19 Git Tips For Everyday Use

September 8, 2015 by Alex Kras — 69 Comments

Update: You can now get this post on Kindle.

I've been using git full time for the past 4 years, and I wanted to share the most practical tips that I've learned along the way. Hopefully, it will be useful to somebody out there.

If you are completely new to git, I suggest reading Git Cheat Sheet first. This article is aimed at somebody who has been using git for three months or more.

Table of Contents:

`git reset --hard HEAD` and `git status -s`

10. Revert a commit, softly
    `git revert -n`
11. See diff-erence for the entire project (not just one file at a time) in a 3rd party diff tool
    `git difftool -d`
12. Ignore the white space
    `git diff -w`
13. Only "add" some changes from a file
    `git add -p`
14. Discover and zap those old branches
    `git branch -a`
15. Stash only some files
    `git stash -p`
16. Good commit messages
17. Git Auto-completion
18. Create aliases for your most frequently used commands
19. Quickly find a commit that broke your feature (EXTRA AWESOME)
    `git bisect`

1. Parameters for better logging

Sample Command `git log --oneline --graph`

Chances are, by now you've used git log. It supports a number of command line parameters, which are very powerful, especially when used in combination. Here are the ones that I use the most:

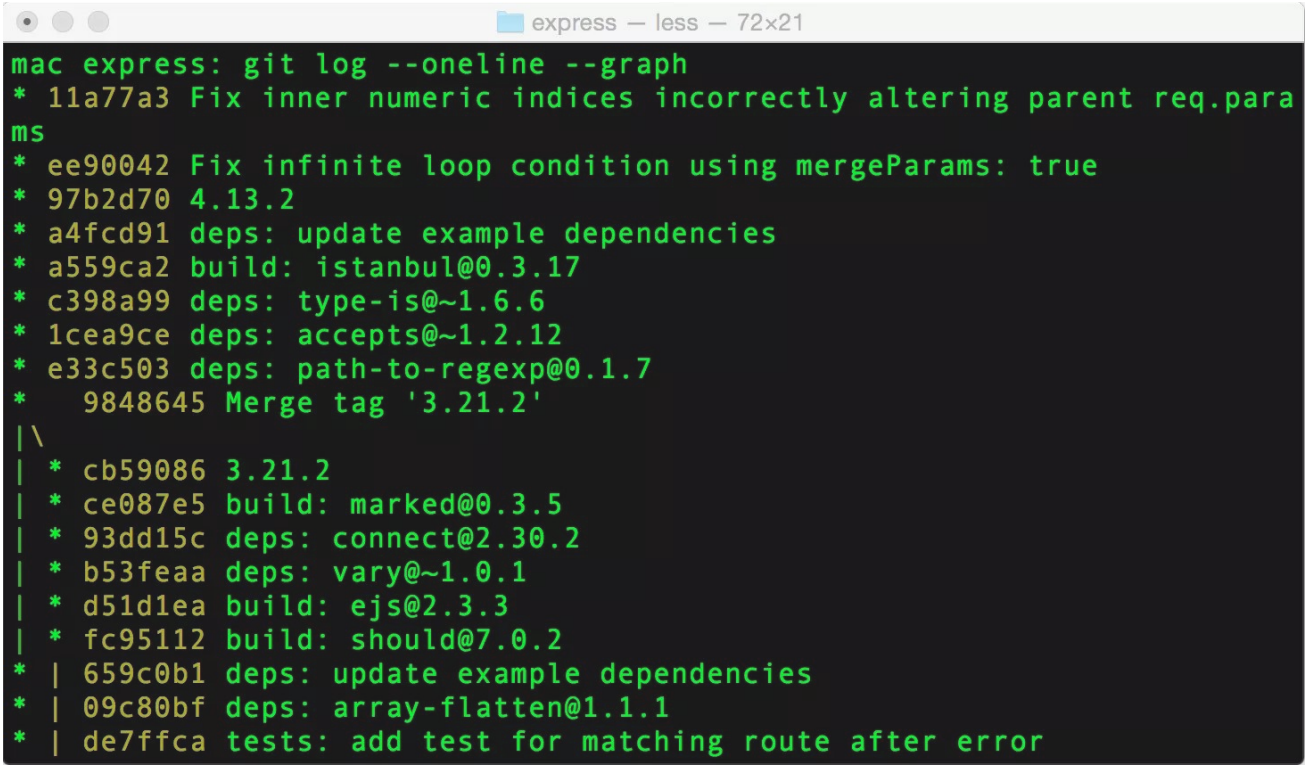
- `--author="Alex Kras"` – Only show commits made by a certain author
- `--name-only` – Only show names of files that changed
- `--oneline` – Show commit data compressed to one line
- `--graph` – Show dependency tree for all commits
- `--reverse` – Show commits in reverse order (Oldest commit first)
- `--after` – Show all commits that happened after certain date
- `--before` – Show all commits that happened before certain data

For example, I once had a manager who required weekly reports submitted each

Friday. So every Friday I would just run: `git log --author="Alex Kras" --after="1 week ago" --oneline`, edit it a little and send it in to the manager for review.

Git has a lot more command line parameters that are handy. Just run `man git-log` and see what it can do for you.

If everything else fails, git has a `--pretty` parameter that let's you create a highly customizable output.

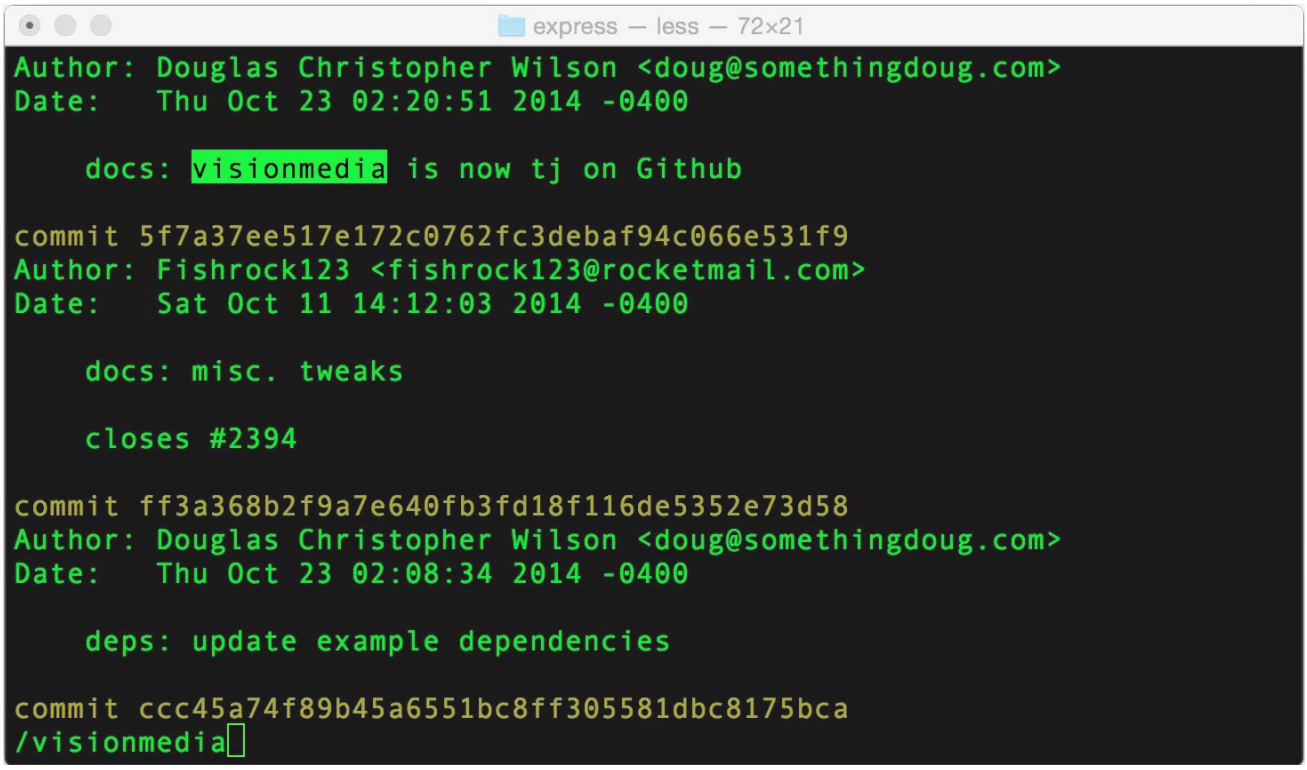


2. Log actual changes in a file

Sample Command `git log -p filename`

`git log -p` or `git log -p filename` lets you view not only the commit message, author, and date, but actual changes that took place in each commit.

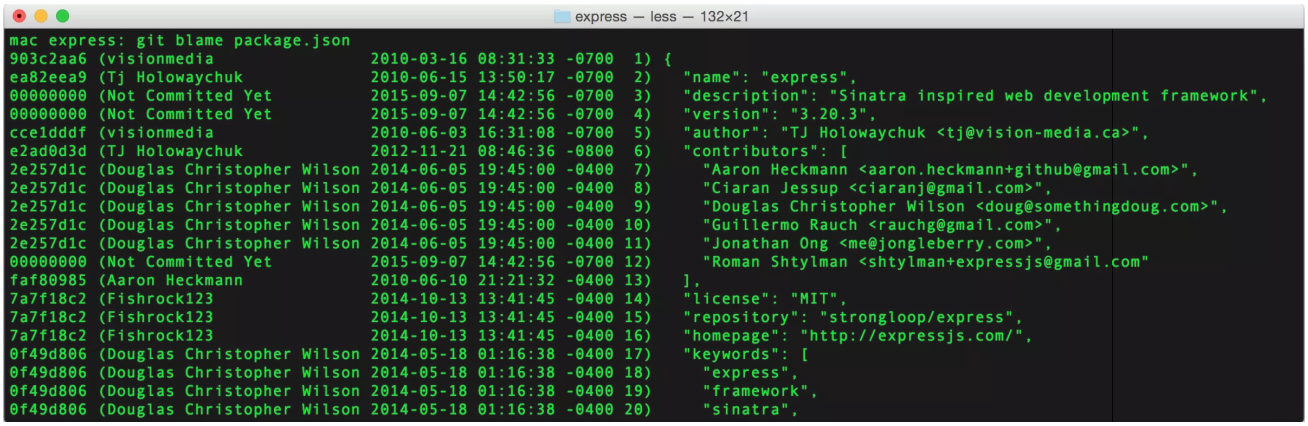
Then you can use the regular `less` search command of "slash" followed by your search term`/{{your-search-here}}` to look for changes to a particular keyword over time. (Use lower case n to go to the next result, and upper case N to go to the previous result).



3. Only Log changes for some specific lines in a file

Sample Command `git log -L 1,1:some-file.txt`

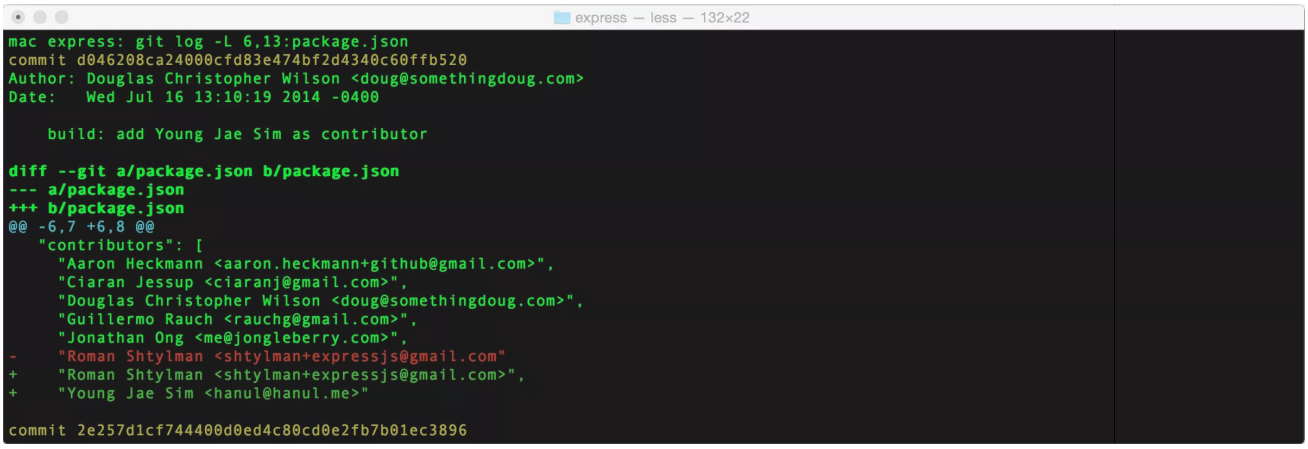
You can use `git blame filename` to find the person responsible for every line of the file.



`git blame` is a great tool, but sometimes it does not provide enough information.

An alternative is provided by `git log` with a `-L` flag. This flag allows you to specify particular lines in a file that you are interested in. Then Git would only log changes relevant to those lines. It's kind of like `git log -p` with focus.

`git log -L 1,1:some-file.txt`



4. Log changes not yet merged to the parent branch

Sample: `git log --no-merges master..`

If you ever worked on a long-lived branches, with multiple people working on it, chances are you've experienced numerous merges of the parent branch(i.e. master) into your feature branch. This makes it hard to see the changes that took place on the master branch vs. the changes that have been committed on the feature branch and which have yet to be merged.

`git log --no-merges master..` will solve the issue. Note the `--no-merges` flag indicate to only show changes that have not been merged yet to ANY branch, and the **master..** option, indicates to only show changes that have not been merged to master branch. (You must include the **..** after master).

You can also do `git show --no-merges master..` or `git log -p --no-merges master..` (output is identical) to see actual file changes that are have yet to be merged.

5. Extract a file from another branch

Sample: `git show some-branch:some-file.js`

Sometimes it is nice to take a pick at an entire file on a different branch, without switching to this branch.

You can do so via `git show some-branch-name:some-file-name.js`, which

would show the file in your terminal.

You can also redirect the output to a temporary file, so you can perhaps open it up in a side by side view in your editor of choice.

```
git show some-branch-name:some-file-name.js > deleteme.js
```

Note: If all you want to see is a diff between two files, you can simple run: `git diff some-branch some-filename.js`

6. Some notes on rebasing

Sample: `git pull --rebase`

We've talked about a lot of merge commits when working on a remote branch. Some of those commits can be avoided by using `git rebase`.

Generally I consider rebasing to be an advanced feature, and it's probably best left for another post.

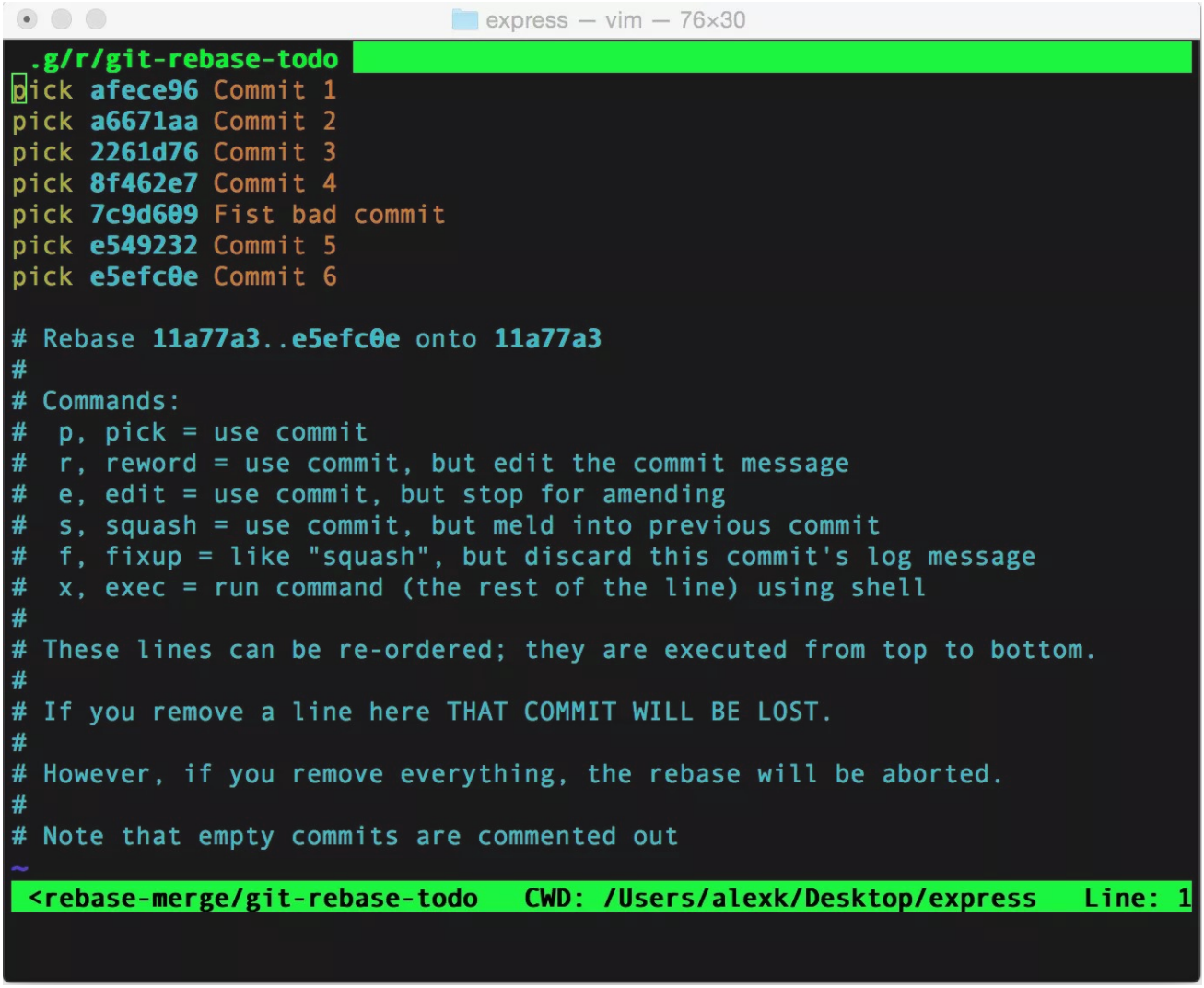
Even git book has the following to say about rebasing.

> Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:
>
> Do not rebase commits that exist outside your repository
>
> If you follow that guideline, you'll be fine. > If you don't, people will hate you, and you'll be scorned by friends and family.

https://git-scm.com/book/en/v2/Git-Branching-Rebasing#The-Perils-of-Rebasing

That being said, rebasing is not something to be afraid of either, rather something that you should do with care.

Probably the best way to rebase is using **interactive rebasing**, invoked via `git rebase -i {{some commit hash}}`. It will open up an editor, with self explanatory instruction. Since rebasing is outside of the scope of this article, I'll leave it at that.



One particular rebase that is very helpful is `git pull --rebase`.

For example, imagine you are working on a local version of a master branch, and you made one small commit. At the same time, somebody else checked in a week worth of work onto the master branch. When you try to push your change, git tells you to do a `git pull` first, to resolve the conflict. Being a good citizen that you are, you do a `git pull` to end up with the following commit message auto generated by git.

> “ Merge remote-tracking branch ‘origin/master’

While this is not a big deal and is completely safe, it does clutter log history a bit.

In this case, a valid alternative is to do a `git pull --rebase` instead.

This will force git to first pull the changes, and then re-apply(rebase) your un-pushed commits on top of the latest version of the remote branch, as if they just

took place. This will remove the need for merge and the ugly merge message.

7. Remember the branch structure after a local merge

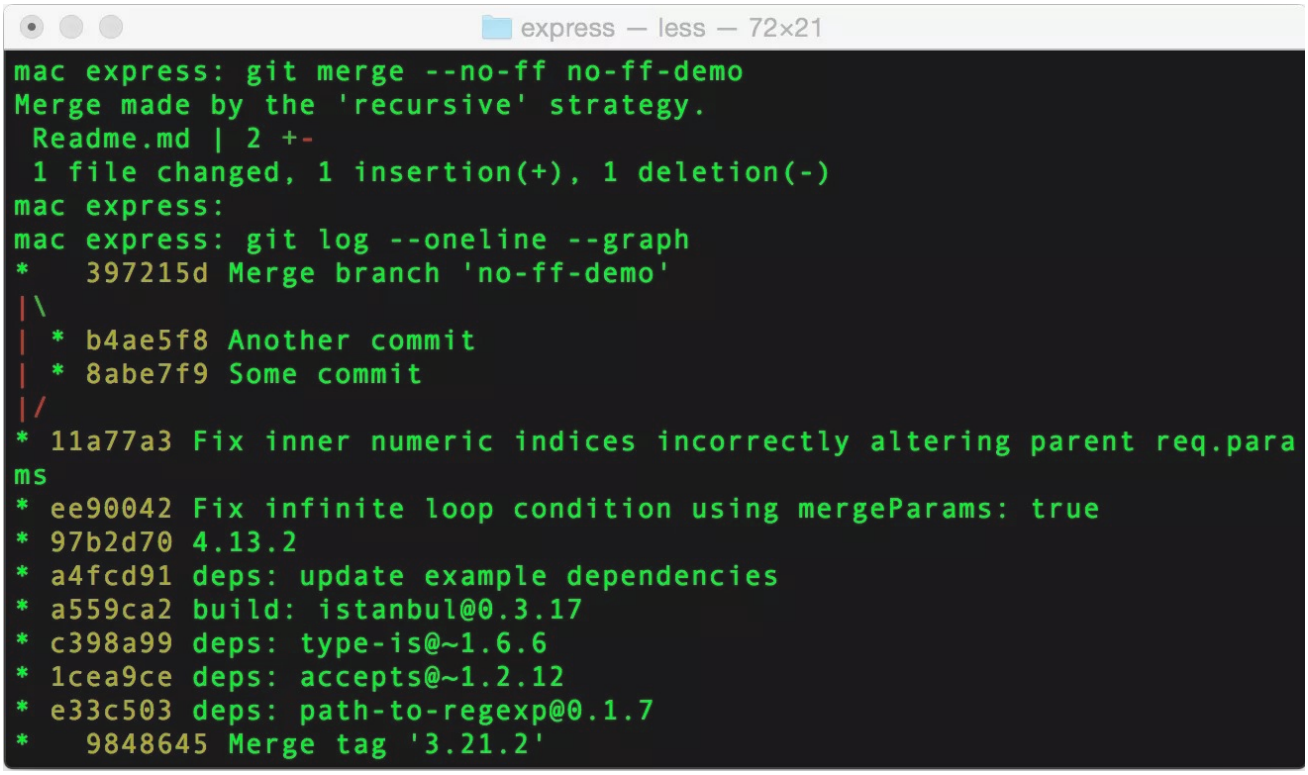
Sample: `git merge --no-ff`

I like to create a new branch for every new bug or feature. Among other benefits, it helps me to get a great clarity on how a series of commits may relate to a particular task. If you ever merged a pull request on github or a similar tool, you will in fact be able to nicely see the merged branch history in `git log --oneline --graph` view.

If you ever try to merge a local branch, into another local branch, you may notice git has flatten out the branch, making it show up as a straight line in git history.

If you want to force git to keep branches history, similarly to what you would see during a pull request, you can add a `--no-ff` flag, resulting in a nice commit history tree.

```
git merge --no-ff some-branch-name
```



8. Fix your previous commit, instead of making a new commit

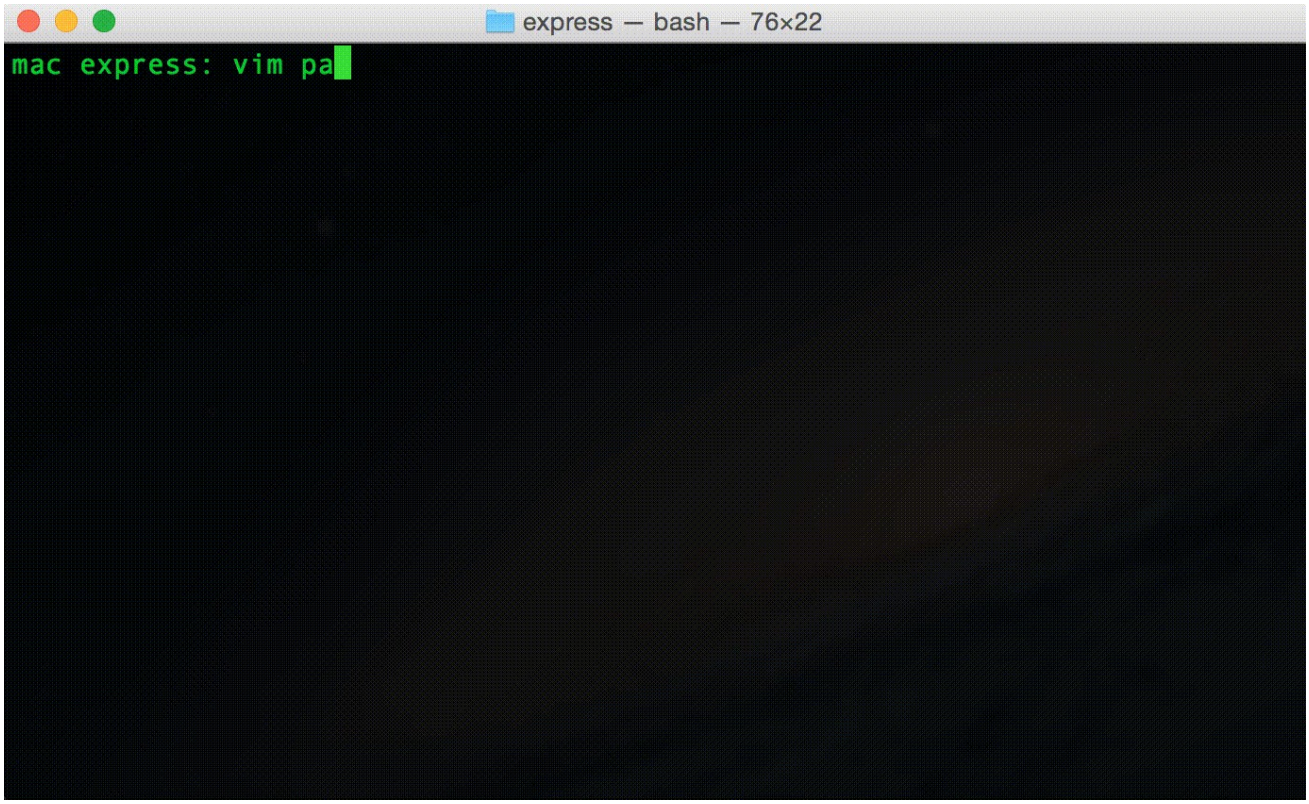
Sample `git commit --amend`

This one is pretty straightforward.

Let say you made a commit and then realized you made a typo. You could make a new commit with a "descriptive" message **typo**. But there is a better way.

If you haven't pushed to the remote branch yet, you can simply do the following:

1. Fix your typo
2. Stage the newly fixed file via `git add some-fixed-file.js`
3. Run `git commit --amend` which would add the most recent changes to your latest commit. It will also give you a chance to edit the commit message.
4. Push the clean branch to remote, when ready



If you are working on your own branch, you can fix commits even after you have pushed, you would just have to do a `git push -f` (-f stands for force), which would over-ride the history. But you **WOULD NOT want to do this** on a branch that is **being used by other people** (as discussed in rebase section above). At that point, a "typo" commit, might be your best bet.

