# git-notes

m.sz

September 28, 2018

# Contents

# Part I

# Help

# Chapter 1

# CLI

Short version of help:

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

Full manual:

```
git help <command>
or
man git <command>
```

# Part II

# Architecture

# Chapter 2

# Working Directory

This is a single checkout of one version of the project. It is our working area to do current modifications. Coloquially - it is what we see on the screen when working on the project. It contains:

- project objects
- git *metadata*

**Modified** means that changes are neither *staged* nor *committed*.

# Chapter 3

# Staging Area, Index

Mediates between working area and local repo - contains those **changes which are intended to commit to *Local Repo***.

- **in sync** with the repo after *checkout*
- **behind** the repo after something is *fetched*
- **ahead** after some changes were *added*

**Staged** means *modified objects* which are yet not committed.

# Chapter 4

# Stashing Area

'A pocket', intended to store changes that haven't been commited. useful when we need to switch a branch, but the changes are not ready to commit yet.

# Chapter 5

# Local Repo

This is a project database stored locally. It is a **DAG** containing all project snapshots. 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

# Chapter 6

# Remote Repo

# Chapter 7

# ./.git

This is a subdirectory created in project root directory by *git init* command. Its location can be customised by editing GIT_DIR environmental variable or by executing

```
git init --separate-git-dir
```

Contains initially:

- **.gitconfig**, project-local config variables
- **HEAD**, a file with a pointer to the last checked out commit.

## 7.1 HEAD

Location - /.git.
This is a file which contains a pointers to the currently checked out commit(in *Local Repo*). **One pointer for each tracked file.**

# Part III

# Configuration

## 7.2 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"
  ```

## 7.3 configurable properties

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

  `color.ui=`

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

## 7.4 config listing

- all

  ```
  git config --list
  ```

- specific

  ```
  git config user.email
  ```

## 7.5 .gitignore

```
https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository
https://git-scm.com/docs/gitignore
```

The file specifies the file which should stay ignored. **The files already tracked are not affected!**. In order to untrack them use

```
git -rm --cached
```

See: *'Common Tasks.delete.rm –cached'*.

**Glob** patterns (it is simplified Regex) is a format used to specify files to be excluded from tracking.

- syntax
  - `* ? [abc] [a-c1-6] !`
  - # starts a comment line; blank lines are ignored
  - ** means 'match all directories'
- project scope of ignore

  create and edit .gitignore (without extention) in the repository root
- per-user ignore

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

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

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

### 7.5.1 Glob Patterns

- **characters**

  ```
  [xyz] - ignore all string of 'x',' y' or 'z', where x, y, z may be
  any characters (alphabetical, numerical,
  special characters)
  x-y - any character in range from x to y...
  ```
- **wildcards, negation**

  ```
  ? - any single character
  * - arbitrary number of any characters
  ! - negation...
  ```

- **directories**

  ```
  ** - any dir path
  / - if starts a pattern - current directory
                            avoids recursivity
    - if ends a pattern - specify the exact directory
                          the pattern is addressed to
  abc/ - all directories rooted by abc directory
  ```

Good source of predefined .gitignore files:
`https://github.com/github/gitignore`

## 7.6 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 PS!='\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

## 7.7 aliases

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

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

# Part IV

# Basic Tasks

# Chapter 8

# Creating a Local Repo

## 8.1 Transforming Existing Dir into Local Repo

```
git init
```

Creates new local repo rooted by current dir. Creates /.git directory.

### 8.1.1 Empty Repo in the Specified Dir

```
git init <dir>
```

## 8.2 Clonning Remote Repo

See: *Remotes.Managing Repo's*

## 8.3 git init –bare dir

Creates a repo without Working Area. The common usage is to create **central repo**. No commits, edits, just pushes and pulls only.

## 8.4 Local Repo from a Template

```
git init --template template=template_dir
```

Uses a template to create .git directory. A default template usually reside in */usr/share/git-core/templates*. It is a good reference for how to create a custom template. For more see: *Git Hook*

# Chapter 9

# Commits

## 9.1 Committing

### 9.1.1 With Attached Message

```
commit -m 'message'
```

### 9.1.2 Without Staging

```
commit -a
```

Combines 'add' and 'commit'. Ignores deleted and new files. In order to include new files perform '*git add ..*' first.

## 9.2 Modification

Permitted for last commit only. Reasonable is to use this for minor changes only. In all other cases it is preferable to revert and then make a proper commit.

### 9.2.1 Replacing Last Commit

```
git commit --amend
```

Replaces last commit with current staged content, last message is a starting point for new one. If any changes are present from last commit they are appended to it without creating new commit. If there are no changes, then it is effectively just an edit of last message.

### 9.2.2 Changing a Message

```
git commit --amend 'message'
```

This replaces the message, too.

## 9.3 Review

### 9.3.1 gitk

Wow!

### 9.3.2 log

Lists commits.

- 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

### 9.3.3 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` )

# Chapter 10

# Managing Staging Area

## 10.1 Staging, Tracking

