GIT CHEAT SHEET - QUICK COMMAND REFERENCE

Quick reference

Here are the list of commands we use quite often.

**remote** means remote repository uri,  
for example, https://github.com/User/myrepo.git

Clone & branches

1. To clone a remote git repository with depth = 1 and specifying my project name:
2. $ git clone --depth=1 remote myproject
3. To clone a remote git repository with specific branch (eg. master):
4. $ git clone -b master remote
5. To see the branches after cloned all branches (cloned w/o specifying any):
6. $ git branch -a
7. \* dev
8. \* master
9. remotes/origin/HEAD -> origin/master
10. remotes/origin/dev
11. remotes/origin/master
12. Switching branches (**checkout**):
13. $ git checkout dev
14. Switched to branch 'dev'
15. Create a branch off master base (**checkout -b <new branch> <base off branch>**):
16. $ git checkout -b hotfix master
17. Switched to a new branch 'hotfix'

About remote repository

1. To setup local repo with a remote repositories.  
   First, we need to use **git init** and then use **git remote add remote-name remote-url**:
2. $ git init

This will create **.git** directory into our current working directory. Now we want to add the remote repo:

$ git remote add origin https://github.com/User/repo.git

Here the 4th one (**origin**), we can name it whatever we want to. But **origin** is a kind of convention.

1. To see what are the remote repositories:
2. $ git remote -v
3. origin https://github.com/User/repo.git (fetch)
4. origin https://github.com/User/repo.git (push)

If we want to use the same name, we can drop our project name which is the last argument.

1. To switch remote repositories from repo1 to repo2:  
   This is my current remote:
2. $ git remote -v
3. origin https://github.com/User/repo1.git (fetch)
4. origin https://github.com/User/repo1.git (push)

Here it is:

$ git remote set-url origin https://github.com/repo2.git

Note that we overwrote (replaced with a new remote) an existing one.  
If we wanted to keep the old one, we could have used **add** instead of **set-url**:

$ git remote add origin2 https://github.com/repo2.git

Then, we have two remote repos:

$ git remote -v

origin https://github.com/User/repo1.git (fetch)

origin https://github.com/User/repo1.git (push)

origin2 https://github.com/User/repo2.git (fetch)

origin2 https://github.com/User/repo2.git (push)

Staging

There are couple of options when we do stage files.

1. Staging only "modified (including deleted)":
2. $ git add -u

or

$ git add --update

If we want to do commit at one shot:

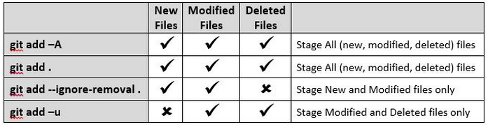
$ git commit -a

1. Staging all (modified + new + deleted):  
   "git add -A" is equivalent to "git add --all"
2. $ git add -A

or

$ git add --all

As we can see from the table below, as far as Git 2, there is not difference "git add -A" and "git add ."



Source : [Difference between "git add -A" and "git add ."](http://stackoverflow.com/questions/572549/difference-between-git-add-a-and-git-add/16162511#16162511)

Un-staging

To un-stage a file, we can use **HEAD** with a file name. For example, to unstage a README:

$ git status

Changes to be committed:

new file: 1.rb

modified: README

Let's unstage the README:

$ git reset HEAD README

Unstaged changes after reset:

M README

$ git status

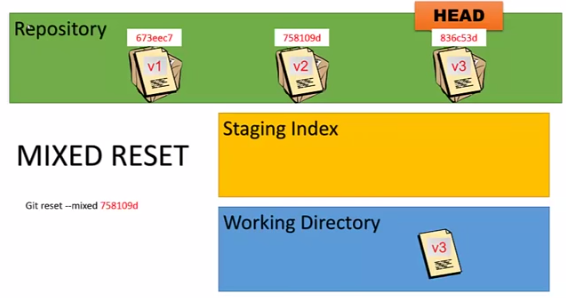
Changes to be committed:

new file: 1.rb

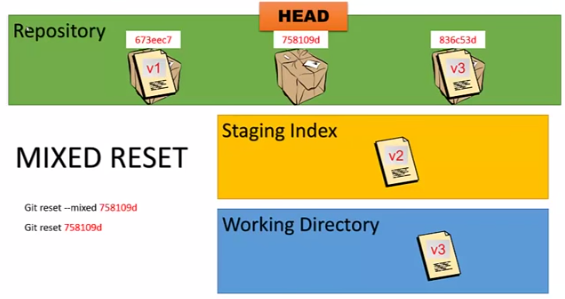
Changes not staged for commit:

modified: README

**Before**:



**After**:



Picture credit: [Learn Git 20: The Mixed Reset](https://www.youtube.com/watch?v=A3s8KxgDYRk)

git stash - saves changes working directory into "stash"

