ClearVision

Presents

Git 201

Advanced

What you've always wanted to know about Git but were too afraid to ask.

clearvision-cm.com/git

Contents

Introduction 1

The Git command line 2

How to undo a Git commit 7

Git commit messages 8

What to do when you commit to the wrong Git branch 11

Git tracking branches 14

Welcome to Git 201!

Written by Visionaries that use Git every day.

Git 201 covers:

- Git commits.
- Commit messages.
- The Git command line.
- What to do when something goes wrong!

For a guide to the basics, check out our Git 101 white paper.

The Git Command Line

Git is first and foremost a command line tool.

Graphical interfaces still have some catching up to do to match the functionality and power that a knowledgeable user can achieve on the command line.

That being said, the command line isn't always the most welcoming of UIs! Let's explore what the command line can open up for you.

Command line tips

Graphics without graphics

Being able to visualise some of the constructs within Git can help a lot, and some Git commands include an element of basic visualisation without any graphics.

Try, for example:

```
git show-branch
```

You may get an output similar to this:

```
Philip Armour@CLEARVISION-PC ~/parmour-project (develop)
$ git show-branch
! [bugfix1] bugfix1: fix to error 23
 * [develop] Merge branch 'featdev_02' into develop
  ! [featdev_02] featdev_02: commit 1
   ! [master] Second part of 6001 fix
----
!     [bugfix1] bugfix1: fix to error 23
 -    [develop] Merge branch 'featdev_02' into develop
 *+   [featdev_02] featdev_02: commit 1
 *    [develop^] develop: minor fix for issue 34
```

What does this tell us?

The output above the ---- tells us that we have 4 local branches and that the current branch is called **develop** (it has a * rather than a !). Below the ---- are recent commit messages.

To see which of these commits are in a given local branch, you look in the relevant column (below the * or !). In the illustration above for example, the commit **[featdev_02] featdev_02: commit 1** is present in both the **develop** and the **featdev_02** branches.

Another good example of command-line 'graphics' is when you use the --graph option on a Git log to show the branches reachable from your current branch:

```
cassius@localhost:~/parmour-project (develop) $git log --oneline --graph --decorate -10
*   e7f1dad (HEAD, origin/develop, develop) Merge branch 'dev_feat_1' into develop
|\
| * e18a0c2 (origin/dev_feat_1, dev_feat_1) dev_feat_1 commit 2
| * 5bf60b5 dev_feat_1 commit 1
* | 1303df5 develop change
* | 7f67f78 redchange_a commit 1
|/
* 2620486 Gerrit column data
* 20ee9ae test4444 commit
* d021cd2 Change the wood_turtle file - 3
*   de8c1ca Merge "subbran222 work" into develop
|\
| * d7f3e09 subbran222 work
```

Customize your commands

If you create a new script and name it **git-<script name>** and make sure it is located somewhere in your **PATH**, it will run like a Git command.

Let's use the example **git-newscript**.

You can then type:

```
git newscript
```

This can be a seamless way of extending and customising Git with scripts tailored to a specific system configuration and workflow.

Along the same lines, you can create aliases for commonly used commands. Reduce the wear on your keyboard! For example:

```
git config --global alias.clog 'log --oneline --graph --decorate -10'
```

Would mean you could simply type:

```
git clog
```

Note, since aliases are added to the Git config, they can be applied at any one of three levels of scope: --system (for any one using that installation of Git), --global (for any repository owned by a given user), and --local: for the current Git repository alone.

Autocompletion

> Partially type your Git command.

> Press tab.

> If there is more than one Git command possible, each possibility will be shown.

> If what you've typed is unambiguous, you'll get the full command.

"I have no doubt that the ways we can use Git via a graphical interface will improve even further in the near future, and lead to real benefits in user experience.

But for tackling those truly interesting and challenging Git problems, I will always want to see that cursor flashing in front of me!"

Philip Armour
Clearvision Technical Consultant

This even works for command options! For example:

```
git log --gra<tab>
```

Will expand to:

```
git log -graph
```

Autocompletion is configured for Git Bash (git-for-windows). For Linux, you need to check that you have a file called **git-completion.bash in /etc/bash_completion.d/**, and run the source command on that file in the session you're using for Git.