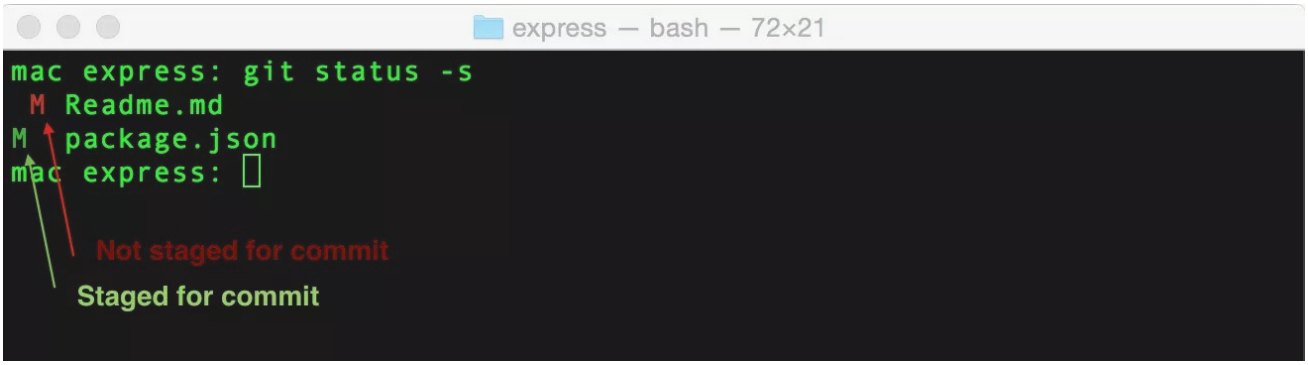
9. Three stages in git, and how to move between them

Sample `git reset --hard HEAD` and `git status -s`

As you may already know by now, a file in git can be in 3 stages:
1. Not staged for commit
2. Staged for commit
3. Committed

You can see a long description of the files and state they are in by running `git status`. You move a file from *"not staged for commit"* stage to *"staged for commit"* stage, by running `git add filename.js` or `git add .` to add all files at once.

Another view that makes it much easier to visualize the stages is invoked via `git status -s` where `-s` stand for short (I think), and would result in an output that looks like that:



Obviously, `git status` will not show files that have already been committed, you can use `git log` to see those instead 🙂

There are a couple of options available to you to move the files to a different stage.

Resetting the files

There are 3 types of reset available in git. A reset allows you to return to a particular version in git history.

1. `git reset --hard {{some-commit-hash}}` – Return to a particular point in history. **All changes made after this commit are discarded**.
2. `git reset {{some-commit-hash}}` – Return to a particular point in history. **All changes made after this commit are moved "not yet staged for commit" stage**. Meaning you would have to run `git add .` and `git commit` to add

them back in.
3. `git reset --soft {{some-commit-hash}}` – Return to a particular point in history. **All changes made after this commit are moved to "staged for commit" stage**. Meaning you only need to run `git commit` to add them back in.

This may appear as useless information at first, but it is actually very handy when you are trying to move through different version of the file.

Common use cases that I find myself using the reset are bellow:

1. I want to forget all the changes I've made, clean start – `git reset --hard HEAD` (Most common)
2. I want to edit, re-stage and re-commit files in some different order – `git reset {{some-start-point-hash}}`
3. I just want to re commit past 3 commits, as one big commit – `git reset --soft {{some-start-point-hash}}`

Check out some files

If you simply want to forget some local changes for some files, but at the same time want to keep changes made in other files, it is much easier to check out committed versions of the files that you want to forget, via:
`git checkout forget-my-changes.js`

It's like running `git reset --hard` but only on some of the files.

As mentioned before you can also check out a different version of a file from another branch or commit.

`git checkout some-branch-name file-name.js` and
`git checkout {{some-commit-hash}} file-name.js`

You'll notice that the checked out files will be in a "staged for commit" stage. To move them back to "un-staged for commit" stage, you would have to do a `git reset HEAD file-name.js`. You can run `git checkout file-name.js` again, to return the file to it's original state.

Note, that running `git reset --hard HEAD file-name.js` does not work. In general, moving through various stages in git is a bit confusing and the pattern is not always clear, which I hoped is to remedied a bit with this section.

10. Revert a commit, softly

Sample `git revert -n`

This one is handy if you want to undo a previous commit or two, look at the changes, and see which ones might have caused a problem.

Regular `git revert` will automatically re-commit reverted files, prompting you to write a new commit message. The `-n` flag tells git to take it easy on committing for now, since all we want to do is look.

11. See diff-erence for the entire project (not just one file at a time) in a 3rd party diff tool

Sample `git difftool -d`

My favorite diff-ing program is Meld. I fell in love with it during my Linux times, and I carry it with me.

I am not trying to sell you on Meld, though. Chances are you have a diff-ing tool of choice already, and git can work with it too, both as a merge and as a diff tool. Simply run the following commands, making sure to replace meld with your favorite diff tools of choice:

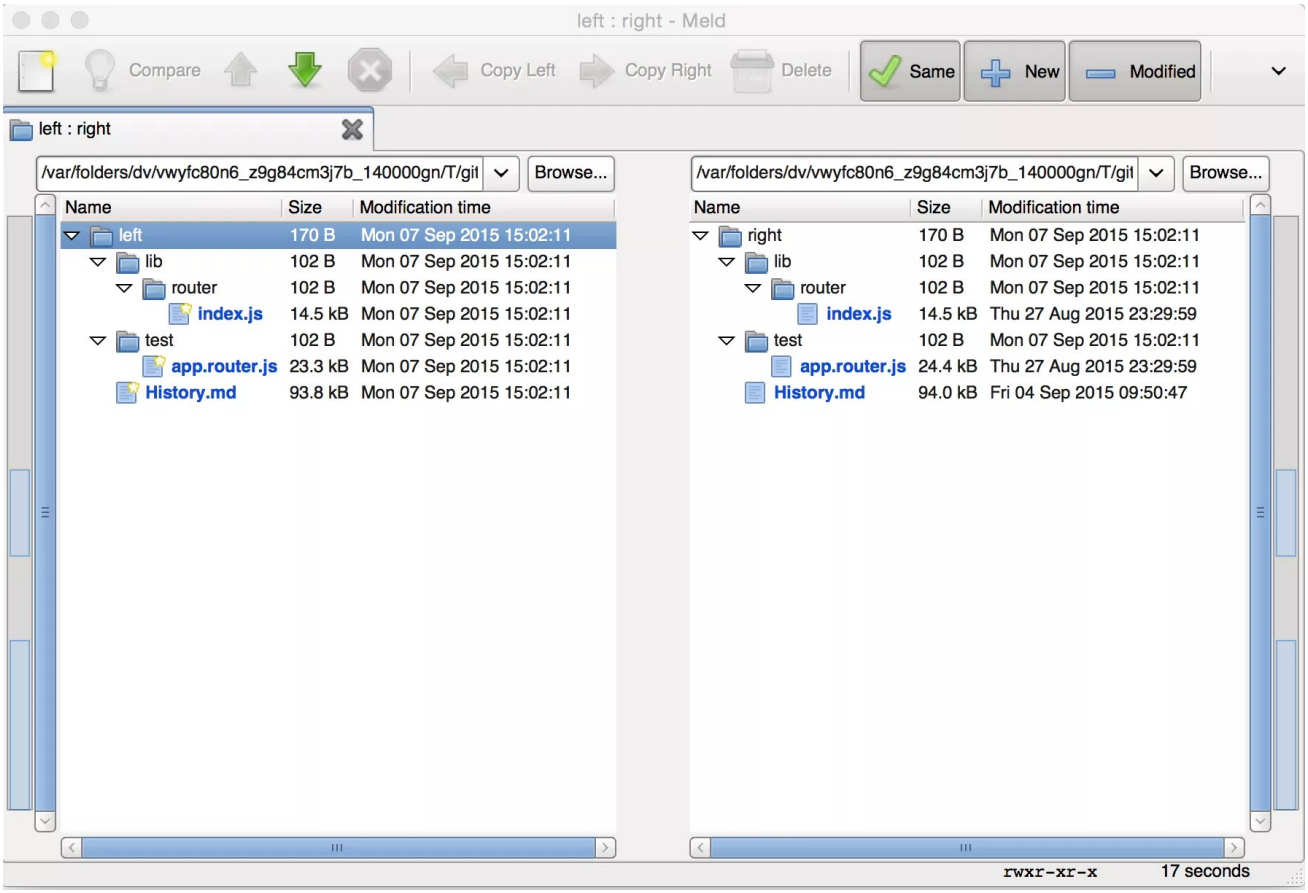
```
git config --global diff.tool meld
git config --global merge.tool meld
```

After that all you have to do is run `git difftool some-file.js` to see the changes in that program instead of the console.

But some of the diff-ing tools (such as meld) support full directory diffs.

If you invoke `git difftool` with a `-d` flag, it will try to diff the entire folder. Which could be really handy at times.

```
git difftool -d
```



12. Ignore the white space

Sample `git diff -w` or `git blame -w`

Have you ever re-indented or re-formatted a file, only to realize that now `git blame` shows that you are responsible for everything in that file?

