

git-notes

September 3, 2018

Contents

1	Help	1
1.1	CLI	1
2	Architecture	2
3	Configuration	3
3.1	config ranges	3
3.2	configurable properties	3
3.3	config listing	4
3.4	customised prompt	4
3.5	aliases	4
4	Common Tasks	5
4.1	diff	5
4.2	show	5
4.3	delete	5
4.4	rename/move	6
4.5	commit	6
4.6	undo	6
4.7	git reset HEAD	7
4.8	HEAD in detached mode	7
4.9	untracked files - delete	7
4.10	.gitignore	8
4.11	untracking files	8
4.12	referencing ancestor commits	8
4.13	content listing	8
4.14	git log	9
4.15	stashing	10
5	Branches	11
5.1	new branch	11
5.2	delete	11
5.3	list of branches	11
5.4	switching a branch	12

5.5	rename	12
5.6	merging	12
5.6.1	merge	12
5.6.2	resolving conflicts	12
6	Remotes	14
6.1	origin/master	14
6.2	Managing URLs	14
6.2.1	list	14
6.2.2	add	14
6.2.3	remove	15
6.3	Collaboration	15
6.3.1	create local and remote repositories	15
6.3.2	send	15
6.3.3	receive	16
6.3.4	remote branches	16
7	GitHub	18
7.1	help	18
7.2	GitHub pages	18
8	Resources	19

Chapter 1

Help

1.1 CLI

```
git <command> -h
```

Chapter 2

Architecture

- working directory
the 'visible' part which lets do modification
 - **staging index**
'waiting area' - mediates between working area and local repo. Initially in sync with the repo, but allows to store changes done in working area and then to send the changes to local repo 'in-one-go'
Can be behind the repo after some new content has been fetched.
 - **stashing area**
'a pocket', intended to store changes that haven't been committed. useful when we need to switch a branch, but the changes are not ready to commit yet.
- local repo this is a project tree stored locally. Two pointers operate on it:
 - **HEAD**
points to the commit which copy is present in the working directory
 - **remote_repo/remote_branch**
usually 'origin/master' - points to the last commit fetched from a remote repo
- remote repo

Chapter 3

Configuration

3.1 config ranges

- System
`git config --system`
- User
`git config --global`
- Project
`git config`
- edit examples
`git config --global user.email "abc@mail.com"`
`git config --global core.editor "vim"`

3.2 configurable properties

- user
 - `user.name=`
 - `user.email=`
- core
 - `core.editor=`
 - `core.excludesfile=`
- color

```
color.ui=
```

- remote URLs
look at 'Remote' chapter, 'Managing URLs' section

3.3 config listing

- all

```
git config --list
```
- specific

```
git config user.email
```

3.4 customised prompt

UNIX-like

```
export PS1='\W$__git_ps1 "($s)" > '
add it to .bashrc (.bash_profile):

# uncomment current prompt export if exist
# export PS1="some_user"
....
# add at the end of the script
export PS1='\W$__git_ps1 "($s)" > '
```

Windows

similar format should be preconfigured. to get exact the same:

```
export PS1='\W$__git_profile "(%s)" > '
```

save as **.bash_profile** in **/Users/current_user** dir

3.5 aliases

```
git config --<scope> alias.<abbreviation> original_command
```

put original command between double quotes if it contains space(s)

Chapter 4

Common Tasks

4.1 diff

- `git diff <file_name>`
compares **working** directory against **staging** area - for each file, on line-by-line basis
- `git diff --staged <file_name>` (OBSOLETE: `git diff --cached`)
compares **staging** against **repository**
- `git diff <SHA>`
compares state in a particular commit with the current state in a working dir
- `git diff <SHA>..<SHA>`
compares two particular commits
- use switches to change the way the differences are displayed
`git diff --color-words <file_name>`

4.2 show

For commits it shows the log message and textual diff. It also presents the merge commit in a special format as produced by `git diff-tree -cc`. (git doc, <https://git-scm.com/docs/git-show>)

4.3 delete

- `git rm file_name`

if present - removes definitely the file from working directory (it doesn't go to the trash bin)
stages 'delete' operation

4.4 rename/move

- `git mv old_name new_name`

4.5 commit

- `commit -a`
combines 'add' and 'commit'. Ignores deleted and new files.

