Git Memento

Author:	Jérémie DECOCK
Contact:	jd.jdhp@gmail.com
Revision:	0.1
Date:	20/05/2016
License:	Creative Commons 4.0 (CC BY-SA 4.0)

Table of Contents

1 Setup Git

1.1 Configuration files

`.git/config`
: Repository-specific configuration settings (manipulated with the `--file` option of `git config`)

`~/.gitconfig`
: User-specific configuration settings (manipulated with the `--global` option of `git config`)

`/etc/gitconfig`
: System-wide configuration settings (manipulated with the `--system` option of `git config`)

1.2 Configuration example

User information:

```
git config --global user.name "Jeremie DECOCK"
git config --global user.email "jd.jdhp@gmail.com"
```

Setup push.default (see http://stackoverflow.com/questions/23918062/simple-vs-current-push-default-in-git-for-decentralized-workflow):

```
git config --global push.default simple
```

Setup colors:

```
git config --global color.branch auto
git config --global color.diff auto
git config --global color.grep auto
git config --global color.interactive auto
git config --global color.status auto
```

Some useful aliases:

```
git config --global alias.ci commit
git config --global alias.co checkout
git config --global alias.st status
git config --global alias.br branch
git config --global alias.unstage "reset HEAD --"
git config --global alias.graph "log --oneline --decorate --graph --all"
```

Add a GPG key (see https://help.github.com/articles/telling-git-about-your-gpg-key/ and https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work ; for GitHub users see https://github.com/blog/2144-gpg-signature-verification):

```
git config --global user.signingkey PUBLIC_KEY_ID
```

1.3 List current configuration settings

```
git config -l
```

1.4 Some advanced aliases

Some advanced aliases are available at https://git.wiki.kernel.org/index.php/Aliases).

Use graphviz for display (https://git.wiki.kernel.org/index.php/Aliases#Use_graphviz_for_display):

```
[alias]
    graphviz = "!f() { \
        echo 'digraph git {' ; \
        git log --pretty='format:  %h -> { %p }' \"$@\" | \
            sed 's/[0-9a-f][0-9a-f]*/\"&\"/g' ; \
        echo '}'; \
        }; f"
```

1.5 Get tab completion of branches, tags, subcommands, ...

Git contains a set of completion scripts for *bash* (`git-completion.bash`), *tcsh* (`git-completion.tcsh`) and *zsh* (`git-completion.zsh`). Usually these files are already installed in the "git-core" directory of your git installation. In case, you can find them with the following command:

```
find / -type f -name "git-completion.*" 2> /dev/null
```

or you can download them at https://github.com/git/git/tree/master/contrib/completion

Let's say you use Bash and your completion script is in `/usr/share/git-core/` (adapt the following lines to your case), then to activate git completion, simply add the following lines to your shell startup file (e.g. `~/.bashrc`):

```
# Define completion for Git
git_completion_path=/usr/share/git-core/git-completion.bash
[ -r ${git_completion_path} ] && source ${git_completion_path}
```

1.6 Show the current branch in the shell prompt

Git contains a script to show the current branch in the shell prompt (`git-prompt.sh`). Usually this file is already installed in the "git-core" directory of your git installation. In case, you can find it with the following command:

```
find / -type f -name "git-prompt.sh" 2> /dev/null
```

or you can download it at https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh (note for *Debian8* users: the script is provided by the `git` package in `/usr/lib/git-core/` and has been renamed `git-sh-prompt`).

Let's say you use Bash (otherwise adapt the following lines to your case). To activate git information in prompt, simply add the following lines at the end of your shell startup file (e.g. `~/.bashrc`):

```
# Define prompt for Git
git_prompt_path=/usr/share/git-core/git-prompt.sh
[ -r ${git_prompt_path} ] && source ${git_prompt_path}

# Define the prompt shell
PS1='\u@\h:\w`__git_ps1 " (%s)"`\$ '
```

or if you want a bit of color, replace the last line by:

```
PS1='\u@\h:\w\[\033[31;1m\]`__git_ps1 " (%s)"`\[\033[0m\]\$ '
```

2 Ignore some files

2.1 Exclusion patterns files

- *Project specific* exclusion patterns should be shared in `.gitignore` files
- *User specific* exclusion patterns should be retained in the `.git/info/exclude` file