Turns out, git is smart enough to know the difference. You can invoke a lot of the commands (i.e. `git diff`, `git blame`) with a `-w` flag, and git will ignore the white space changes.

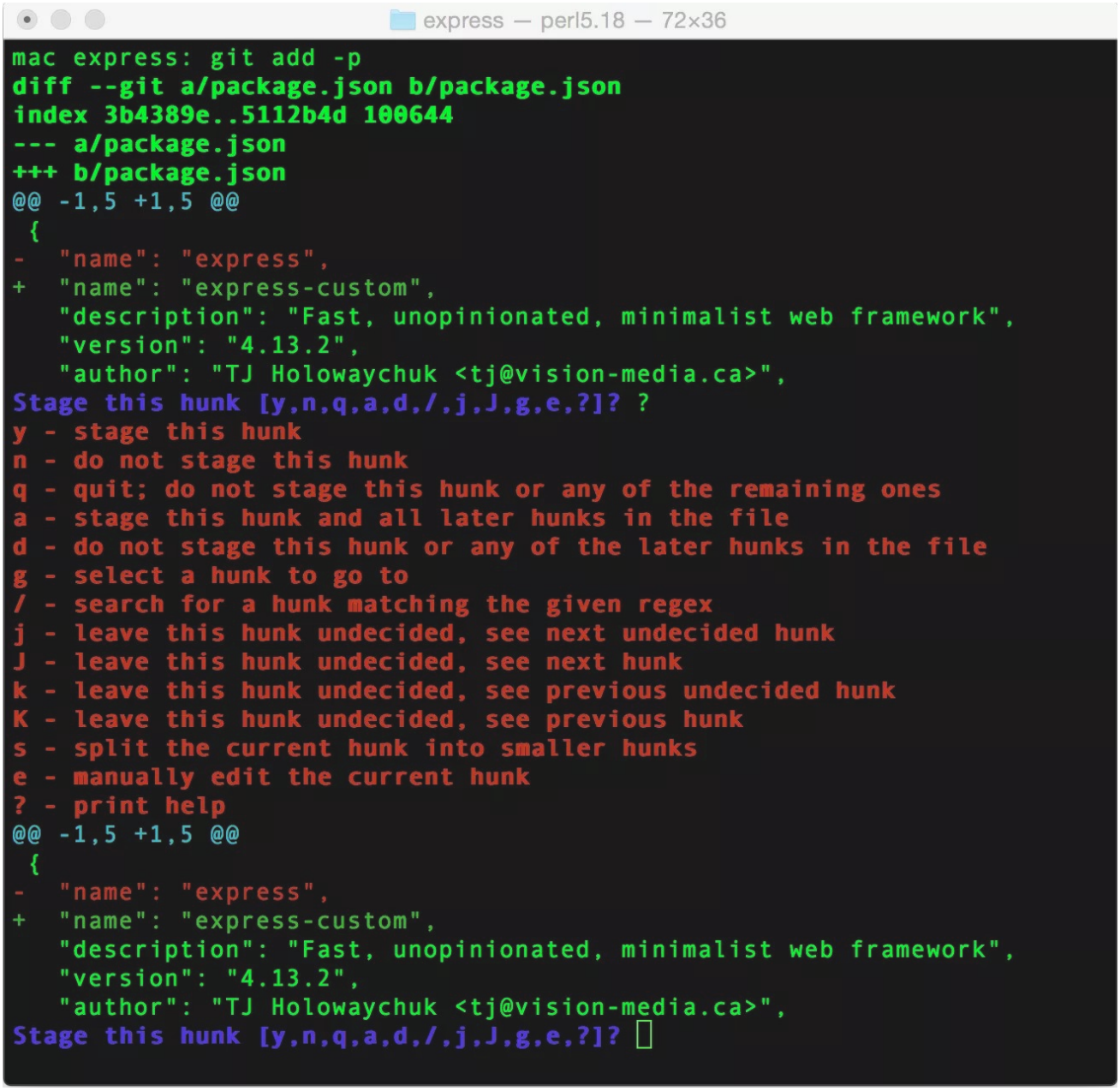
13. Only “add” some changes from a file

Sample `git add -p`

Somebody at git must really like the `-p` flag, because it always comes with some handy functionality.

In case of `git add`, it allows you to interactive select exactly what you want to be

committed. That way you can logically organize your commits in an easy to read manner.



14. Discover and zap those old branches

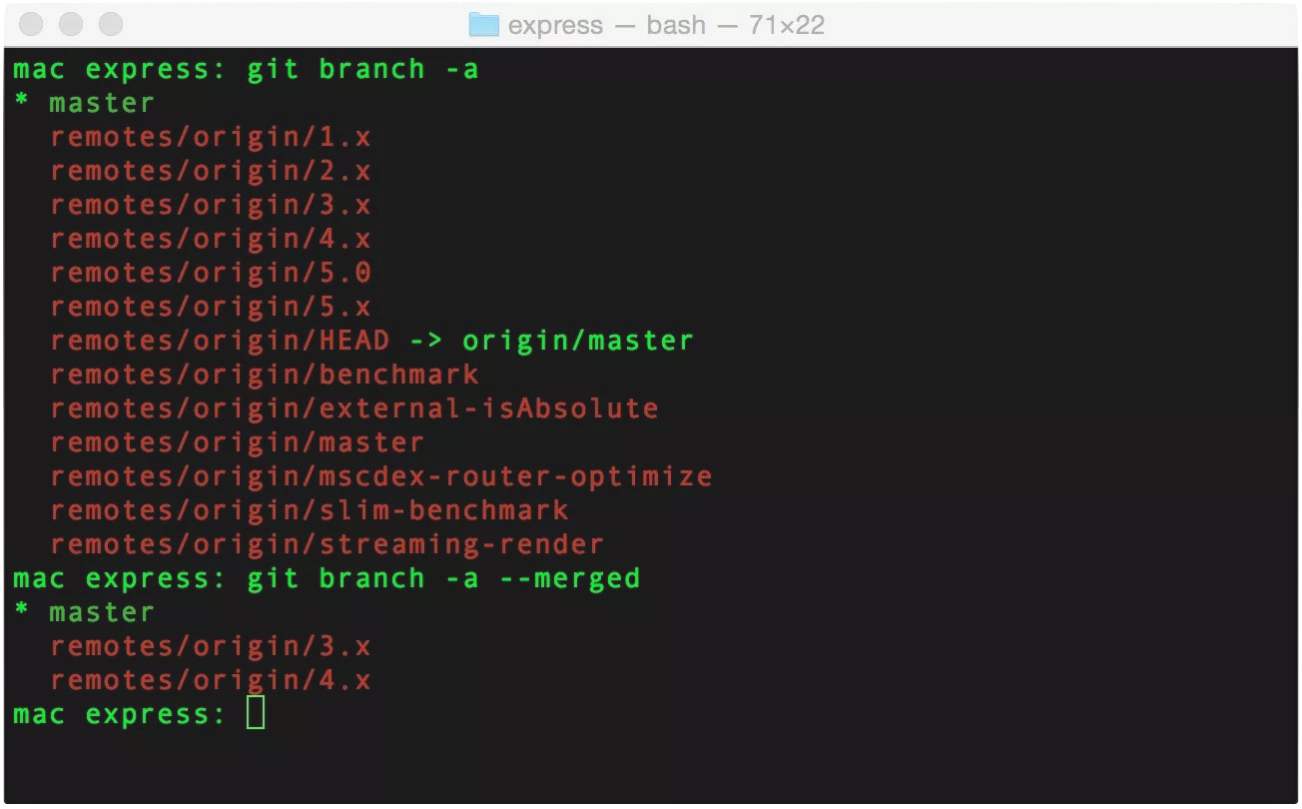
Sample `git branch -a`

It is common for a large number of remote branches to just hang around, some even after they have been merged into the master branch. If you are a neat freak (at least when it comes to code) like me, chances are they will irritate you a little.

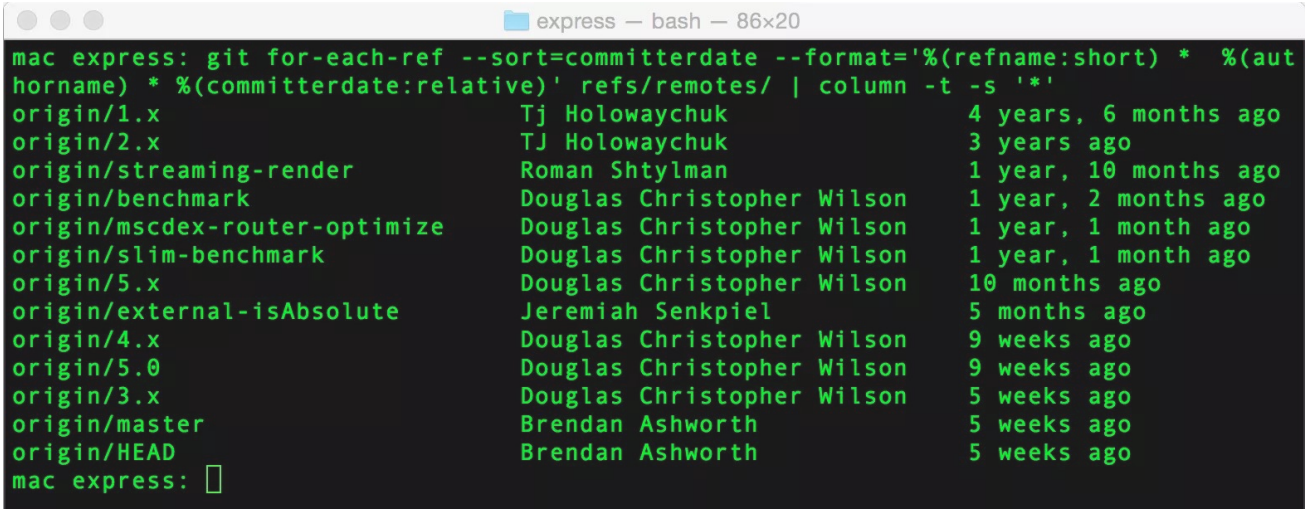
You can see all of the remote branches by running git branch with the `-a` flag (show all branches) and the `--merged` flag would only show branches that are fully merged

into the master branch.