Sometimes, we want to switch to other branch but we have uncommitted changes. In that case, we can issue **git stash**, which saves our working directory and index to "stash". Basically, it restores our working directory and index to the most recent commit. We can then work on other branches, make commits, etc. When we're ready to get back to where we were, we may want to type **git stash pop**.

git reset --hard to clean-up working directory :

If we have lots of files to cleanup (undo any lines of code that are not working), we use **reset --hard**. This will put the files back to the original state before staging:

$ git status

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: 2.rb

new file: 3.rb

new file: 4.rb

...

$ git reset --hard

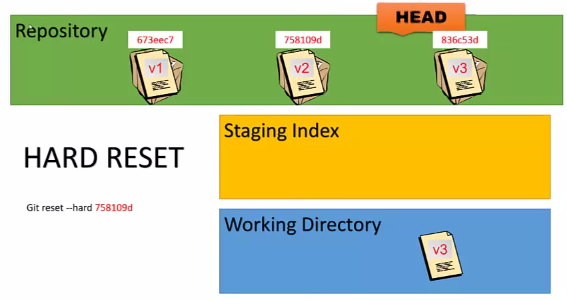
$ git status

On branch master

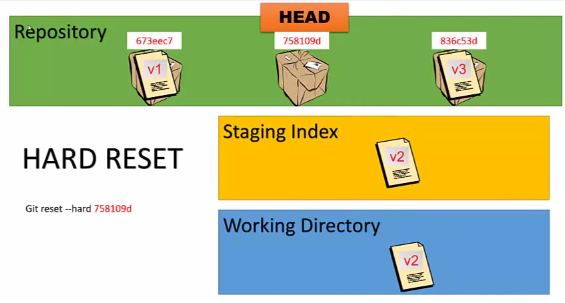
nothing to commit, working directory clean

With **reset --hard**, if a file was newly created and added, it will be gone (deleted) from our working directory while a file was modified, it'll have the content before any modification.

**Before**:



**After**:



Picture credit: [Learn Git 21: The Hard Reset](https://www.youtube.com/watch?v=T82UZAd47K0)

This **reset --hard** can be used with id (SHA hash):

$ git log --graph --oneline

\* 35e529b README update

\* 94b269f ruby script

\* 5de1450 README added

\* 5dcafb2 C3 commit

\* 316f4dd C2 commit

\* 2b61646 initial commit

$ git reset --hard 5dcafb2

HEAD is now at 5dcafb2 C3 commit

$ git log --graph --oneline --all

\* 5dcafb2 C3 commit

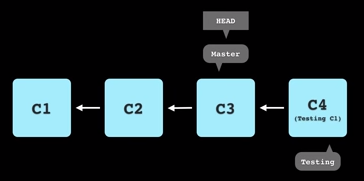
\* 316f4dd C2 commit

\* 2b61646 initial commit

Fast-forward merge - the simplest merge

We created a new branch (**hotfix**) base off a **master** branch, and now we want to merge it back to "master" branch after worked on the new branch.

**Before**:



$ git branch

dev

\* hotfix

master

Since no changes have been made to the "master", when we merge our "hotfix" back to "master", internally git is doing so-called **fast-forward merge**: just moving forward the **HEAD** and **Master** pointers to a new commit:

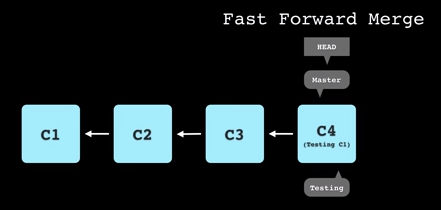
$ git checkout master

$ git merge hotfix

Updating 5dcafb2..85d564c

Fast-forward

**After**:



Picture credit: [Version Control with Git - 05 - Branching](https://www.youtube.com/watch?v=z1cs8ams3xM)