2.2 Syntax of exclusion patterns files

- Blank lines are ignored
- Lines *starting* with # are ignored (used for comments)
- Directory names are ended with `/` (doesn't work with symbolic links)
- Globbing characters work like in Unix shells (`*`, ...)
- When a `!` starts a line, the meaning of its pattern is inverted (i.e. files are explicitly kept instead of being explicitly ignored)

3 Logging

3.1 Show all commits

```
git log
```

3.2 List the changes during the last two weeks

```
git log --since="2 weeks ago"
```

3.3 List the changes made on a given file

```
git log FILE_PATH
```

3.4 List the changes made on files contained in a given directory

```
git log DIRECTORY_PATH
```

3.5 List the changes containing a given string pattern

List the commits where the "foo()" string has been added or deleted:

```
git log -S'foo()'
```

4 Viewing difference

4.1 Show changes in the working tree

Show changes in the working tree that haven't been staged or committed yet:

```
git diff
```

Show changes in the working tree that have been staged:

```
git diff --cached
```

Show changes between the working tree (staged or not) and the repository:

```
git diff HEAD
```

4.2 Show changes between two git objects

Show changes between two given *commits*, *tags*, *branches*, *trees* or *blobs*:

```
git diff A B
```

or:

```
git diff A..B
```

where A and B can be *commits*, *tags*, *branches*, *trees* or *blobs*.

Note that A is supposed to be older than B as what is shown is the modification applied to go from A to B. If you reverse the order of A and B, addition lines (+) will become deletion lines (-) and vice versa.

You can also use:

```
git diff A B FILENAME1 [FILENAME2 ...]
```

or:

```
git diff A..B FILENAME1 [FILENAME2 ...]
```

or:

```
git diff A B -- FILENAME1 [FILENAME2 ...]
```

or:

```
git diff A..B -- FILENAME1 [FILENAME2 ...]
```

to get differences from A to B for specific files or directories `FILENAME1`, `FILENAME2`, ...

The `--` notation is only required the files you want to compare have contentious name (like "-f").

Also note that `FILENAME1`, `FILENAME2`, ... have to be paths relative to the current working directory. Paths relative to the root of the repository won't work.

To compare a git object A to HEAD, simply use:

```
git diff A
```

Examples:

```
git diff ff20b  ea76d ./src/main.c
git diff ff20b  ea76d
git diff ff20b..ea76d
git diff v1  v2
git diff v1  v2 ./src/
git diff v1..v2 ./src/
git diff master  expermiental
git diff master  expermiental ./src/main.c ./Makefile
git diff master..expermiental ./src/main.c ./Makefile
```

4.2.1 Show changes between two commits

```
git diff COMMIT1 COMMIT2
```

or:

```
git diff COMMIT1..COMMIT2
```

Examples:

```
git diff a80b0d8  04079b1
git diff a80b0d8..04079b1
```

4.2.2 Show changes between two commits for a given file

```
git diff COMMIT1 COMMIT2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff COMMIT1 COMMIT2 -- FILENAME1 [FILENAME2 ...]
```

or:

```
git diff COMMIT1..COMMIT2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff COMMIT1..COMMIT2 -- FILENAME1 [FILENAME2 ...]
```

where `FILENAME1`, `FILENAME2`, ... are file paths or directory paths relative to the current working directory.

Examples:

```
git diff a80b0d8  04079b1 ./src/main.c
git diff a80b0d8  04079b1 ./src/main.c ./Makefile
git diff a80b0d8  04079b1 ./src/
git diff a80b0d8..04079b1 ./src/main.c
git diff a80b0d8..04079b1 -- ./src/main.c
```

4.2.3 Show changes between two tags

```
git diff TAGNAME1 TAGNAME2
```

or:

```
git diff TAGNAME1..TAGNAME2
```

Examples:

```
git diff v1.0  v2.0
git diff v1.0..v2.0
```

4.2.4 Show changes between two tags for a given file

```
git diff TAGNAME1 TAGNAME2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff TAGNAME1 TAGNAME2 -- FILENAME1 [FILENAME2 ...]
```

or:

```
git diff TAGNAME1..TAGNAME2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff TAGNAME1..TAGNAME2 -- FILENAME1 [FILENAME2 ...]
```

where `FILENAME1`, `FILENAME2`, ... are file paths or directory paths relative to the current working directory.