4.6 undo

- in working directory
 - `git checkout <commit_SHA> <branch_name> <file_name>`
 - `git checkout -- file_name`
to redo to the same state as at the pointer in repository
 - `git checkout file_name`
branch name not required if it is current one and there is no branch with the same name as the file
 - `git checkout SHA_number <branch_name> <file_name>`
sets both **working directory** and **stage area** to the state in a particular commit identified by SHA
 - `git revert SHA_number`
reverts all changes done in a particular commit - updates working directory and makes **new commit** (this can be switched out) with those reverted changes. **revert** is for simple changes, **merge** for complex.
 - * `git revert SHA_number --in`
doesn't make commit. allows to append further modification to next commit
 - `git reset ...`
look at 'reset' subsection
- in staging area
 - `git reset HEAD <file_name>`

removes all changes when no file provided. look 'reset' section for more about reset.

- in **repository**

```
git commit --amend <-m "...">
```

only **last commit** editable! it appends staged changes to current commit, the message can be changed as well

4.7 git reset HEAD ...

'Rewind' - sets the head on a particular commit; consecutive commits and logs becomes invisible. They can be accessed only by their SHA.

- `git reset --soft ...`
sets pointer to new position, makes no changes to working dir and index (staging area)
- `git reset ...`, `git reset --mixed ...`
default mode. sets the pointer; sets staging index to match repository at the pointer. working directory stays intact
- `git reset --hard`
sets the pointer; sets both working dir and staging index to match repository at the pointer. later changes, commits, are lost

4.8 HEAD in detached mode

This means that the HEAD pointer is on some particular commit **outside** any branch. It could've happen after particular commit was checked out using its SHA1. To attach back **checkout** any branch.

4.9 untracked files - delete

- `git clean -n`
tests, which files will be deleted
- `git clean -f`
destructive command - permanently removes untracked files

4.10 .gitignore

- syntax

* ? [abc] [a-c1-6] !

starts a comment line; blank lines are ignored

- project scope of ignore

create and edit .gitignore (without extension) in the repository root

- per-user ignore

```
git config --global core.excludesfile <file_path>
```

to tell where .gitignore file is, then edit the file. typical filepath:

```
~/.gitignore_global <- Linux  
/Users/user_name/.gitignore <- Windows
```

4.11 untracking files

to stop tracking the file by staging index:

```
git rm --cached <file_name>
```

then add the file to **.gitignore** in order to prevent git to ask to add the file in the future.

the file is still kept in **working dir** and **repository**, but it isn't tracked for further changes by staging index

4.12 referencing ancestor commits

for instance from HEAD:

```
HEAD~ = HEAD~1 = HEAD^  
HEAD~2 = HEAD^^  
HEAD~3 = HEAD^^^
```

4.13 content listing

```
git ls-tree <tree-ish(es)>
```

tree-ish - branch (last commit), SHA, tag, dir

4.14 git log

- my preferred

```
git log --online --graph --all --decorate
```

- short

```
git log --oneline
```

- tree of branches

```
git log --graph
```

- detailed

```
git log -p
```

shows changes as shown by **diff** command

- particular tree-ish in detail

```
git show <tree-ish>
```

shows:

- **content** of files, directories
- particular **commits** like by "log -p"

- time

```
git log --since=".." --until="..."
```

- some popular:

```
git log --grep="..."  
git log --author="..."
```

- more at git help

4.15 stashing

- saving in stash

```
git stash save "some message"
```

like **git reset --hard HEAD**, but changes are stashed

- listing a stash

```
git stash list
```

stash@{0}: <branch_name>: ".." - stash reference

- showing changes saved in stash

```
git stash show -p <stash_ref>
```

Shows what changes this stash would apply

stash is not bound to any commit. can be taken from one working dir and applied to some other working dir

- applying changes from stash

```
git stash pop
git stash apply
```

apply changes to current working dir;

pop drops stash, **apply** allows multiple use

- removing from stash

```
- git stash drop stash@{id}
- git stash clear
  removes all
```

Chapter 5

Branches

5.1 new branch

- `git branch branch_name`
`git branch...` creates new branch
- `git checkout -b branch_name`
`git checkout -b...` creates new branch and switches to it

5.2 delete

Must not be current branch

- `git branch -d branch_name`
works for **fully merged** branches only
- `git branch -D branch_name`
works for **unmerged** branches, too

5.3 list of branches

- `git branch`
lists **local** branches
- `git branch -r`
lists **remote** branches
- `git branch -a`
lists **all** branches (local + remote)

5.4 switching a branch

```
git checkout branch_name
```

- *'swapping context'*
- working dir should be *clean* - all modifications should be committed, stashed or discarded

5.5 rename

```
git branch -m old_name new_name
```

-m! Not -mv!