Rebase - re-writing commit history

We can see how it works from:

$ git rebase --help

**Before**:

$ git log --graph --all --oneline

\* eae656e C5 commit on dev

\* 0dca29b C3 commit on dev

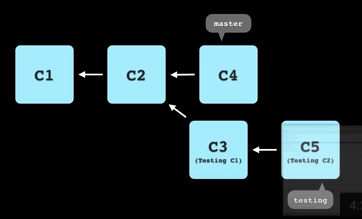
| \* 63c73eb C4 commit on master

|/

\* ab8a9f6 C2 commit on master

\* db98840 C1 commit on master

**Before rebase**:



To rebase, issue the following command on a testing branch (**dev**):

$ git branch

\* dev

master

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: C3 commit on dev

Applying: C5 commit on dev

$ git log --graph --all --oneline

\* c6b00e2 C5 commit on dev

\* ae2ce80 C3 commit on dev

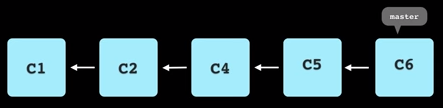
\* 63c73eb C4 commit on master

\* ab8a9f6 C2 commit on master

\* db98840 C1 commit on master

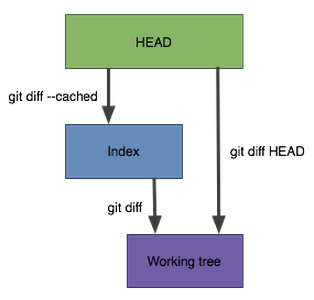
Note that while git is doing rebase, it applies each commit one by one, and it may stop doing rebase if git finds any conflicts.

**After rebase**:



Picture credit: [Version Control with Git - 07 - Rebase](https://www.youtube.com/watch?v=cSf8cO0WB4o)

Diff



Just for demonstration purpose, I put my file (**a.txt**) in several stage with a text of that stage name:

1. A file on working directory (working tree) - "My Working directory file"
2. A file in staging area - "My Staged file"
3. A file already committed - "My HEAD file"
4. A file already pushed to remote - "My Remote file"

So, when we issue a diff command, we can see the diff more clearly.

1. After we staged a file, and want to see the diff of the file from the one that's already in the HEAD (local repo):
2. $ git diff --staged a.txt
3. -My HEAD file
4. +My Staged file
5. Worked on a file that's on our working directory (or working tree), and want to see the diff from the one already staged:
6. $ git diff a.txt
7. -My Staged file
8. +My Working directory file
9. To see the diff of my file on working tree with the one already in the HEAD
10. $ git diff HEAD a.txt
11. -My HEAD file
12. +My Working directory file
13. To see the diff of my file on working tree with the remote one.  
    To be more clear, we may want to know how the remote looks like:
14. $ git remote -v
15. origin https://github.com/User/repo.git (fetch)
16. origin https://github.com/User/repo.git (push)

Now, let's do diff:

$ git diff origin/master:a.txt a.txt

-My remote file

+My Working directory file

We can switch the positions

$ git diff HEAD:a.txt origin/master:a.txt

-My HEAD file

+My remote file

Log & Display

To see tree like git history:

$ git log --graph --oneline --all

To see the changes by commits, we can use:

1. **git show** with **HEAD** for the most recent commit:

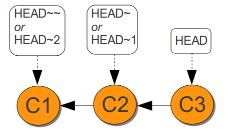
$ git show

Or:

$ git show HEAD

1. To go back further, we can use **HEAD~**, or one more backwards: **HEAD~~**:

$ git show HEAD~3



Pull request

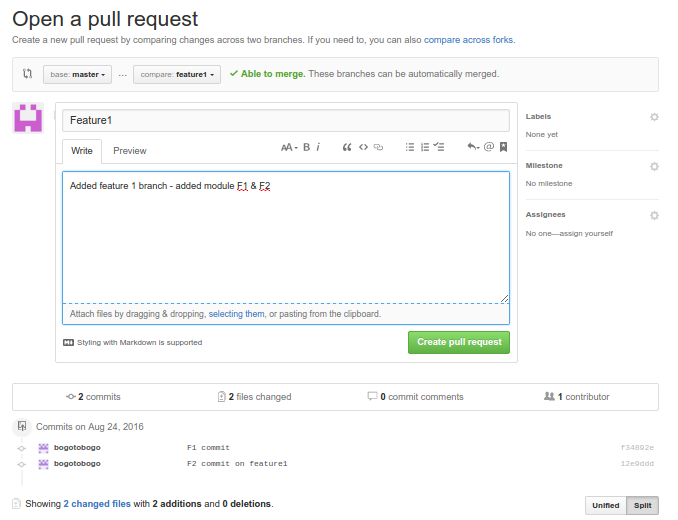
After merge our **feature1** branch to **master** branch, we want to push **feature1** branch to remote repository:

$ git checkout master

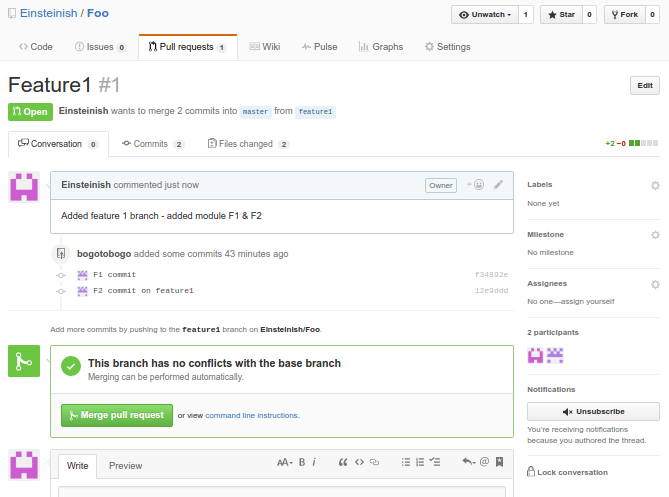
$ git merge feature1

$ git push -u origin master

On a Github page, select "feature1" branch and click "New pull request". Then, the following page will be opened:



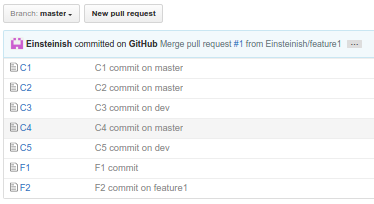
Hit "Create pull request".



Hit "Merge pull request".

Next, after a code review, one of our colleague will merge "feature1" branch to "master" branch by clicking "Confirm merge".

Now, our "master" branch has the new features (F1 & F2 commits):



Resets

The "git reset"s are confusing topic even for seasoned git users. We have 3 types of resets.

It's helpful if we picture it from the top level (indexed - committed), then staged, and then working directory, in that order.

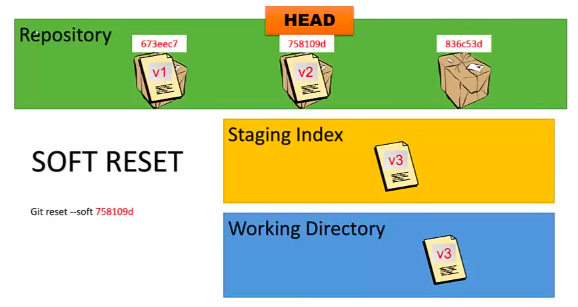
Let's assume we have committed v1/v2/v3.



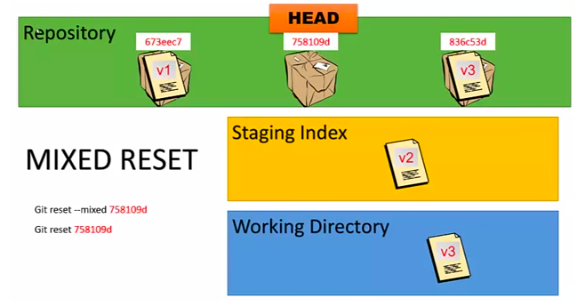
All the 3 types of resets will move back the HEAD to the previous commit (v2).

Starting from soft reset to mixed, and then hard reset.

Soft resets undo the indexed (committed), and put it back to the state before the commit. In other words, the status will be back to staging. So, v3 still in the staging level and it's ready to be committed again.



The mixed resets undo down to the staging. So, v3 is in the working directory:



The hard resets undo all down to working directory, wiping out all. So, v3 is not even in the working directory, gone:

