Git/Introduction

Here, we will introduce the simplest git commands: creating a new repository, adding and committing files, removing files, reverting bad commits, throwing away uncommitted changes, and viewing a file as of a certain commit.

Creating a git repository

Creating a new git repository is simple. There are two commands that cover this functionality: git-init, and git-clone. Cloning a pre-existing repository will be covered later. For now, let's create a new repository in a new directory:

```
$ git init myrepo
Initialized empty Git repository in /home/mikelifeguard/myrepo/.git/
```

If you already have a directory you want to turn into a git repository:

```
$ cd $my_preexisting_repo
$ git init
```

Taking the first example, let's look what happened:

```
$ cd myrepo
$ ls -a
.  ..  .git
```

The totality of your repository will be contained within the .git directory. Conversely, some SCMs leave droppings all over your working directory (ex, .svn, .cvs, .acodeic, etc.). Git refrains, and puts all things in a subdirectory of the repository root aptly named .git.

Adding and committing files

Unlike most other VCSs, git doesn't assume you want to commit every modified file. Instead, the user adds the files they wish to commit to a staging area (also known as "the index"). Whatever is in the index is what gets committed. You can check what will be committed with git status or git diff --staged.

To add files use the command git-add.

```
$ nano file.txt
hack hack hack...
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      file.txt
nothing added to commit but untracked files present (use "git add" to track)
```

This shows us that we're using the branch called "master" and that there is a file which git is not tracking. Git helpfully notes that the file can be included in our next commit by doing git add file.txt:

```
$ git add file.txt
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   file.txt
#
```

After adding the file, it is shows as ready to be committed. Let's do that now:

```
$ git commit -m 'My first commit'
[master (root-commit) be8bf6d] My first commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
```

In most cases, you will not want to use the -m 'Commit message' form - instead, leave it off to have $EDITOR opened so you can write a proper commit message. We will describe that next, but in examples, the -m 'Commit message' form will be used so the reader can easily see what is going on.

Excluding files from Git

Often there are files in your workspace that you don't want to add to the repository. For example, `emacs` will write a backup copy of any file you edit with a tilde suffix, like `filename~`. Even though you can manually avoid adding them to the commit, they clutter up the status list.

In order to tell Git to ignore certain files, you can create a file in your workspace called `.gitignore`. Each line in the `.gitignore` file represents a specification (with wildcards) of the files to be ignored. Comments can be added to the file by starting the line with a blank or a `#` character.

For example:

```
# Ignore emacs backup files:
*~

# Ignore everything in the cache directory:
app/cache
```

Good commit messages

Tim Pope [1] writes [2] about what makes a model Git commit message:

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.
```

```
Write your commit message in the present tense: "Fix bug" and not "Fixed
bug."  This convention matches up with commit messages generated by
commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

Let’s start with a few of the reasons why wrapping your commit messages to 72 columns is a good thing.

- Git log doesn’t do any special special wrapping of the commit messages. With the default pager of less -S, this means your paragraphs flow far off the edge of the screen, making them difficult to read. On an 80 column terminal, if we subtract 4 columns for the indent on the left and 4 more for symmetry on the right, we’re left with 72 columns.
- git format-patch --stdout converts a series of commits to a series of emails, using the messages for the message body. Good email netiquette dictates we wrap our plain text emails such that there’s room for a few levels of nested reply indicators without overflow in an 80 column terminal.

Vim users can meet this requirement by installing my vim-git runtime files, or by simply setting the following option in your git commit message file:

```
:set textwidth=72
```

For Textmate, you can adjust the “Wrap Column” option under the view menu, then use ^Q to rewrap paragraphs (be sure there’s a blank line afterwards to avoid mixing in the comments). Here’s a shell command to add 72 to the menu so you don’t have to drag to select each time:

```
$ defaults write com.macromates.textmate OakWrapColumns '( 40, 72, 78 )'
```

More important than the mechanics of formatting the body is the practice of having a subject line. As the example indicates, you should shoot for about 50 characters (though this isn’t a hard maximum) and always, always follow it with a blank line. This first line should be a concise summary of the changes introduced by the commit; if there are any technical details that cannot be expressed in these strict size constraints, put them in the body instead. The subject line is used all over Git, oftentimes in truncated form if too long of a message was used. The following are just a handful of examples of where it ends up:

- git log --pretty=oneline shows a terse history mapping containing the commit id and the summary
- git rebase --interactive provides the summary for each commit in the editor it invokes
- If the config option `merge.summary` is set, the summaries from all merged commits will make their way into the merge commit message
- git shortlog uses summary lines in the changelog-like output it produces
- git format-patch, git send-email, and related tools use it as the subject for emails
- reflogs, a local history accessible with git reflog intended to help you recover from stupid mistakes, get a copy of the summary
- gitk has a column for the summary
- Gitweb and other web interfaces like GitHub [3] use the summary in various places in their user interface

The subject/body distinction may seem unimportant but it’s one of many subtle factors that makes Git history so much more pleasant to work with than Subversion.

Removing files

Let's continue with some more commits, to show you how to remove files:

```
$ echo 'more stuff' >> file.txt
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#      modified:   file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Although git doesn't force the user to commit all modified files, this is a common scenario. As noted in the last line of git status, use git commit -a to commit all modified files without adding them first:

```
$ git commit -a -m 'My second commit'
[master e633787] My second commit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

See the string of random characters in git's output after committing (bolded in the above example)? This is the abbreviation of the identifier git uses to track objects (in this case, a commit object). Each object is hashed using SHA-1, and is referred to by that string. In this case, the full string is `e6337879cbb42a2ddfc1a1602ee785b4bfbde518`, but you usually only need the first 8 characters or so to uniquely identify the object, so that's all git shows. We'll need to use these identifiers later to refer to specific commits.