Examples:

```
git diff a80b0d8  04079b1 ./src/main.c
git diff a80b0d8  04079b1 ./src/main.c ./Makefile
git diff a80b0d8  04079b1 ./src/
git diff a80b0d8..04079b1 ./src/main.c
git diff a80b0d8..04079b1 -- ./src/main.c
```

4.2.5 Show changes between two branches (local or remote)

```
git diff BRANCH1 BRANCH2
```

or:

```
git diff BRANCH1..BRANCH2
```

or with remote branches:

```
git diff [REMOTENAME1/]BRANCH1 [REMOTENAME2/]BRANCH2
```

or:

```
git diff [REMOTENAME1/]BRANCH1..[REMOTENAME2/]BRANCH2
```

Examples:

```
git diff bugfix
git diff master  bugfix
git diff master..bugfix
git diff upstream/master  bugfix
git diff upstream/master..origin/master
```

4.2.6 Show changes between two branches (local or remote) for a given file

```
git diff BRANCH1 BRANCH2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff BRANCH1 BRANCH2 -- FILENAME1 [FILENAME2 ...]
```

or:

```
git diff BRANCH1..BRANCH2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff BRANCH1..BRANCH2 -- FILENAME1 [FILENAME2 ...]
```

or with remote branches:

```
git diff [REMOTENAME1/]BRANCH1 [REMOTENAME2/]BRANCH2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff [REMOTENAME1/]BRANCH1 [REMOTENAME2/]BRANCH2 -- FILENAME1 [FILENAME2 ...]
```

or:

```
git diff [REMOTENAME1/]BRANCH1..[REMOTENAME2/]BRANCH2 FILENAME1 [FILENAME2 ...]
```

or:

```
git diff [REMOTENAME1/]BRANCH1..[REMOTENAME2/]BRANCH2 -- FILENAME1 [FILENAME2 ...]
```

where `FILENAME1`, `FILENAME2`, ... are file paths or directory paths relative to the current working directory.

Examples:

```
git diff bugfix
git diff master  bugfix
git diff master..bugfix
git diff upstream/master  bugfix
git diff upstream/master..origin/master

git diff bugfix  ./src/main.c
git diff master  bugfix ./src/main.c ./Makefile
git diff master..bugfix ./src/
git diff upstream/master  bugfix ./src/main.c
git diff upstream/master..origin/master -- ./src/main.c
```

4.3 Show changes with the first common ancestor of two objects

Show changes between B and the first common ancestor of A and B:

```
git diff A...B
```

4.4 Show changes between two git objects using external tools with graphical interface

Git support many *external diff tools* out of the box to show differences between two git objects, including *opendiff*, *kdiff3*, *tkdiff*, *xxdiff*, *meld*, *kompare*, *gvimdiff*, *diffuse*, *diffmerge*, *ecmerge*, *p4merge*, *araxis*, *bc*, *codecompare*, *vimdiff* and *emerge* (the default one is *opendiff*).

To select which tool to use, type:

```
git config [--global] diff.tool TOOL_NAME
```

For instance:

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

To show changes between two given *commits*, *tags*, *branches*, *trees* or *blobs* with the selected external tool:

```
git difftool [-d] A B
```

or:

```
git difftool [-d] A..B
```

where `A` and `B` can be *commits*, *tags*, *branches*, *trees* or *blobs*.

Use the `-d` (or `--dir-diff`) option to perform a *directory diff* i.e. to examine all files with changes at the same time (otherwise files with changes are opened one by one in the diff tool).

You can also use:

```
git difftool [-d] A B FILENAME1 [FILENAME2 ...]
```

or:

```
git difftool [-d] A B -- FILENAME1 [FILENAME2 ...]
```

or:

```
git difftool [-d] A..B FILENAME1 [FILENAME2 ...]
```

or:

```
git difftool [-d] A..B -- FILENAME1 [FILENAME2 ...]
```

to get differences from A to B for specific files or directories `FILENAME1`, `FILENAME2`, ...

The `--` notation is only required the files you want to compare have contentious name (like "-f").

Also note that `FILENAME1`, `FILENAME2`, ... have to be paths relative to the current working directory. Paths relative to the root of the repository won't work.

To compare a git object A to `HEAD`, simply use:

```
git difftool [-d] A
```

You can make changes and save them from your external diff tool but only changes concerning the current working directory (`HEAD`) will be actually saved.

