific to a particular branch, but by using "directory tree" like names one can emulate branch tags

```
$ git checkout crazy_stuff
$ git tag crazy_stuff/pre-release1
```

Tags can also be specified retroactively by specifying a revision

```
$ git tag working-version7 HEAD~42
```

Finally, tags are deleted by running

```
$ git tag -d working-version7
```

2.4 Moving, renaming and deleting files

As projects evolve, it often becomes necessary to move, rename or even retire files. For example, to rename and move a file in the repository do

```
$ git mv foo.py raboof/bar.py
```

If you want to delete a file or directory use

```
$ git rm myfile
```

Note that this does not delete the file irrevocably, since going back to a previous revision will bring it back (as it should).

3 Working with remote repositories

Even though revision control can be very useful in a one-man universe, it's when collaborating with others that git reveals its true power. The two basic operations when working with a remote repository are the pull and push operations, to retrieve and publish changes, respectively. Still, it's good to keep in mind that your own repository is a full-fledged repository, even when you are working with a remote repository. The only difference to the "master" repository is that a symbolic pointer called 'origin' is set up to point back to where the repository was cloned from[3]. The symbolic name 'origin' is simply a shorthand for the url of the remote repository, i.e. if you do `git clone me@repos.foo.org/proj.git`, then 'origin' is set to `me@repos.foo.org/proj.git`.

[3]You can in fact set up multiple remote repositories if you like, see section 6.10, p. 19

3.1 Remote branches in git

When you clone a repository, the repository you cloned will be referred to as a remote repository (even if it's on the same machine). The default remote repository is called 'origin'. The repository you just cloned has at least one branch ('master'), but probably more. When you clone a repository all the branches in that repository are renamed in your local copy by prefixing the branch name with '`origin/`'. Thus, the 'master' branch in the remote repository will be named 'origin/master' in your local copy. These branches are called 'remote branches' or 'remote-tracking branches', to distinguish them from *your* local branches. The point of having these branches is that after an update, they will always be in the exact same state as the corresponding branch in the remote repository, i.e. they track the remote repository.

To view all remote branches, run

```
$ git branch -r
```

Whenever you run `git fetch origin` *all* remote branches and tags are updated to the exact state of the remote repository. You can inspect the changes on those branches by either using `gitk`, or a combination of `git log`, `git whatchanged`, `git diff` and `git annotate` (see section 4, p. 9). For example, to inspect the latest changes on the most important remote branch:

```
$ git fetch origin
$ gitk origin/master
```

Note that `git fetch` updates the remote tracking branches, not your working branches! Never, ever, checkout a remote branch directly unless you absolutely want trouble.

3.2 Tracking remote branches

In order to incorporate changes in remote branches you need to merge your working copy with the remote branch, e.g.

```
$ git checkout mybranch
$ git merge origin/master
```

If you get a conflict, run `git mergetool` which will fire up the merge tool of your choice. When you have resolved all issues, run `git commit` to complete the merge (see section 6.2, p. 14).

The `git pull` command basically does a `git fetch` and `git merge` in one go. I recommend you *do not* use it unless you very carefully read the man page first, since it has some pitfalls!

As mentioned earlier, never work on a remote branch directly! This will cause terrible problems when you try to sync with the remote repository. Instead create a local branch *from* the remote branch and do your work on it instead:

```
$ git branch foobar origin/foobar
$ git checkout foobar
```

3.3 Publishing changes

At some point you will hopefully have produced a nice set of commits, leading up to something you find worthy of sharing with others. Before you push your changes to the remote server it is essential that you make sure that you are in sync with the branch you are going to push to. For a push to work it must result in a so called fast-forward merge.

Fast-forward means that the files the files on the other branch have not changed since the last pull. If a push fails because it is not fast-forward, you must first fetch, merge and possible resolve any conflicts before (re)doing the push.

When pushing changes I recommend being very explicit in order to avoid any surprises, e.g. to push the changes on mybranch to an *existing* branch called myuserid in the repository where we cloned from (i.e. origin)

```
$ git push origin mybranch:myuserid
```

If you have created tags that you want to make public you need to push them explicitly

```
$ git push --tags origin
```

3.4 Creating remote branches

To create a new branch on a remote server, git requires a very explicit syntax. To create a remote branch and push the specified branch to it, do

```
$ git push origin <branch>:refs/heads/<branch>
```

As a technical side note, the 'refs/heads' syntax refers to the actual directory structure in the internal git repository in `.git/`. Once you have created the branch, you can use the same syntax as when operating on local branches.