You might want to run `git fetch -p` (fetch and purge old data) first, to make sure your data is up to date.



If you want to get really fancy, you can get a list of all the remote branches, and the list of last commits made on those branches by running: `git for-each-ref --sort=committerdate --format='%(refname:short) * %(authorname) * %(committerdate:relative)' refs/remotes/ | column -t -s '*'`.



Unfortunately, there is no easy way (that I know of) to only show merged branches. So you might have to just compare the two outputs or write a script to do it for you.

15. Stash only some files

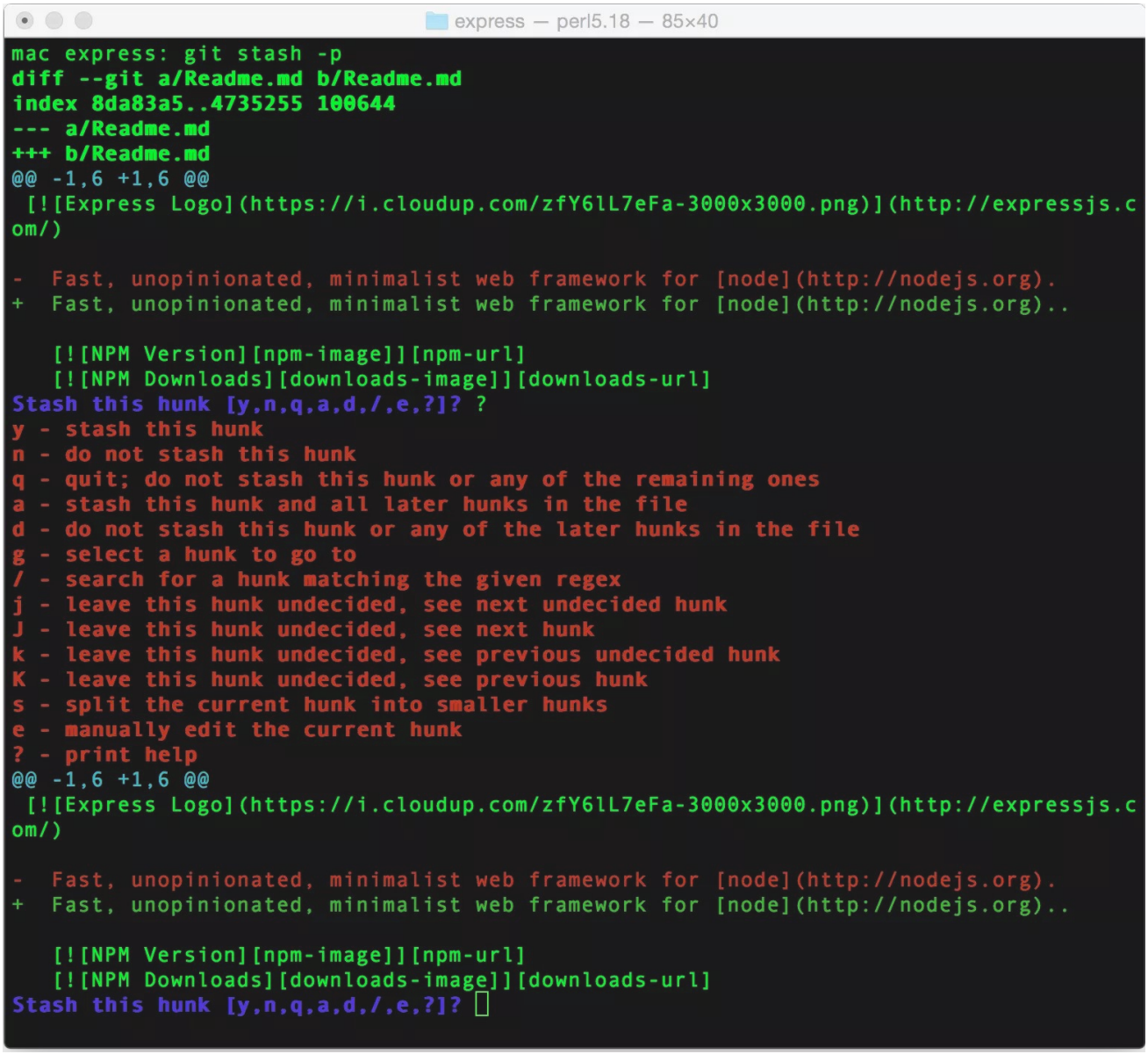
Sample `git stash —keep-index` or `git stash -p`

If you don't yet know what `git stash` does, it simply puts all your unsaved changes on a "git stack" of sorts. Then at a later time you can do `git stash pop` and your changes will be re-applied. You can also do `git stash list` to see all your stashed changes. Take a look at `man git-stash` for more options.

One limitation of regular `git stash` is that it will stash all of the files at once. And sometimes it is handy to only stash some of the file, and keep the rest in your working tree.

Remember the magic `-p`command? Well it's really handy with `git stash` as well. As you may have probably guessed by now, it will ask you to see which chunks of changes you want to be stashed.

Make sure to hit `?` while you at it to see all available options.



Another handy trick, for stashing only some of the files, is to:

1. `add` the files that you DO NOT want to get stashed (i.e. `git add file1.js, file2.js`)
2. Call `git stash --keep-index`. It will only stash files that have not been added.
3. Call `git reset` to un-stage the added files and continue your work.

16. Good commit messages

A little while ago I came across a great article on how to write a good commit message. Check it out here: How to Write a Git Commit Message

One rule that really stood out for me is, **"every good commit should be able to**

complete the following sentence”

When applied, this commit will: {{ YOUR COMMIT MESSAGE}}

For example:
– *When applied this commit will* **Update README file**
– *When applied this commit will* **Add validation for GET /user/:id API call**
– *When applied this commit will* **Revert commit 12345**

17. Git Auto-completion

Git packages for some operating systems (i.e. Ubuntu) come with git auto completion enabled by default. If your operating system did not come with one(Mac doesn’t), you can easily enable it by following these guidelines:

https://git-scm.com/book/en/v1/Git-Basics-Tips-and-Tricks#Auto-Completion

18. Create aliases for your most frequently used commands

TLDR; Use git or bash aliases for most commonly used long git commands

Best way to use Git is via command line, and the best way to learn the command line is by doing everything the hard way first (typing everything out).

After a while, however, it might be a good idea to track down your most used commands, and create an easier aliases for them.

Git comes with built in aliases, for example you can run the following command once:

```
git config --global alias.l "log --oneline --graph"
```

Which would create a new git alias named `l`, that would allow you to run: `git l` instead of `git log --oneline --graph`.

Note that you can also append other parameters after the alias (i.e. `git l --author="Alex"`).

Another alternative, is good old Bash alias.

For example, I have the following entry in my .bashrc file.

`alias gil="git log --oneline --graph"`, allowing me to use `gil` instead of the long command,which is even 2 character shorter than having to type `git l` :).

19. Quickly find a commit that broke your feature (EXTRA AWESOME)

Sample: `git bisect`

`git bisect` uses divide and conquer algorithm to find a broken commit among a large number of commits.

Imagine yourself coming back to work after a week long vacation. You pull the latest version of the project only to find out that a feature that you worked on right before you left is now broken.

You check the last commit that you've made before you left, and the feature appear to work there. However, there has been over a hundred of other commits made after you left for your trip, and you have no idea which of those commits broke your feature.

At this point you would probably try to find the bug that broke your feature and use `git blame` on the breaking change to find the person to go yell at.

If the bug is hard to find, however, you could try to navigate your way through the commit history, in attempt to pin point where the things went bad.

The second approach is exactly where `git bisect` is so handy. It will allow you to find the breaking change in the fastest time possible.

So what does `git bisect` do?

After you specify any known bad commit and any known good commit, `git bisect` will split the in-between commits in half, and checkout a new (nameless) branch in the middle commit to let you check if your future is broken at that point in time.

Let say the middle commit still works. You would then let git know that via `git bisect good` command. Then you only have half of the commits left to test.

Git would then split the remaining commits in half and into a new branch(again), letting you to test the feature again.

`git bisect` will continue to narrow down your commits in a similar manner, until the first bad commit is found.

Since you divide the number of commits by half on every iteration, you are able to find your bad commits in log(n) time (which is simply a "big O" speak for very fast).