Examples:

```
git difftool -d ff20b ea76d ./src/main.c
git difftool -d ff20b ea76d
git difftool -d v1 v2
git difftool -d v1 v2 ./src/
git difftool -d master expermiental
git difftool -d master expermiental ./src/main.c ./Makefile
git difftool -d upstream/master bugfix ./src/main.c
```

See section Show changes between two git objects for more usage examples specific to *commits*, *tags* and *branches*: simply replace `git diff` by `git difftool` or `git difftool -d`.

See also https://git-scm.com/docs/git-difftool for more information.

5 Patches

5.1 Create a patch of current changes

```
git diff > FILE_NAME
```

Example:

```
git diff > foo.patch
```

5.2 Create a patch of changes introduced by a commit (without commit metadata)

```
git diff COMMIT_ID~ COMMIT_ID > FILE_NAME
```

or

```
git show COMMIT_ID > FILE_NAME
```

Example:

```
git diff ff6876ef~ ff6876ef > foo.patch
```

5.3 Create a patch of changes between 2 commits (without commit metadata)

```
git diff COMMIT_ID1 COMMIT_ID2 > FILE_NAME
```

Example:

```
git diff ff6876ef efe57652a > foo.patch
```

5.4 Apply a patch (without commit metadata)

```
git apply FILE_NAME
```

This command applies the patch but does not create a commit.

Example:

```
git apply foo.patch
```

5.5 Create a patch of changes introduced by a commit (with commit metadata)

```
git format-patch ... TODO
```

Example:

```
TODO
```

5.6 Apply a patch (with commit metadata)

```
git am ... TODO
```

This command applies the patch and make commits.

Example:

```
TODO
```

6 Remotes

6.1 Clone a remote repository

```
git clone REMOTE [LOCAL_DIRECTORY]
```

Example:

```
git clone https://github.com/jdhp-docs/git-memento.git
```

Only the `master` branch is available in the local repository. To get other branches from the remote repository see get_remote_branch.

7 Tags

7.1 Create a lightweight tag

```
git tag TAG_NAME [SHA1]
```

Example:

```
git tag v1.4
```

7.2 Create an annotated tag

```
git tag -a TAG_NAME -m "MESSAGE" [SHA1]
```

where `-a` means *annotated*.

Example:

```
git tag -a v1.4 -m "My version 1.4"
```

7.3 List local tags

```
git tag
```

or

```
git tag -l
```

7.4 List remote tags

```
git ls-remote --tags REMOTE
```

Example:

```
git ls-remote --tags origin
```

7.5 Push an annotated tag to a remote repository

```
git push REMOTE TAG_NAME
```

Example:

```
git push origin v1.4
```

7.6 Push all tags to a remote repository

```
git push --tags
```

or

```
git push --follow-tags
```

The latter is safer but anyway, it's generally considered a bad practice to push *all* tags automatically with these two commands. Be sure you're not pushing unwanded tags.

8 Branches

8.1 Create a local branch

```
git branch BRANCH_NAME
```

Example:

```
git branch experimental
```

8.2 Change the current branch

```
git checkout BRANCH_NAME
```

Example:

```
git checkout experimental
```

8.3 Create a local branch and switch to it

```
git checkout -b BRANCH_NAME
```

This is shorthand for:

```
git branch BRANCH_NAME
git checkout BRANCH_NAME
```

Example:

```
git checkout -b experimental
```

8.4 List local branches

```
git branch
```

The current branch is the starred one.

8.5 Rename a local branch

To rename any local branch:

```
git branch -m OLD_NAME NEW_NAME
```

Example:

```
git branch -m experimental testing
```

To rename the current branch:

```
git branch -m NEW_NAME
```

Example:

```
git branch testing
```

8.6 Delete a local branch

For branches merged with the current branch:

```
git branch -d BRANCH_NAME
```

Example:

```
git branch -d experimental
```

For branches not merged with the current branch (dangerous):

```
git branch -D BRANCH_NAME
```