```
git add ...
```

Git sees files not present in previous commits as 'untracked'. As a consequence of this fact is that *add* command works two-fold:

- **tracking** - it starts tracking untracked files; it changes file's status from *untracked* to *tracked.*
- **staging** - it stages those *modified* files that are intended to be included in next *commit.*

This command defines which files are assigned to be included in next commit (**to be staged**):

- **untracked** files; **all new files are *untracked* initially**
- **deleted** files
- **updated**

All comits create a stream of snapshots for each file individually - from their creation until their deletion.

### 10.1.1 add -A

Stages **all** changes, including **new files in entire project**.

### 10.1.2 add .

Stages te changes present in current directory and subdirectories. **Ignores changes in hiher directories of the project!**

## 10.2 Unstaging

### 10.2.1 An Arbitrary File

```
git reset HEAD filename
```

This sets a staged file to be the same as its current committed version. It effectively **discards** all changes which have been applied to this file in **staged area** after last commit.

# Chapter 11

# status -s

It's a short version.

- A - added
- M - modified
- left column - staging area item right column - working area.

Examples:
M_ - modified & staged
_M - modified & not-staged
MM - modified, staged, again modified in working area.

# Chapter 12

# diff

- `git diff <file_name>`

  compares **working** directory against **staging** area - for each file, on line-by-line basis.
  Displays the differences still **unstaged** only, **not added modifications**.

- `git diff --staged <file_name> (OBSOLETE: git diff --cached)`

  compares **staging** against **repository**

- `git diff <SHA>`

  compares state in a particular commit wit 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>
```

# Chapter 13

## ...

## 13.1 delete

- `git rm file_name`
  - **Removes definitely the file from working directory** (it doesn't go to the trash bin)
  - there are **no staged modifications**
- `git rm --cached <file_name>`

  Effectively removes files from staging area. Usefull to untrack a file.

  ```
  -q
  --quiet
  ```

  Surpresses default behaviour - one output line displayed for each removed file. rm –cached <file>Effectively removes files from staging area.
- `git rm -f`

  ..*forced* remove. For files **modified** and **staged**.

See '***Common tasks.untracking files***' to know how to delete from staging area only.

## 13.2 rename/move

- `git mv old_name new_name`

## 13.3 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) whith 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

## 13.4 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

## 13.5 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.

## 13.6 untracked files - delete

**Destructive command - permanently removes untracked files!**

- `git clean -n`

  tests, which files will be deleted

- `git clean -f`

## 13.7 untracking files

to stop tracking the file by staging index:

- 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
- **rm –cached <file>**
  Effectively removes files from staging area.

  `-q`
  `--quiet`

  Surpress default behaviour - one output line displayed for each removed file.

## 13.8 referencing ancestor commits

for instance from HEAD:

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

## 13.9 content listing

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

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

## 13.10 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 aplied 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

# Part V

# Branches

### 13.11 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

### 13.12 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

### 13.13 list of branches

- `git branch`

  lists **local** branches

- `git branch -r`

  lists **remote** branches

- `git branch -a`

  lists **all** branches (local + remote)

### 13.14 switching a branch

`git checkout branch_name`

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

### 13.15 rename

`git branch -m old_name new_name`

**-m**! Not **-mv**!

## 13.16 merging

### 13.16.1 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

### 13.16.2 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

# Part VI

# Remotes

## 13.17 Managing Repo's

### 13.17.1 Creating Local Repo from Remote

- implicit

  ```
  git clone <remote-repo-url> <local-dir>
  ```

  - *local-dir* is optional
  - *origin* is set as an identifier for the remote repo.

- explicit

  ```
  git remote add <id> <remote-repo-url> <local-dir>
  ```

  - *local-dir* is optional
  - *id* sets custom identifier for the remote repo. **origin** is a default identifier.

  ```
  git fetch <id>
  ```

  - fetches remote repo using previous specified *remote-repo-url* for this *id*
  - makes remote repo locally accessible as *id/master*.
  - *id* may be ommitted when defaults to *origin*
  - manual *merge* still needed

    ```
    or
    git pull
    ```

    * combines *fetch* + *merge*
    * current **local branch** must be set up to **track a remote branch.**

### 13.17.2 Remote from Local

```
git remote add ...
git push -u <remote_name master>
```

## 13.18 Managing URLs

### 13.18.1 list

- `git remote`
  returns a list of remote identifiers
- `git remote -v`
  detailed info including URLs

### 13.18.2 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`

### 13.18.3 rename

*Origin* is an informal default identifier for remote repo. It can be change:

`git remote rename old-remote-id new-remote-id`

### 13.18.4 remove

`git remote -rm <remote_name>`

remote name as listed by
`git remote -v`

## 13.19 Collaboration

### 13.19.1 Setting Upstream

If repo was cloned then there is no need to set upstream - the address of the remote repo used to clone the project is used. Otherwise we need to *push -u/push –set-upstream* first.

### 13.19.2 push

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

**push -u**

Sets a local branch to track a remote one. Necessery when local and remote repos are not synchronised yet (it is first push). This is an equivalent of *git push –set-upstream.* It additionaly sets upstream URL for a branch.

### 13.19.3 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

### 13.19.4 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)

# Part VII

# GitHub

# Chapter 14

# Resources

- 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/`

# Chapter 15

# GitHub pages

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

# Part VIII

# Git Hook

https://www.atlassian.com/git/tutorials/git-hooks

# Part IX

# Resources

- official documentation:
  `https://git-scm.com/docs`
- tutorials:
  - `https://www.atlassian.com/git/tutorials/what-is-version-control`