To remove files, use the "rm" subcommand of git:

```
$ git rm file.txt
rm 'file.txt'
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      deleted:    file.txt
#
$ ls -a
.  ..  .git
```

Note that this deletes the file from your disk. If you only want to remove the file from the git repository but want to leave the file in your working directory, use git rm --cached.

```
$ git commit -m 'Removed file.txt'
[master b7deafa] Removed file.txt
 1 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 file.txt
```

Reverting a commit

To revert a commit, use git revert:

```
$ git revert HEAD
Finished one revert.
[master 47e3b6c] Revert "My second commit"
 1 files changed, 0 insertions(+), 1 deletions(-)
$ ls -a
.  ..  file.txt  .git
```

You can specify any commit instead of HEAD. For example:

The commit before HEAD

git revert HEAD^

The commit five back

git revert HEAD~5

The commit identified by a given hash

git revert e6337879

Throwing away local, uncommitted changes

To throw away your changes and get back to the most recently-committed state:

```
$ git reset --hard HEAD
```

As above, you can specify any other commit:

```
$ git reset --hard e6337879
```

If you only want to reset one file (where you have made some stupid mistake since the last commit), you can use

```
$ git checkout filename
```

This will delete all changes made to that file since the last commit, but leave the other files untouched.

Get a specific version of a file

To get a specific version of a file that was committed, you'll need the hash for that commit. You can find it with git log:

```
$ git log
commit 47e3b6cb6427f8ce0818f5d3a4b2e762b72dbd89
Author: Mike.lifeguard <myemail@example.com>
Date:   Sat Mar 6 22:24:00 2010 -0400

    Revert "My second commit"

    This reverts commit e6337879cbb42a2ddfc1a1602ee785b4bfbde518.

commit e6337879cbb42a2ddfc1a1602ee785b4bfbde518
Author: Mike.lifeguard <myemail@example.com>
```

```
Date:   Sat Mar 6 22:17:20 2010 -0400

    My second commit

commit be8bf6da4db2ea32c10c74c7d6f366be114d18f0
Author: Mike.lifeguard <myemail@example.com>
Date:   Sat Mar 6 22:11:57 2010 -0400

    My first commit
```

Then, you can use git show:

```
$ git show e6337879cbb42a2ddfc1a1602ee785b4bfbde518:file.txt
hack hack hack...
more stuff
```

Conclusion

You now know how to create a new repository, or turn your source tree into a git repostiory. You can add and commit files, and you can revert bad commits. You can remove files, view the state of a file in a certain commit, and you can throw away your uncommitted changes.

Next, we will look at the history of a git project on the command line and with some GUI tools, and learn about git's powerful branching model.

References

[1] http://tpo.pe/
[2] http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
[3] http://github.com

Git/Branching & merging

Branching is supported in most VCSs. Branching can be an expensive and time consuming operation like in centralized systems that require making a copy of the whole source tree. Branching can also be given a high priority like **D**VCSs do to make it faster. In git, creating branches is easy, and *very* fast.

Branching

View your branches

Use git branch with nothing else to see what branches your repository has:

```
$ git branch
* master
```

The branch called "master" is the default main line of development. You can rename it if you want, but it is customary to use the default. When you commit some changes, those changes are added to the branch you have checked out - in this case, master.

Create new branches

Let's create a new branch we can use for development - call it "dev":

```
$ git branch dev
$ git branch
  dev
* master
```

This only creates the new branch, it leaves your current HEAD where you remain. You can see from the * that the master branch is still what you have checked out. You can now use git checkout dev to switch to the new branch.

Alternatively, you can create a new branch and check it out all at once with

```
$ git checkout -b newbranch
```

Delete a branch

To delete the current branch, again use *git-branch*, but this time send the -d argument.

```
$ git branch -d <name>
```

If the branch hasn't been merged into master, then this will fail:

```
$ git branch -d foo
error: The branch 'foo' is not a strict subset of your current HEAD.
If you are sure you want to delete it, run 'git branch -D foo'.
```

Git's complaint saves you from possibly losing your work in the branch. If you are still sure you want to delete the branch, use git branch **-D** <name> instead.

Pushing a branch to a remote repository

When you create a local branch, it won't automatically be kept in sync with the server. Unlike branches obtained by pulling from the server, simply calling `git push` isn't enough to get your branch pushed to the server. Instead, you have to explicitly tell git to push the branch, and which server to push it to:

```
$ git push origin <branch_name>
```

Deleting a branch from the remote repository

To delete a branch that has been pushed to a remote server, use the following command:

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

This syntax isn't intuitive, but what's going on here is you're issuing a command of the form:

```
$ git push origin <local_branch>:<remote_branch>
```

and giving an empty branch in the `<local_branch>` position, meaning to overwrite the branch with nothing.

Merging

Branching is the central concept of DVCS, but without good merging support, branches would be of little use.

```
git merge myBranch
```

This command merges the given branch into the current branch. If the current branch is a direct ancestor of the given branch, a *fast-forward* merge occurs, and the current branch head is redirected to point at the new branch. In other cases, a *merge commit* is recorded that has both the previous commit and the given branch tip as parents. If there are any conflicts during the merge, it will be necessary to resolve them by hand before the merge commit is recorded.