Example:

```
git branch -D experimental
```

8.7 List remote branches

```
git branch -a REMOTE
```

Example:

```
git branch -a origin
```

8.8 Rename a remote branch

TODO

8.9 Delete a remote branch

For branches merged with the current branch:

```
git push REMOTE --delete BRANCH_NAME
```

or:

```
git push REMOTE :<BRANCH_NAME>
```

Example:

```
git push origin --delete experimental
```

8.10 Get a graphical representation of all branches (local + remote)

Get a graphical representation of all branches (local and remote):

```
git log --oneline --decorate --graph --all
```

or:

```
gitk --all
```

8.11 Push a local branche to a remote repository

```
git checkout LOCAL_BRANCH_NAME
git push REMOTE REMOTE_BRANCH_NAME
```

Example:

```
git checkout experimental
git push origin experimental
```

To automatically set `REMOTE REMOTE_BRANCH_NAME` as *upstream* for the current local branch (check the difference with `git branch -vva`):

```
git checkout LOCAL_BRANCH_NAME
git push -u REMOTE REMOTE_BRANCH_NAME
```

Once upstream is set for the current local branch, there is no need to specify `REMOTE_BRANCH_NAME` for a `git push`/`git push`:

```
git push REMOTE
```

Example:

```
git checkout experimental
git push -u origin experimental
...
git push origin
```

8.12 Get a given branche from a cloned remote repository

```
git checkout -b LOCAL_BRANCH_NAME REMOTE/REMOTE_BRANCH_NAME
```

Example:

```
git checkout -b experimental origin/experimental
```

8.13 Remove the upstream information for a given branch

To remove the upstream information for `LOCAL_BRANCH_NAME` (i.e. the default remote to use with `git pull` and `git push`):

```
git branch --unset-upstream LOCAL_BRANCH_NAME
```

If no branch is specified it defaults to the current branch.

Check the result with:

```
git branch -vv
```

9 Merge

9.1 Some definitions

Merge commit

A commit having more than one parent.

Octopus merge

A merge where more than two branches are involved (rarely used in practice).

9.2 When does a merge occurs ?

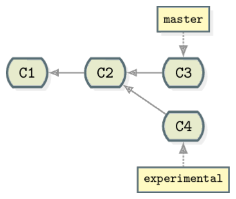
Either:

- explicitly with the `git merge` command;
- or implicitly with the `git pull` command.

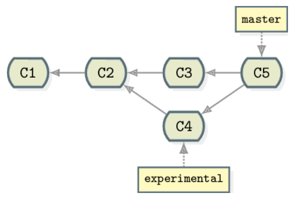
9.3 Merge a given local branch in the current branch

```
git merge LOCAL_BRANCH_NAME
```

Example:



```
git checkout master
git merge experimental
```



9.4 Merge a given remote branch in the current branch

```
git merge REMOTE/REMOTE_BRANCH_NAME
```

Example:

```
git merge upstream/master
```

9.5 Merge a given remote tag in the current branch

```
git fetch REMOTE
git merge TAG_NAME
```

Example:

```
git fetch upstream
git merge v0.1
```

9.6 Cancel an uncommited merge

To reset the working tree to the state it was before an uncommitted merge (e.g. when there are conflicts):

```
git merge --abort
```

9.7 Show conflicts (if there are any)

Get the list of files with unresolved conflicts after a `git merge` or `git pull`:

```
git status
```

Get the details of unresolved conflicts:

```
git diff
```

9.8 Solve conflicts manually

Get the list of files with unresolved conflicts after a `git merge` or `git pull`:

```
git status
```

Edit these files to solve conflicts.

Once you have solve conflicts, stage edited files:

```
git add FILE_NAME1 [FILE_NAME2 ...]
```

Staged files are considered resolved.

Then check that all conflicts are solved:

```
git status
```

Finally make the merge commit:

```
git commit
```

It's recommended to keep the default commit message.

9.9 Solve conflicts with graphical tools

Git support many *external diff tools* out of the box to resolve merge conflicts, including *opendiff*, *kdiff3*, *tkdiff*, *xxdiff*, *meld*, *kompare*, *gvimdiff*, *diffuse*, *diffmerge*, *ecmerge*, *p4merge*, *araxis*, *bc*, *codecompare*, *vimdiff* and *emerge* (the default one is *opendiff*).

To select which tool to use:

```
git config [--global] merge.tool TOOL_NAME
```

For instance:

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

To resolve conflicts with the selected graphical tool after a `git merge` or `git pull`:

```
git mergetool [FILE_NAME1, ...]
```

Specifying a directory will include all unresolved files in that path. If no `FILE_NAME` is specified, `git mergetool` will run the merge tool program on every file with merge conflicts.