Git colours and prompt

Have you been using the Git command line in Monochrome?

Why not try a more colourful experience?

The config command to use is:

```
git config --global color.ui true
```

This **StackExchange post** has some more info on configuring the colours to your exact requirements.

Finally, it can be useful to have Git status information forming part of your Bash prompt. The Bash prompt can be set up to show which branch you are currently working on, saving you time checking.

Again, this is configured for you already in Git Bash. For Linux you can activate it by executing:

```
source /etc/bash_completion.d/git-prompt.sh
```

```
PS1='\u@\h:\w$(__git_ps1 " (%s)") \$ '
```

More details about what can be done with prompt information and how, are available in the comments of the **git-prompt.sh file.**

How to undo a Git commit

What do you do when you commit something that you never intended to commit?

Well, you undo it:

```
git reset HEAD~1
```

What this means is that you reset the HEAD of the current branch by one commit back in history. You can even undo multiple commits in one go, all you have to do is increase the number at the end.

All the changes that the reverted commit contained are now local changes in your workspace. You can either commit these again, e.g. to another branch or to include additional changes, or you can throw them away with **git reset --hard.**

If you pushed your commits before you started undoing them, there are a couple of things that you need to bear in mind:

> You need to push your undone commits with the **--force** option. Please note, this will also undo any commits on the target branch made by other users, so take care when pushing with **--force** and make sure there are no other commits on the branch that follow yours.

> Also, if you pull (rather than push) at this point, then your undone commits will be undone, i.e. you will end up where you started.

Git Commit Messages

A Good Commit Message

Good commit messages are arguably just as important as writing meaningful comments within your source code.

Why?

Within Git, updates to code normally span multiple source files, meaning commit messages have a more 'aerial view' perspective than the 'ground-level' comments in source code.

> They tell the story of how the software has evolved.

> They can help greatly when your boss asks if bugfix ABC went into customer branch DEF, for example, by using **git log --grep="ABC"**.

> They represent a form of passive *'to whom it may concern'* collaboration, which results in hard-to-quantify, but tangible and definite long-term benefits.

Get the most from Git commit messages...

A template for good messages

When learning Git, we often begin by messing around with our personal 'sandbox' repositories from the command line. During this time, it's understandable when we create quick commits like:

```
git commit -m "change to some code"
```

But don't get into this habit when modifying real production code!

It is better to simply use **git commit** and let Git invoke our default editor for us.

When we do this, Git brings up a default message with some advisory comments, which can be tailored to specific requirements using the commit template variable in our Git config.

```
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.

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
```

Be careful what you commit...

It's far from trivial to rectify a situation where historical Git commit messages have undesired or sensitive content. It requires careful use of tools like **git filter-branch** to completely rebuild the DAG.

Given the potential difficulty of changing old commit messages, what happens if your commit and message turn out to be incorrect?

The Git "sticky note"

Luckily, there is a lightweight solution. In cases where you need to associate some extra information with an old commit message, you can turn to **git notes.**

You can use this feature to add further information to a historical commit by typing:

```
git notes add <commit ID>
```

Opens the default editor. To add to this later, use:

```
git notes append <commit ID>
```

The key takeaway? Think of future Developers when you write your commit messages!

Commit to the wrong branch?

What to do when you commit to the wrong Git branch

It's a sunny day, you're having a great time coding, and being the good boy/girl that you are, you commit regularly.

And then it hits you - several commits, all to the wrong branch. What now?

Well, this is Git we are talking about. So the good news is that it's relatively straightforward!

Let's say you committed to master, and you meant to commit to a new branch called **myfeature.** There are two things that need fixing. First of all, you need to revert master back to where it originally was. And secondly, you need to get your changes on that new branch.

The options:

> Undo commits on master, checkout new branch **myfeature**, and commit all your changes as one commit.

> Modify refs manually for master to point back to where you were, and for your new branch ref to.

The first solution is simple and easy to apply. There's a relatively small margin for error. The downside is that you don't get your original commit messages on the new branch, and all changes are applied as a single commit with a new commit message, etc.

The second solution is a little more involved, but it means all commits are passed over to the new branch (though it is slightly easier to get wrong).