3.5 Deleting remote branches and tags

If you for some reason want to delete a branch on the remote server, just push an empty branch to the remote branch:

```
$ git push origin :<branch>
```

To delete a tag on the master you need to explicitly push an empty tag

```
$ git push origin :refs/tags/<tag>
```

3.6 Cleaning up

Branches come and go. Some branches have very long lifetimes and others just exist for a short while. When remote branches get deleted, git does not automatically register this when you do a `git fetch`, it just ignores them. Thus, the number of stale remote branches can grow, cluttering the branch listings. To get rid of all these stale branches simply run

```
$ git remote prune
```

To delete single stale branch you can use

```
$ git branch -d -r <remote>/<branch>
```

3.7 The aim of the game

With so many possibilities for organising both repositories and branches it's important not to forget that the ultimate aim should be to get your beautiful new features merged with the master branch on the master server! It's like the musketeers, "All for one, one for all!".

It's important to update often from the master repository, partly not to drift too far away from the master branch, and also to incorporate all bug fixes etc. At the same time it's also important to push to the master often and in small increments, since this makes it a lot easier for other developers to stay in sync with *your* work.

3.8 A word of advice

Git might seem a bit overwhelming in the beginning, so here are some recommendations for how to work with git until you get used to it: When you clone a remote master repository, you will get a set of remote branches (see section 3.1, p. 7 for more info on branches), and one local working branch called 'master'. This branch is in the exact state of the remote master branch (i.e. 'origin/master' and 'master' are identical). I suggest you keep it this way, and create a new working branch for yourself, e.g.

```
$ git checkout -b work
```

which creates branch 'work' and checks it out in one go. Now work like you normally do, and every now and then (every morning for example) run

```
$ git fetch origin
$ gitk origin/master
# if you like what you see
$ git pull origin master
```

Every time you have made changes which can be considered complete in some sense (you know best), do a commit

```
$ git status
# hmm, what did I actually change?
$ git diff
# oh yes...
$ git add file(s)
$ git commit
```

It's much better with many small commits than a few big commits as you will see in section 6.4, p. 15. Small commits matters also for commit review, and when finding bugs using `git bisect` (see section 6.8, p. 17), or for `git blame` analysis (see section 4.4, p. 11).

4 Examining repositories

Every now and then, on a more or less daily basis, I tend to forget which files I have modified so far. This frequently happens during debugging sessions, where you easily end up all over the place in the hunt for the offending piece of code.

Then it can be highly convenient to get a listing of files which have changed since the last commit. If you run

```
$ git status
```

you will get a compact status report of which files have modifications, which files are not under revision control and files staged for a commit (with `git add`) but not committed yet. Often you have a bunch of files that should not be under revision control and that you really don't care about, e.g. object files and the like. To avoid having git status always list these files you can edit the file `.gitignore` in the project directory and list files and file patterns (one per line) that you want to ignore:

```
# example .gitignore file
*.[oa]
*~
*.bak
```

Have a look at `man gitignore` for a detailed description of how patterns are interpreted. It's usually a good idea to place the `.gitignore` file under revision control too.

4.1 Examining logs and changes

Often it's nice to be able to browse the commit logs on a branch, either to identify a particular commit or just to see what others might have committed. To view the commit log run

```
$ git log
```

If the information provided by git log is not enough, and viewing actual changes is too much, then

```
$ git whatchanged
```

shows the commit log *including* a listing of which files had modifications in a particular commit.