If you use `meld` as `merge.tool` (probably the most popular mergetool), then update and save the **middle pane** only (the one called either *BASE* or *MERGED*). With `meld`, the left pane (named *LOCAL*) shows the contents of the file on the current branch (e.g. `master` in section Merge a given local branch in the current branch) and the right pane (named *REMOTE*) show the contents of the file on the branch being merged (e.g. `experimental` in section Merge a given local branch in the current branch). See this page for more information.

If, while you are editing the merge conflicts in your selected mergetool, you wish to cancel changes, then quit your graphical tool without saving anything.

A `.orig` file is created for each edited file. These are safe to remove once a file has been merged. Setting the `mergetool.keepBackup` configuration variable to `false` causes `git mergetool` to automatically remove the backup as files are successfully merged.

Once you have solve conflicts, edited files are automatically staged. Check that all conflicts are solved with:

```
git status
```

Check the differences with the former "*LOCAL*" branch:

```
git diff --cached
```

Finally make the merge commit:

```
git commit
```

It's recommended to keep the default commit message.

Good to know: `git mergetool` has no equivalant option to `git difftool --dif-diff` (i.e. it cannot open all conflicted files simultaneously and perform a *directory diff*).

See https://git-scm.com/docs/git-mergetool for more information.

9.10 Abort a merge for some files only

Typing:

```
git merge --abort
```

in the middle of a merge conflict resolution would reset all files. To reset only one given file, use this command instead:

```
git checkout -m FILENAME
```

If `FILENAME` has been migrated into the index by error, then it can be solved again with:

```
git mergetool FILE_NAME
```

See: http://stackoverflow.com/questions/6857082/redo-merge-of-just-a-single-file

9.11 TODO...

```
git merge -s ours BRANCH_NAME
```

Example:

```
git merge -s ours experimental
```

9.12 TODO...

```
git merge -s recursive -X theirs BRANCH_NAME
```

Example:

```
git merge -s recursive -X theirs experimental
```

10 Submodules

10.1 Few definitions

A submodule allows you to keep another Git repository in a subdirectory of your repository. The other repository has its own history, which stays completely independent and does not interfere with the history of the current repository. This can be used to have external dependencies such as third party libraries.

A repository that contains *subprojects* (called *submodules* in the git terminology) is here called *superproject*.

See https://git-scm.com/book/en/v2/Git-Tools-Submodules for more explanations about *submodules*.

10.2 Add a *submodule* to a *superproject*

10.2.1 Short version

From *superproject* (i.e. from the repository where you want to insert the *submodule*):

```
git submodule add SUBMODULE_REPOSITORY_URL [SUBMODULE_PATH]
git commit -m "Add a submodule"
```

where the optional argument `SUBMODULE_PATH` is the relative location for the cloned submodule to exist in the *superproject*. `SUBMODULE_PATH` is also used as the submodule's logical name in its configuration entries (unless --name is used to specify a logical name). For GitHub users, `SUBMODULE_REPOSITORY_URL` should be an HTTPS URL (i.e. not an SSH one) to grant everyone access to *submodule* within *superproject*.

10.2.2 More explanations

The `git submodule add` command creates the following objects in the working tree (and the git index):

```
.gitmodules
SUBMODULE_PATH
```

Where:

- `.gitmodules` (see gitmodules(5)) is a text file that assigns a logical name to the submodules and describes the default URL the submodules shall be cloned from;
- `SUBMODULE_PATH` has a special mode `160000` (see `git diff`, `git ls-tree HEAD` or `git cat-file -p master^{tree}` outputs) that indicates it's not a *tree object* but a *git link* that refer to a commit in the *submodule* repository. The content pointed by this link is not tracked in *superproject* repository.

The `git commit` command is used to add these two files to the *superproject* repository. Indeed they should be version-controlled (i.e. pushed and pulled with the rest of your project) so that users who clone the *superproject* can easily fetch the *submodule* (see the next subsection).

The `git submodule add` command doesn't write anything into the `.git/` repository, it only create the two objects previously described.

10.3 Clone a repository containing submodules (additional steps after cloning)

10.3.1 Short version

When cloning or pulling a repository containing submodules, these will not be checked out by default (i.e. the *submodule* directory is there but empty). The `init` and `update` subcommands are required to maintain submodules checked out and at appropriate revision in your working tree:

```
git submodule init
git submodule update
```

There is simpler way to clone a project and init/update all its submodules:

```
git clone --recursive SUPERPROJECT_REPOSITORY_URL
```

The `--recursive` option in `git clone` command automatically initializes and updates each submodule.

10.3.2 More details

The `git submodule init` command only update the `.git/config` file (adding a `[submodule]` entry).

The `git submodule update` actually fill the *submodule* directory with the content attached to the *submodule* commit pointed by the current *gitlink* in *superproject*.

10.4 Update a given submodule in a *superproject*

When a *submodule* repository is updated, the *superproject* doesn't automatically follow these changes ; the reason is you may want to stick to a particular approved version of the *submodule*. Thus updates have to be made explicitly.

From the *superproject*:

```
cd SUBMODULE_DIRECTORY
git fetch
git merge origin/master
```

or simply:

```
git submodule update --remote SUBMODULE_DIRECTORY
```

10.5 Update all submodules in a *superproject*

From the *superproject*:

```
git submodule update --remote
```

11 Altering commits

11.1 Warning

Do not alter commits that have been shared with other users (e.g. pushed, pulled or cloned) !

11.2 Amending the most recent commit message

```
git commit --amend -m "New commit message"
```

11.3 Revert *HEAD* to a known commit

```
git reset --soft COMMIT_ID
```

11.4 Revert *HEAD* and *index* to a known commit

```
git reset --mixed COMMIT_ID
```

or simply:

```
git reset COMMIT_ID
```

11.5 Revert *HEAD*, *index* and the *working directory* to a known commit

```
git reset --hard COMMIT_ID
```

11.6 Rebase

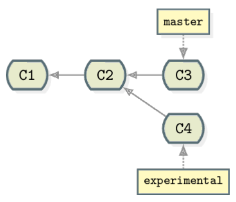
Take all the changes that were committed on one branch and replay them on another one:

```
git checkout SOURCE_BRANCH_NAME
git rebase DESTINATION_BRANCH_NAME
```

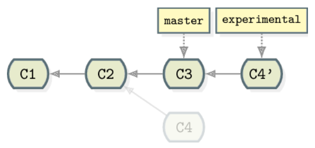
Then you can go back to the `DESTINATION_BRANCH_NAME` branch and do a fast-forward merge:

```
git checkout DESTINATION_BRANCH_NAME
git merge SOURCE_BRANCH_NAME
```

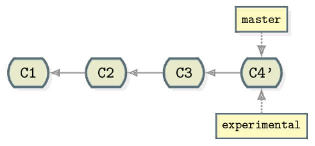
Example:



```
git checkout experimental
git rebase master
```



```
git checkout master
git merge experimental
```



12 Plumbing commands

12.1 Show the content of a commit

```
git show COMMIT_ID
```

12.2 Show the content of a file in the current branch

```
git show :FILE_PATH
```

Examples:

```
git show :README.rst
git show :content/introduction.rst
```

12.3 Show the content of a file in a given branch

Show the content of the FILE_PATH file in the BRANCH_NAME branch:

```
git show BRANCH_NAME:FILE_PATH
```

Examples:

```
git show experimental:README.rst
git show experimental:content/introduction.rst
```

12.4 Show the content of a file in a given branch of a given remote

Show the content of the FILE_PATH file in the BRANCH_NAME branch of REMOTE_NAME remote:

```
git show REMOTE_NAME/BRANCH_NAME:FILE_PATH
```

Examples:

```
git show origin/experimental:README.rst
git show origin/experimental:content/introduction.rst
```

12.5 Extract the content of a file in a given branch of a given remote

```
git show REMOTE_NAME/BRANCH_NAME:FILE_PATH > OUTPUT_FILE
```

Examples:

```
git show origin/experimental:README.rst > /tmp/README.rst
git show origin/experimental:content/introduction.rst > introcution.old.rst
```

12.6 Show the content of a file at a time point of the current branch

```
git show COMMIT_ID:FILE_PATH
```

Examples:

```
git show HEAD^:README.rst
git show HEAD^:content/introduction.rst
```

12.7 Extract the content of a file at a time point of the current branch

```
git show COMMIT_ID:FILE_PATH > OUTPUT_FILE
```

Examples:

```
git show HEAD^:README.rst > /tmp/README.rst
git show HEAD^:content/introduction.rst > introcution.old.rst
```

12.8 Show the type (blob, tree, commit or tag) of a git object

```
git cat-file -t OBJECT_ID
```

Examples:

```
git cat-file -t 33f4ea63
git cat-file -t HEAD
```

13 GitHub collaboration workflow

13.1 Roles

Source: https://guides.github.com/activities/contributing-to-open-source/

Owner

The user or organization that created the project.