Undo and commit to new branch

Make sure you are on the branch to which you have been committing. Use git log to check how many commits you want to roll back. Then, undo the commits with **git reset HEAD-N** where "N" is the number of commits you want to undo.

For example, to undo one commit:

```
git reset HEAD~1
```

Then, create a new branch and check it out in one go and add and commit your changes again.

```
git checkout -b newbranch
```

```
git add -A
```

```
git commit -m "Committed on new branch"
```

Be careful with the `add -A`! You may be adding unrelated, uncommitted files and directories. Have a look with `git status` before you commit.

Move commits to the other branch

Follow these steps carefully (there is slightly more margin for error with this method).

The first step is to make a note of the commit id you want to make the head of the new branch.

```
git log
```

Copy the commit id somewhere safe. Then, reset your current branch back by one commit (or however many commits you need to go back):

```
git reset --hard HEAD~1
```

And the final step: move the commits that follow to the new branch:

```
git checkout -b newbranch
```

```
git reset --hard < commit_id >
```

And it's done! Time to push both branches (with **--force** if needed, i.e. if you had previously pushed the changes). However, as always, when using **--force** make sure there are no other commits that follow yours as they would be undone.

Git Tracking Branches

The way we talk about a tracking branch within Git can become confusing, because there are two different concepts.

Context 1: The remote tracking branch

Firstly, a 'remote tracking branch' is not really remote - which doesn't really help!

It exists in your local Git repo under **.git/refs/remotes,** for example:

```
.git/refs/remotes/origin/fix22
```

It is also not the kind of branch we usually deal with in Git - since we cannot work on it. But it is a branch in the Git sense: a thing which points to a commit.

So just imagine there is a branch called **fix22** for the repo on the remote machine (our origin), and we have not created or ever edited this branch locally.

When we communicate with the remote machine, e.g. via **git fetch origin,** a local file called **.git/refs/remotes/origin/fix22** will be updated or created. The file contains the latest commit of this branch on the remote repo.

So a **remote tracking branch** records the state of a branch in a remote repo, but it cannot be worked on.

Context 2: The local tracking branch

Unlike the remote tracking branch, we can work on this branch!

It is a local branch which appears when you look at the files under **.git/refs/heads**.

Again, imagine there is a branch called **fix22** on the remote repo (our origin). If we want to work on this branch we can create a local tracking branch as follows:

```
git checkout --track -b fix22 origin/fix22
```

Or in recent versions of Git, you can simply type:

```
git checkout fix22
```

When we communicate with the remote machine, e.g. via **git fetch origin,** Git will know that this local tracking branch is associated with the branch on the remote machine, and will do the right thing when we push and pull changes.

To see what **local tracking branches** are configured in a repo and how they map to branches on the remote machine, you can execute:

```
git checkout fix22
```

With a better understanding of tracking branches, we can now look at the similarities and differences between git pull and git fetch.

Similarities between 'git pull origin' and 'git fetch origin'

Both the git pull origin and git fetch origin commands update the remote tracking branches.

This means that the files under .git/refs/remotes/origin in your local repo get updated. If new branches have appeared on the remote machine, then new files appear.

Both commands will bring the latest commits from the remote machine, which you do not already have in your repo. In this context, the behaviour of fetch and pull is identical for branches that are not currently the active branch (checked out).

Differences between 'git pull origin' and 'git fetch origin'

For **git pull origin** to work, a local tracking branch must be active (checked out).

git pull origin will, in addition to the **git fetch** operations, do an automatic merge or fast-forward of the active (local tracking) branch.

So essentially, **git pull** does everything that **git fetch** does – plus a little more on your active branch.

You've reached the end of Git 201 - but the learning doesn't have to stop here!

Want to go further with Git?

Clearvision's Git training is available for teams of all sizes, of all abilities. From our Basics and Advanced Courses to SmartGit and Gerrit Courses, Clearvision's training is hands on and interactive.

Take your teams on a journey, with a pre-training skills gap analysis and post-training refresher material!

Explore training courses

clearvision-cm.com
enquiries@clearvision-cm.com
@clearvisioncm
/company/clearvision

Clearvision