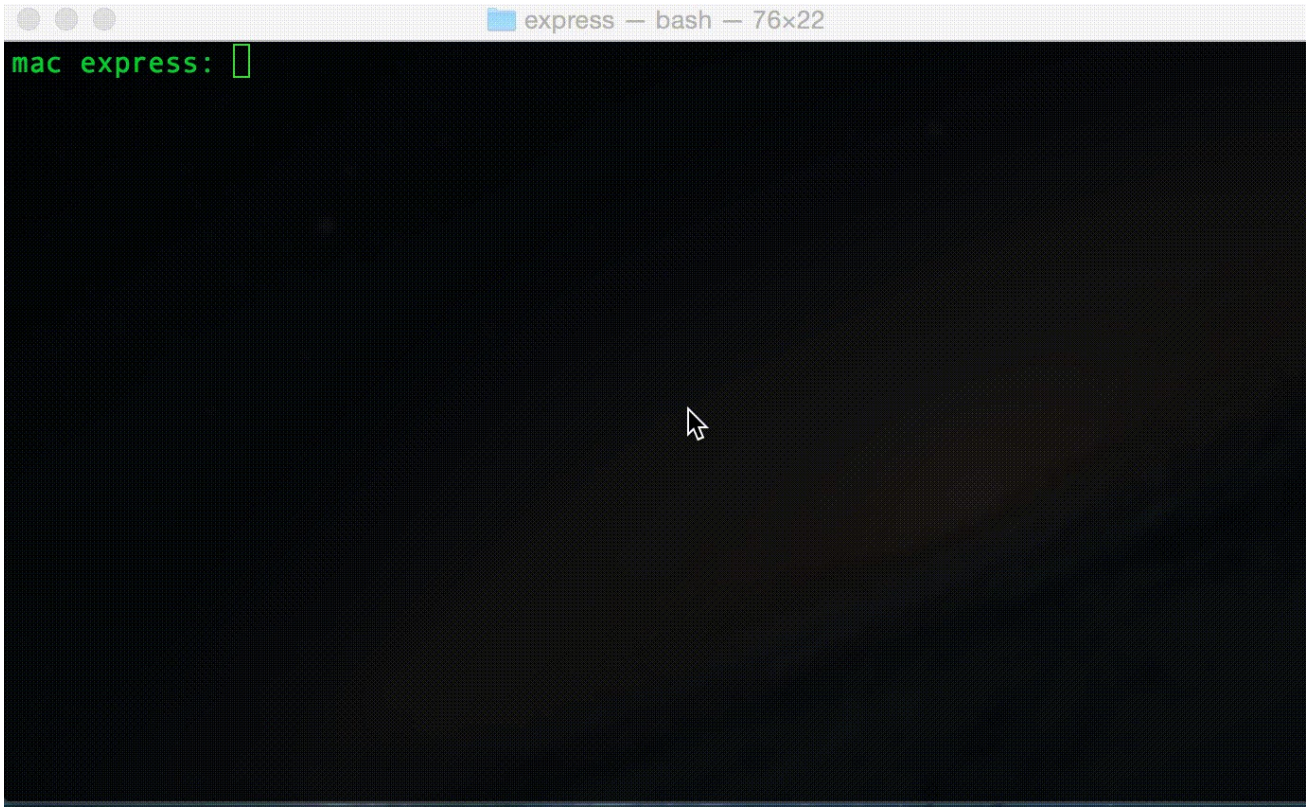
The actual commands you need to run to execute the full `git bisect` flow are:

1. `git bisect start` – let git know to start bisecting.
2. `git bisect good {{some-commit-hash}}` – let git know about a known good commit (i.e. last commit that you made before the vacation).
3. `git bisect bad {{some-commit-hash}}` – let git know about a known bad commit (i.e. the HEAD of the master branch). `git bisect bad HEAD` (HEAD just means the last commit).
4. At this point git would check out a middle commit, and let you know to run your tests.
5. `git bisect bad` – let git know that the feature does not work in currently checked out commit.
6. `git bisect good` – let git know that the feature does work in currently checked out commit.
7. When the first bad commit is found, git would let you know. At this point `git b`

`isect` is done.

8. `git bisect reset` – returns you to the initial starting point of `git bisect` process, (i.e. the HEAD of the master branch).
9. `git bisect log` – log the last `git bisect` that completed successfully.

You can also automate the process by providing `git bisect` with a script. You can read more here: http://git-scm.com/docs/git-bisect#_bisect_run



Final Note

If you liked this post, please consider reviewing it on Amazon.

Filed Under: Git
Tagged With: git

New posts only. No other messages will be sent.

Email Address

SUBSCRIBE

AFFILIATES

AirBNB – Get $40 off your first stay: https://abnb.me/e/ogMDXkHUsU

Hired.com: https://hired.com/x/YjFDXP

Programming Interviews Exposed or anything else from Amazon: https://amzn.to/2VDTSzC

Linode, where this site is hosted: https://www.linode.com/?r=0c6b1d50096034351b7feb80d175d7e315cc9a96

BlueHost, if you don't want to manage your own server: https://www.bluehost.com/track/akras14/

Coinbase, if you are into Cryptocurrency https://www.coinbase.com/join/59c3348a22766100f93d9662

Comments

Leave a Reply

Jhay Jong says
January 5, 2018 at 6:05 pm

Valuable writing – Just to add my thoughts , if your company are searching for a service to merge two images , my boss encountered a service here https://goo.gl/EPmyVk.

Reply

Harry Moreno (@morenoh149) says
November 7, 2016 at 11:26 am

tip #4 doesn't work for me

```
git log --no-merges master..
```

returns

```
fatal: ambiguous argument 'master..': unknown revision or p
ath not in the working tree.
Use '--' to separate paths from revisions, like this:
'git <command> [<revision>...] -- [<file>...]'
```

should it be updated?

Reply

Alex Kras says
November 20, 2016 at 6:06 am

What git version are you on? I am on `git --version` 2.9.2

Reply

peitor says
October 18, 2016 at 12:35 pm

I would add “Use ‘Git Credential Manager’ for Windows Users”.

Provides secure Git credential storage for Windows
https://github.com/Microsoft/Git-Credential-Manager-for-Windows

Reply

Harry Moreno (@morenoh149) says
August 18, 2016 at 6:37 am

`git commit -v` will put a diff in your editor. Super useful for writing meaningful commit messages.

Reply

Alex Kras says
August 18, 2016 at 3:24 pm

Nice one!

Reply

Wojtek Ryrych (@wryrych) says
May 7, 2016 at 7:52 pm

Hi! I think that `git log --no-merges master..` is similar to (or the same)

`g log head --not origin/master`. This is at least from what I see when pass the output to `| wc -l`

Reply

JP says
May 6, 2016 at 6:43 pm

“Table of Contexts” should be “Table of Contents”, shouldn’t it?

Reply

Alex Kras says
May 15, 2016 at 10:55 pm

Yes, thank you. I’ve fixed it.

Reply

Gaurav Ramesh (@ggauravr) says
May 2, 2016 at 1:58 am

A small suggestion: `man git log` just invokes `man` for `git`. `man git-log` might be more specific.

Reply

Alex Kras says
May 2, 2016 at 2:03 am

Thank you, it was a typo. I’ve fixed it.

Reply

Gaurav Ramesh (@ggauravr) says
May 2, 2016 at 2:15 am

Great. Same with `man git stash` vs `man git-stash`! Awesome writeup, otherwise. Thanks!

Reply

Alex Kras says
May 2, 2016 at 2:57 am

Thanks, fixed both.

Reply

Vilson Vieira says
April 22, 2016 at 11:16 pm

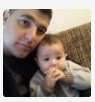
A small fix:

alias gil=”git log –online –graph”

to:

alias gil=”git log –oneline –graph”

Reply



Alex Kras says
April 27, 2016 at 1:45 am

Thanks, fixed it.

Reply

Dhananjay says
April 22, 2016 at 1:38 pm

Nice

Reply

Harry Moreno says
April 21, 2016 at 3:09 pm

If anyone could show how to get filemerge working as the diff tool for OSX users. I'd greatly appreciate it.

Reply

impshum says
April 20, 2016 at 10:52 pm

SourceTree

Reply

Deekshith Allamaneni says
April 20, 2016 at 2:44 pm

Great article. I never knew we can log changes in single file. That too for specific lines. Thats very helpful. Thanks.

I made a commit message template if you would like to check it out: https://gist.github.com/adeekshith/cd4c95a064977cdc6c50

Reply

John says
April 20, 2016 at 10:15 am

Awesome post, thank you.

Reply

Rahil Wazir says
April 20, 2016 at 9:00 am

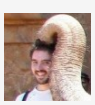
Another one would be `git checkout -p` to checkout specific lines

Reply

college football picks says
October 13, 2015 at 12:10 am

Quality posts is the key to interest the users to pay a quick visit the web site, that's what this website is providing.

Reply



Matthieu Moy says
October 11, 2015 at 8:11 pm

> Unfortunately, there is no easy way (that I know of) to only show merged branches

What about `git branch --merged` ?

(tag and for-each-ref will soon have the same –merged option)

Reply

Mircea Vasiliniuc says
September 10, 2015 at 3:59 pm

Don't forget about git reflog http://git-scm.com/docs/git-reflog. The reflog covers all recent actions and basically you can undo any of your actions. For example if did a reset –hard accidentally you can recover with reflog.