4.2 Examining changes

One of the most important aspects of revision control is that it allows you to follow how files change over time. We have already looked at how to examine the development history through logs and by listing which files have changed. We shall now turn our focus on how to examine how the actual files change between revisions. For this purpose git provides a very powerful command:

```
$ git diff
```

Simply running this command without any arguments prints out the differences between your working copy and the last checked in (staged) version[4]. This

[4]A technical detail, which most people safely can ignore: git-diff shows difference between the index (staging area) and the working copy, not between the HEAD (last commit) and the working copy.

can be extremely useful at times. `git diff` can also show differences between arbitrary revisions, branches, tags and so on. To view the differences between two branches

```
$ git diff branch1 branch2
```

Alternatively, when you finally have located a commit or a file revision that you want to examine in some detail, the versatile `git show` command can be useful. `git show` is also convenient for retrieving older revisions of *files*. Since git only deals with whole revisions, i.e. the *state* of the repository at a given time, it's in principle not possible to retrieve the revision of a single file. Sometimes however it's convenient to be able to reset a single file to an older state. So for example to save a particular file 7 revisions back:

```
$ git show HEAD~7:foobar.c >foobar~7.c
```

If you want to reset a file to given state, you can directly use

```
$ git checkout HEAD~7 foobar.c
```

4.3 Searching a repository

Quite often one has a need to search all or some of the files in a source tree for a particular string. Of course, it's reasonably simple to quickly filter the relevant files on the command line and `grep` for the string. Things become substantially more complicated if you want to search in an older revision. Luckily, git has a simple solution:

```
$ git grep regexp
```

This will match the regexp for all files in the current revision. `git grep` has a lot of flags to limit and refine searches.

4.4 Who to blame

We have all been there, someone has messed up your code and you don't know who to blame[5]. Fortunately git comes to our rescue:

```
$ git blame file
```

This command prints out the file with every line nicely annotated with who changed it and when. You can also use

```
$ git gui blame file
```

for graphical blame.

If you want to find what commit introduced given change, you can use so called *pickaxe* search:

```
$ git log -S'<changed line>' file
```

[5]This is because in 9 out of 10 cases it's you yourself, with a perfect, albeit short, memory.

5 Using git for collaboration

Revision management goes well beyond just source code management for a group of programmers. Revision management is useful for most tasks which are expected to evolve with time, like for example manuscripts. Since git is very easy to set up, and supports a wide range of communication protocols, git can be useful for many collaborative tasks. In the following section we will examine how git can be used to collaborate in a highly disconnected environment, where none of the participants have access to a common server or each other's machines. This is a typical situation which arises for shorter-term projects, like when collaborating on a scientific manuscript. To facilitate this situation git offers a powerful e-mail facility for communication changes.

Suppose you are working on a LaTeX manuscript and you want to have the whole manuscript under revision control:

```
$ cd ~/tex/manus/
$ git init
$ git add manus.tex fig1.ps fig2.ps
$ git commit
```

That's it! Now you can work happily, and remember to commit every now and then so that you always can go back in history if you need to.

At the point when you are ready to send the manuscript to your collaborators, you can make an archive of the whole project and send it by email to your collaborators[6].

```
$ cd /tmp
$ git clone ~/tex/manus
$ tar vfcz manus.tgz manus
$ rm -rf manus
# mail and attach /tmp/manus.tgz
```

Alternatively you can use `git bundle` described below.

Now you and your collaborators continue to work the manuscript. After some time, and a number of commits, it's time to share your changes with the others. The first step is to identify the commits you want to send. The commits can easily be identified by running `git log`. Suppose you have made 3 commits since you last distributed your changes:

```
$ mkdir patches/
$ git format-patch -3
```

This creates 3 patch files in the current directory, which now can be attached and sent to your collaborators using your favourite mailer. Alternatively you can use `git send-email` to do the job.

```
$ git send-email --subject '[PATCH] my latest changes' --to foo@bar.org \
  --cc raboof@foobar.edu *.patch
$ rm *.patch
```

[6]If the file is very big it's probably better to provide a (hidden) link to your home page, as many mail servers will not accept excessively large files

`git send-mail` can also be configured using `git configure` to avoid having to write the long command line every time.

When you receive changes from your collaborators by e-mail, just save the mail(s) in your project directory and apply the changes:

```
$ git am --3way mailfile(s)
$ rm mailfile(s)
```

It might be a good idea to create and switch to a temporary branch before applying the patches, since this gives you a better possibility to inspect the changes before merging them with your main branch. Obviously, if there is a conflict it has to be resolved like normal. When the conflict has been resolved, you can continue the merge with `git am --continue`.