5.6 merging

5.6.1 merge

1. ensure this is a destination branch
2. ensure the working directory is clean
3. `git merge <source_branch>`
 - `git branch --merged`
returns a list of fully incorporated branches - they can be merged **fast-forward (ff-merge)**
 - `git merge --no-ff <branch_name>`
created merge commit even if it is ff-merge
 - `git merge --ff-only <branch_name>`
merges only if ff-merge is possible; aborts otherwise

5.6.2 resolving conflicts

- abort

```
git merge --abort
```
- resolve manually
 - open files, find conflict spots, manually fix them; useful:

```
git show <object>
```

, and put SHA1 as an object; look section 2.2
 - stage modified files

- commit
 - * this is merge commit - merge completed!
 - * message unnecessary
- resolving using tools
 - `git mergetool --tool=...`
 - type **git mergetool** to get list of available/recommended tools;
 - a tool can be added to the config file

Chapter 6

Remotes

6.1 origin/master

- this is a pointer to last fetched commit.
- need to be in sync with local and remote master before push:
 - fetch (or pull) to sync with a remote
 - merge locally to resolve conflicts and sync locally (master and origin/master pointing to the same commit)
 - repeat fetch+merge (or pull) until all conflicts are resolved and all syncs established; this makes ff-merge of remote master and master/origin possible

6.2 Managing URLs

6.2.1 list

- `git remote`
returns remote identifiers
- `git remote -v`
detail info including URLs

6.2.2 add

`git remote add <remote_name> <URL>`

it's a convention to call primary remote **origin**.

- https URL:
`https://github.com/<user_name>/<repo_name>.git`

- ssh URL:

```
git@github.com:<user_name>:<repo_name>.git
```

6.2.3 remove

```
git remote -rm <remote_name>
```

remote name as listed by

```
git remote -v
```

6.3 Collaboration

6.3.1 create local and remote repositories

remote repo from local one

1. `git init`
in the root dir of a project
2. `git remote add <remote_repo_name> <URL>`
look at Remotes → add , sec. 4.2.2
3. `git push -u origin master`
pushes local content to remote repo; -u makes local and remote branches in sync; look at Remotes → send

clone remote repo to local repo

- `git clone <remote_URL>`
 - creates local folder using remote repo name
 - clones remote project to the folder
- `git clone <remote_URL> <folder_name>`
 - creates local folder with the specified name
 - clones remote project to the folder

6.3.2 send

It works only if ff-merge is possible on the remote side.
Look at Remote → origin/master.

- `git push <remote_repo> <remote_branch>`
usually:

```
git push origin master
```

or

```
git push
```

if current branch is tracked

- `git push -u ...`
also sets a local branch to track a remote

6.3.3 receive

- `git fetch + git merge`
 - merge works exactly the same as for any other merge
 - `git pull`
does exactly the same if ff-merge is possible (no conflicts)
- `git fetch <remote_name>`
 - we can omit remote name if there's one remote only
 - Non-destructive!
 - updates origin/master, synchronises with remote repo
 - origin/master doesn't reflect current state of a remote repo, it's only a copy of the last fetched state.
 - fetch doesn't do any changes neither to local repo, nor staging area, nor local working dir
- merge with origin/master the same way as with any other branch
 - `git show origin/master`
shows what was fetched
 - `git diff master..master/origin`
shows changes to apply locally

6.3.4 remote branches

- list
 - `git branch -r`
remote only
 - `git branch -a`
all

- create

```
git branch local_branch_name remote_name/remote_branch_name
git checkout -b local_branch_name remote_name/remote_branch_name
```

- creates local and remote branch at the same time
- make the local one tracked and in sync with the remote one
- **git checkout -b...** also switches to this new branch

- delete

```
git push origin --delete remote_branch_name
or (older version)
```

```
git push origin :remote_branch_name
(push 'nothing' to a remote branch)
```

Chapter 7

GitHub

7.1 help

- GitHub doc:
`file:///D:/IT/version%20control/git/web/GitHub%20Help.htm`
- Short github tutorials:
`file:///D:/IT/version%20control/git/web/GitHub%20Guides.htm`
- Customizing GitHub Pages <https://help.github.com/categories/customizing-github-pages/>

7.2 GitHub pages

To create and host web pages based on GitHub repos:
<https://guides.github.com/features/pages/>

Chapter 8

Resources

- official documentation:
<https://git-scm.com/docs>