Reply

Java67 says
September 9, 2015 at 3:37 pm

Great tips, though I am new to Git and doesn't understand all of them, I can see the value. bookmarked for future reference.

Reply

Jason Noble says
September 8, 2015 at 5:37 pm

In section 14, you state: "Unfortunately, there is no easy way (that I know of) to only show merged branches. So you might have to just compare the two outputs or write a script to do it for you."

The command you're looking for is `git branch –merged`.

This command will show you all the local branches that have been merged into your current branch.

Reply

Mike says
September 8, 2015 at 5:13 pm

This is a really thorough collection of git tips. More importantly you've laid out the information in an easily understandable manner. Git can sometimes be really confusing even to those that have been using it for years. Thanks for putting this list together and keep up the good work.

Reply

Tony Estrada says
September 8, 2015 at 4:07 pm

Really enjoyed this article. Kudos to the author!

Reply

Adam Jones says
September 8, 2015 at 3:05 pm

This doesn't apply to everyone but for those using Windows, you need to check out and install Posh-Git (http://dahlbyk.github.io/posh-git/). It adds a ton of useful stuff to the PowerShell prompt like the name of the current branch name, current status of your index, tab expansions for Git commands and branches, etc.

Reply

Alex says
September 8, 2015 at 3:36 pm

Good point, another one that I've used in non-Windows environment was: https://github.com/rtomayko/git-sh

Reply

Jason Edwards says
September 8, 2015 at 2:20 pm

I really enjoyed reading this post and came away from it having learnt a couple of things. Thankyou for taking the time to write this.

Reply

Alex says
September 8, 2015 at 2:51 pm

Thank you, I appreciate it!

Reply

Arnaud Rinquin says
September 8, 2015 at 9:27 am

Isn't there a problem with "When applied this commit will Updated README file"? I bet it should be "Update" instead of "Updated"

Reply

Alex says
September 8, 2015 at 2:17 pm

Yes, that was a typo. Thanks, I fixed it.

Reply

ashok says
September 8, 2015 at 6:04 am

It seems there is some issue with the link “How to Write a Git Commit Message”.

Reply

Alex says
September 8, 2015 at 6:07 am

Thank you, fixed it. Here is the link so you don’t have to look for it.
http://chris.beams.io/posts/git-commit/

Reply

Trackbacks

19 Git Tips For Everyday Use – Linktail says:
July 11, 2018 at 12:00 am
[…] Click here […]

Reply

Front End Developer – Free Tools ! | Marco Gallo Dev says:
December 31, 2017 at 9:01 am
[…] I wrote a few posts on Git. For Git beginners I recommend taking a look at Git Cheat Sheet. More experienced users may enjoy 19 Git Tips For Everyday Use. […]

Reply

Git How To Revert To Previous Commit | Information says:
October 18, 2016 at 2:41 pm
[…] 19 Tips For Everyday Git Use – alexkras.com – I've been using git full time for the past 4 years, and I wanted to share the most practical tips that I've learned along the way. Hopefully, it will be useful to … […]

Reply

Generate Weekly Reports from Your Git Commits says:
August 13, 2016 at 1:07 am
[…] explained underling git command in detail in 19 Git Tips for Everyday Use. In summary it uses python to walk all of the directories. For every directory the script runs git […]

Reply

Collections | Git FAQ says:
July 17, 2016 at 6:15 pm
[…] 19 Tips For Everyday Git Use […]

Reply

How to Add Git Auto-Completion for the Command Line Tools says:

July 15, 2016 at 2:55 pm

[...] I initially learned this tip from Git Book, which I highly recommend to any new or intermediate Git users. You may also like my other posts on git – Git Cheatsheet and 19 Tips for Everyday Git Use. [...]

Reply

Revue de presse de juin | Blog Arolla says:

June 7, 2016 at 9:11 am

[...] Enfin pour terminer voici 19 tips pour vous aider avec Git : https://www.alexkras.com/19-git-tips-for-everyday-use/ [...]

Reply

Weekly News for Designers (N.333) | Digital Develop says:

May 12, 2016 at 9:38 am

[...] 19 Tips For Everyday Git Use By Alex Kras. [...]

Reply

Weekly News for Designers (N.333) | Tech + says:

May 10, 2016 at 7:38 am

[...] 19 Tips For Everyday Git Use By Alex Kras. [...]

Reply

Weekly News for Designers (N.333) - Diseño Multimedia | KornukopiaDiseño Multimedia | Kornukopia says:

May 8, 2016 at 1:31 pm

[...] 19 Tips For Everyday Git Use By Alex Kras. [...]

Reply

19 Tips For Everyday Git Use - Freakinthecage Webdesign Stuttgart - Der Blog says:
May 8, 2016 at 10:54 am
[...] Sourced through Scoop.it from: https://www.alexkras.com [...]

Reply

Weekly News for Designers (N.333) - ELISION says:
May 8, 2016 at 2:54 am
[...] 19 Tips For Everyday Git Use By Alex Kras. [...]

Reply

Weekly News for Designers (N.333) - UX Checklist, Bootstrap 4 Cheat Sheet, Flexbox Grid says:
May 8, 2016 at 12:09 am
[...] 19 Tips For Everyday Git Use By Alex Kras. [...]

Reply

Git Mind Map says:
May 3, 2016 at 4:15 pm
[...] Cheatsheet. Both are aimed towards beginners. If you are an intermediate Git user, you might find 19 Tips For Everyday Git Use to be more [...]

Reply

Java Weekly 18/16: Wildfly Logger, Exceptions in JAX-RS & Git says:
May 2, 2016 at 3:40 am
[…] you use git? Then you should have a look at Alex Kras' 19 Tips For Everyday Git Use. I've already bookmarked that page to have it at hand when I need […]

Reply

Link blog: scott-aaronson, git, dance, jazz – Name and Nature says:
April 25, 2016 at 12:13 am
[…] 19 Tips For Everyday Git Use […]

Reply

issue #25: Shift, Gentoo on Tesla, RPerl, BSD explained and many more! - Cron Weekly: a weekly newsletter for Linux and Open Source sysadmins says:
April 24, 2016 at 2:02 am
[…] 19 Tips For Everyday Git Use […]

Reply

Link: 19 Tips For Everyday Git Use – Braveterry says:
April 21, 2016 at 4:00 pm
[…] Link: 19 Tips For Everyday Git Use […]

Reply

Git Tips - weigandtLabs says:
April 21, 2016 at 9:34 am
[…] Git Tips – https://www.alexkras.com […]

Reply

19 Tips For Everyday Git Use – Internet and Tecnnology Answers for Geeks says:
April 20, 2016 at 6:11 pm
[…] by /u/caligolae [link] […]

Reply

git links | Seemed Worthwhile At the Time says:
April 20, 2016 at 2:23 pm
[…] 19 Tips For Everyday Git Use and hacker news comments– Alex Kras (September, 2015) […]

Reply

Git Tips – Teknoids News says:
April 20, 2016 at 12:29 pm
[…] Extract a file from another branchgit show some-branch:some-file.js […]

Reply

19 Tips For Everyday Git Use | Raony Guimarães says:
April 20, 2016 at 6:28 am
[…] Parameters for better logging git log –oneline –graph […]

Reply

Want some free Git resources? – Sarah Pressler says:

March 9, 2016 at 1:24 pm
[...] – https://www.alexkras.com/19-git-tips-for-everyday-use/ [...]

Reply

日常使用 Git 的 19 个建议 | 從前有個浪得虛名的程序員 says:
December 16, 2015 at 1:34 pm
[...] 2015年12月16日 浪得虚名 发表回复 本文由 伯乐在线 – zaishaoyi 翻译，艾凌风 校稿。未经许可，禁止转载！ 英文出处：Alex。欢迎加入翻译组。[...]

Reply

Git : Tips & Tricks - Liip Blog Liip says:
December 11, 2015 at 3:04 pm
[...] Git tips for everyday use [...]

Reply

5 claves para utilizar bien Git y no sólo aparentarlo - donmik says:
December 4, 2015 at 9:31 am
[...] Si quieres aprender más cosas sobre Git pero no el libro, leéte este artículo: 19 Tips For Everyday Git Use. [...]

Reply

19 Tips For Everyday Git Use | Makzan says:
November 30, 2015 at 8:59 am
[...] Link: 19 Tips For Everyday Git Use [...]

Reply

Confluence: 마케팅워킹그룹 says:
September 30, 2015 at 8:42 am
개발 참고 URL

CSS Transition 이미지전환 / 마우스오버시 이미지 전환 효과 등등에 관한 CSS

Reply

19 Tips For Everyday Git Use | phamduyphong says:
September 16, 2015 at 4:11 am
[...] Three stages in git, and how to move between them [...]

Reply

Les liens de la semaine – Édition #149 | French Coding says:
September 14, 2015 at 11:01 am
[...] 19 conseils d'utilisation de git. [...]

Reply

Links of the Week, W38 2015 | One More Game-Dev and Programming Blog says:
September 13, 2015 at 4:18 am
[...] 19 Tips for Everyday Git Use. [...]

Reply

Git - RITM says:
September 8, 2015 at 4:51 pm
[...] https://www.alexkras.com/19-git-tips-for-everyday-use/ [...]

Reply