Maintainers* and *Collaborators

The users primarily doing the work on a project and driving the direction. Oftentimes the owner and the maintainer are the same. They have write access to the repository.

Contributors

Everyone who has had a pull request merged into a project.

Community Members

The users who often use and care deeply about the project and are active in discussions for features and pull requests.

13.2 Workflow models

Shared Repository Model

When contributors have write access to the project (i.e. they are either *maintainers* or *collaborators*). Then, they can directly push their commits to the project repository.

Fork & Pull Model

When contributors don't have write access to the project (i.e. they are neither *maintainers* or *collaborators*). To contribute, they have to *fork* the project, clone this fork and then *suggest* their patches using GitHub *Pull requests*. Contributors who have write access to the project can also use the *Fork & Pull Model* to be sure the rest of the team agrees with the changes.

See also https://guides.github.com/introduction/flow/

13.3 The Fork & Pull model

13.3.1 Prior step: Fork the upstream repository and clone the fork

Fork the upstream repository on the GitHub website: see https://help.github.com/articles/fork-a-repo/#fork-an-example-repository.

Clone the fork repository (where *git@github.com:YOUR_USER_NAME/REPOSITORY_NAME.git* is the URL of *your* fork):

```
git clone git@github.com:YOUR_USER_NAME/REPOSITORY_NAME.git
cd REPOSITORY_NAME
```

Define the *upstream* remote repository (where *https://github.com/UPSTREAM_USER_NAME/REPOSITORY_NAME.git* is the URL of the *original* repository):

```
git remote add upstream https://github.com/UPSTREAM_USER_NAME/REPOSITORY_NAME.git
```

13.3.2 Step 1: Synchronize with upstream

Synchronize your work directory with upstream:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Synchronize your fork with upstream:

```
git push origin master
```

or simply:

```
git push
```

13.3.3 Step 2: Make your updates on a new branch

Create a new branch:

```
git checkout master
git checkout -b experimental
```

Update files...

Push your new local branch on your fork:

```
git push -u origin experimental
```

13.3.4 Step 3: Make a pull request

Make a pull request for your changes on the upstream repository on the GitHub website.

See https://help.github.com/articles/creating-a-pull-request/

GitHub uses some special words to describe the from and to branches:

- The *base* branch is where you think changes should be applied.
- The *head* branch is what you would like to be applied.

On the pull request creation page, *base* should be set to "master" (we want to merge our changes to upsteam/master) and *head* should be the name of our working branch ("experimental" here).

13.3.5 Step 4: (If requested) correct mistakes and re-push to branch

If *maintainers* or *collaborators* request some correction on your updates prior to validate them, here is the procedure to follow.

Be sure you are still in your updates dedicated branch:

```
git checkout experimental
```

Then correct mistakes in your changes then commit them and push them to your fork:

```
git commit -m 'YOUR MESSAGE'
git push origin experimental
```

or simply:

```
git commit -m 'YOUR MESSAGE'
git push
```

Then your new changes are automatically signaled to the *maintainers* or *collaborators* and your pull request page on github is automatically updated to reflect your last changes.

13.3.6 Step 5: Merge pull request

Once your pull request has been accepted, your update dedicated branch has been added and merged to master in the upstream repository.

TODO: add an image

13.3.7 Step 6: Synchronize with upstream

For the last step, you need to synchronize your working directory and your fork repository with upstream.

Synchronize your work directory with upstream:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Synchronize your fork with upstream:

```
git push origin master
```

or simply:

```
git push
```

TODO: add an image

13.3.8 Step 7: (Optional) remove the working branches

Remove the remote working branch of your fork on Github:

```
git push origin --delete experimental
```

or simply delete the branch from the github interface.

Then remove the local working branch:

```
git branch -d experimental
```

14 Work on Subversion repositories with Git

...

15 Work on Mediawiki with Git

...

16 License

This document is provided under the terms and conditions of the Creative Commons 4.0 (CC BY-SA 4.0) license.