5.0.1 Using bundles

Alternatively you can use `git bundle` for off-line transport. Instead of creating an archive, you can create an initial bundle with

```
$ git bundle create manus.bndl master
$ git tag -f sent-manus master
# second command marks when we send bundle
```

The receiving side would do

```
$ git bundle verify manus.bndl
$ git fetch manus.bndl master:from-him/master
```

If you want to send changes since last bundle, do

```
$ git bundle create manus.bndl sent-manus..master
$ git tag -f sent-manus master
# second command marks when we send bundle
```

The receiving side does the same as before.

6 Advanced git

6.1 Configuring git

Git uses a number of config files, system wide, per user and per repository (see the git config man page for more info). As a minimum you should set the following options:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "my@email.com"
```

These options are written in /.gitconfig, and are used by git to ensure that your commit messages are sensible, since user names and mail addresses are not necessarily set properly on all machines. In addition you might want to enable the following options as well:

```
$ git config --global color.branch auto
$ git config --global color.status auto
$ git config --global color.diff false
```

If you want beautifully colored diffs, you probably need to add the ’-R’ flag to the `LESS` environment variable, and change ’false’ to ’auto’.

If you want to use some external (graphical) merge tool to resolve conflicts:

```
$ git config --global merge.tool meld
```

`meld` is a fantastic merge tool, and I strongly suggest you have a look at it. Other possibilities are `kdiff3` and `xxdiff`.

Git has many tunable bells and whistles. Please refer to the git configure man page for a more complete listing of configurable options.

6.2 Resolving conflicts

Every now and then you end up in a situation where files have overlapping or incompatible changes. This will be flagged as a conflict by git, and you will have to resolve the conflict before you can proceed. When you get a conflict, git will insert markers in the file showing where the conflict occurred. For example, working on branch foobar, you merge with changes on the master branch and get a conflict. The problematic section in the offending file looks like this:

```
<<<<<<< HEAD:foobar
This is the stuff I have in my working copy in branch foobar...
=======
This is what master currently looks like. Now you need to edit the file, pick
the part you want to retain and remove all the markers.
>>>>>>> master:foobar
```

After you have resolved the conflict in your favorite editor, save and recommit:

```
$ git add <file>
$ git commit
```

A more convenient way to handle conflicting merges is to configure `git mergetool` to launch your favourite diff/merge tool:

```
# this only needs to be configured once
$ git configure --global merge.tool meld
$ git mergetool
$ git commmit
```

If you are not familiar with the general purpose graphical diff- and merge tool `meld` I warmly recommend that you familiarise yourself with this excellent piece of software!

6.3 Specifying revisions

Being able to specify revisions is important whenever we want to in any way access older revisions. Git has a number of different ways of specifying a certain revision, some which we have seen already:

1. A branch name

2. A tag

3. The symbolic name HEAD
4. A revision relative to a branch, tag or HEAD
5. The SHA1 hash of a commit

The two first forms are pretty obvious, but the others need a bit of explanation. The symbolic name HEAD will always points to current checked out branch. HEAD is mostly useful to specify older revisions relative to it. There are a number of ways to specify a relative revision, e.g to specify the parent revision (i.e. one below HEAD)

```
git show HEAD~1
```

The general syntax is

```
git show <HEAD|tag|branch|SHA1>~N
```

For more details see the `git-rev-parse` man page, specifically the section "Specifying revisions".

Every commit (every object in fact) is internally identified by a cryptography-strength one-way SHA1 hash consisting of 40 hexdigits, which *uniquely* identifies any commit (or file). These hexdigits can be found by running

```
$ git log
commit 3ff678cc8abc29db9cb33ec9ca4e468496cb8063
Author: Jonas Juselius <jonas@iki.fi>
Date:   Fri Nov 16 12:47:47 2007 +0100

    Updated .pdf version.

...
$
```

where the first line of every log entry starts with the commit label and the hexdigit identifying that particular revision.

Wherever a revision is needed one can give the SHA1 hash to exactly specify the revision (in fact, the first 6-8 hexdigits are usually enough)

```
git checkout bed6ba53
```

Sometimes it's practical to be able to specify a revision range to limit the output

```
git diff HEAD~4..HEAD~1
```

6.4 Cherry picking

There are lots nice tricks we can play with branches. Suppose you want to try out some idea, but you don't know exactly how it will work out. Create a new branch from your current working branch and check it out:

```
$ git branch foobar
$ git checkout foobar
```

Now work hard and commit often. If it turns out that everything is good, and you want to keep all changes, merge them with your working branch and push them to the master:

```
$ git checkout work
$ git merge foobar
$ git push origin work:myuserid
```

Now you can delete the foobar branch for ever and all times

```
$ git branch -d foobar
```

If, on the other hand it turns out that you had a crackpot idea, and you don't want to see any of it anymore, delete the branch and all changes, commits and everything on the branch will be gone for ever! No trace of it. But what if there are partially useful changes to foobar that you want to keep, before discarding the rest? Well that's when small commits are useful because you can cherry pick! Checkout your work branch, fire up gitk on foobar, and find the commits you like to keep. Every commit is identified by a long SHA1 hash (a long sequence of numbers and letters). Now cherry pick the commits you want into the work branch:

```
$ git checkout work
$ gitk foobar &
# find commit, copy the SHA1 hash with the mouse
$ git cherry-pick SHA1
```

Repeat the cherry pick as many times you like. As you see, git gives a lot of flexibility by using branches. Just be a bit careful in the beginning not to make a mess and lose track of what you are doing.

6.5 Stashing

Sometimes when you are working on a branch you temporarily need to switch to another branch to test something, or maybe fix a bug. In a situation like this you cannot just checkout the other branch, since then all your local changes would get lost (don't worry, git will not allow you to do this). However, you might not want to commit your changes either, since they are not ready or complete. In situations like this git allows you to temporarily commit your changes in a "stash". Running `git stash` saves your latest changes and resets the current branch to it's latest checked in state (the HEAD). When you are ready to continue working you just apply changes in the stash.

```
$ git stash save
$ git stash list
stash@{0}: WIP on mybranch: ad6d0aa... foo
$ git checkout other branch
...
$ git checkout mybranch
$ git stash apply
$ git stash clear
```

6.6 Repository maintenance

When git was conceived, it was based on a very simple scheme for storing revisions to files in the repository. Instead of actually figuring out how files changed between revisions and just storing the differences, git just stored a (compressed) copy of the whole file! This is obviously quite simple and efficient, but very wasteful in terms of storage space. It also becomes inefficient as the number of files in the repository grows.

Modern versions of git still retains this simple storage scheme by default. This means that as your repository evolves with time it will grow substantially in size. Fortunately git provides commands to convert the individual objects in the database into a *pack* file, which stores only the differences between revisions. Once the pack has been generated, all the old objects are unnecessary and can be pruned:

```
$ git repack
$ git prune
```

In fact, git has a simpler command which will do this in one go, and perform additional optimisations on the repository. To "garbage collect" and fully optimise your repository run

```
$ git gc --prune --aggressive
```

6.7 Exporting a repository

Sometimes you want to package your source, e.g. for an official release, but you certainly don't want to include the whole repository in the package. One way to do this is to simply make a copy of the repository, checkout the right branch, clean out all generated and unnecessary files, delete the `.git` directory and make a tar file. This process is much simplified using the `git archive` which will create a .tar or .zip file of the wanted revision on the fly

```
# to create a uncompressed archive
$ git archive --prefix=myprog-1.42/ HEAD >../myprog_1.42.tar
# compressed archive are also trivial
$ git archive --prefix=myprog-1.42/ HEAD |gzip -c >../myprog_1.42.tgz
```

This creates an archive of the latest version (HEAD) on the current branch. You can specify any branch specifier or tag you like to export some other version. Please note the trailing '/' in the prefix, without it you will get a bit of a surprise…

6.8 Finding bugs

Everyone doing software development either alone, or in a group have been in the situation where a bug is suddenly found, with very little clue when it has been introduced, much less where it might be. It can be very tedious to go back and figure out where and when the bug was introduced. Fortunately git has a very clever mechanism to aid us in the process, using the command `git bisect`. The way it work is by tagging a starting revision as bad, and then tag some *known, working* revision as good. `git bisect` will checkout a new revision, and then you compile, test and mark it as either good or bad. A few of these cycles, and the offending revision is found! Here is the process:

```
$ git bisect start
$ git bisect bad # current rev is bad
$ git bisect good version-1.21 # tagged version 1.21 works, guaranteed!
# now git will checkout a new revision
$ make; test.sh
# bad?
$ git bisect bad
$ make; test.sh
...
```

In fact, git will allow you to automate the whole process! If your test script can determine if a version is working or not, then just let your script return 0 if the revision is working and 1 it's bad, and run

```
$ git bisect start
$ git bisect run ./test.sh
```

So, now that you have found the offending revision you probably want to go back to the latest revision and start debugging properly. To do this just execute

```
$ git bisect reset
```

6.9 Undoing commits and resetting

Sometimes we screw up. It's embarrassing. And we don't want anybody to know about it. Like committing something we should not have, or even worse, misspelling a commit message. Whatever. Or maybe we want to undo a merge or a pull which screwed up our repository, causing tons of conflicts or breaking things badly. There are two commands for undoing commits, `git revert` and `git reset`. These two differ in the sense that `git revert` will undo changes in a controlled, and in itself reversible manner, whereas `git reset` resets the branch to a specified state without leaving any trace in the history. Even if you reset, it's still possible to recover the "lost" commits through the reflog facility (see below).

Suppose you have been working for a while, committing regularly, and after a while you realise that everything you have done the last few commits is utter garbage, and you want to go back 3 revisions and start over:

```
$ git revert HEAD~3
```

This resets your working copy to the specified revision, and commits the changes the revert introduced. Hence, you can go back when you realise that the garbage you had produced actually was gold after all.

Another scenario is when you realise you have an embarrassing typo in a commit message, or that you forgot to include a file in the commit, or committed too many files. Obviously you can always revert, but that's really quite unnecessary and does not really achieve what you want. In this situation you would do a soft reset, which means that the HEAD revision is reset to point to another resent revision, but your *working copy* is left intact. To undo a commit in this manner you would do the following:

```
$ git commit file1 file2
# uups, let's undo the commit message, and edit file1 and add file3
$ vim file1
$ git add file3
$ git commit --amend file1 file2 file3
```

If you really, really want to reset the state of both the repository and your working copy you need to do a hard reset. Be warned, a hard reset throws away all commits and all changes to your files up to the specified revision forever. There is no way of getting the information back again. To do a hard reset and go two revisions back, and at the same time also reset your working copy to that state:

```
$ git reset --hard HEAD~2
```

If you for some reason suddenly realise that the reset was a mistake, and you want to go back to where you were before the reset, you can reset back to the reflog HEAD:

```
$ git reset --hard HEAD@{1}
```

6.10 Working with multiple remotes

Git provides a lot more flexibility than CVS. Due to it's distributed nature it's actually possible to have multiple "master" servers, or more accurately, multiple remote repositories. A typical situation when it can be desirable with many remotes, is when the central master server has limited push access, e.g. to ensure that only working code is distributed to other developers. In a situation like this one can set up multiple remotes to point to the repositories of the people one is collaborating closely with.

The default remote repository is called 'origin', but you can attach as many remotes as you like. To register a new remote repository, simply run

```
$ git remote add <name> <myserver>:/path/to/git/proj.git
```

Now when you fetch, pull and push use your new remote-tag `<name>` instead of `origin` to the corresponding command. The new development cycle will typically be something like

```
$ git pull origin master
# do some work
$ git push myremote mybranch:mybranch
# or to push all branches automatically
$ git push --all myremote
```

A good solution might be to have a central storage area, with a true "master" repository, to which only a few people have write access to. The other developers can create their own personal repositories in the storage area, and have full access to that repository. By setting up the `gitweb` interface it becomes very easy to track what everybody is doing through the web.

To set up your own sub-master repository follow these steps:

```
$ ssh <myserver>
$ cd /path/to/repos
$ mkdir $USER
$ git clone -s -l --bare proj.git $USER/proj.git
$ cd $USER/proj.git
$ git branch -a
# remove any branches and tags you do not care about
$ git branch -D <branch branch...>
$ git tag -d <tag tag...>
# edit description (for gitweb) to your liking
$ vi description
```

The -s and -l options to `git clone` will cause git to set up the repository to use the master's object database as much as possible, so that it will take up very little space.