Git Memo Documentation

Release v1.1

Marc Zonzon

April 12, 2015

Contents

Contents:

CHAPTER 1

Introduction

License

This is a collection of notes related to git use. These notes have been taken mainly for my own usage, to help me to understand some feature or to remember the way of doing common operations.

There are many good tutorials, but this memo is not a tutorial. Most of the key points you have to learn first are skipped, some points are very easy and common, some are exploring git use more in depth.

This is not either a reference, it's author is not a git expert.

But If ever you find here something useful for your daily use of git, I am happy to share these notes with you.

Beside this memo I have also put pointers to git documentation in the Git section and subsections of my mzlinux site.

CHAPTER 2

Repository Info

Ref: **git tutorial** Git User Manual

2.1 Status

Refs: git-status(1)

Status of the repo: files modified from index, index that differ from the HEAD:

```
git status
```

Don't show untracked files:

```
git status -u no
```

Status of the current directory:

```
git status -- .
```

2.2 Branches

Refs: git-branch(1),

Local branches:

```
$ git branch
```

Remotes branches:

```
$ git branch -r
$ git branch -a # both local and remotes
```

Show also what are the tracking branches, if any.:

```
$ git branch -vv
```

2.3 Tags

Refs: git-tag(1),

To know your local tags:

```
$ gi tag
v1
v2
```

If you want to know the annotation that *may* come with the tag you have to use the option `--list <pattern>` abr. `-l<pattern>` wich list the tages that match a shell pattern, and `n<num>` that allow *n* lines of annotation.

To get all the tags with annotations:

```
$ git tag -n100 -l \*
```

See also *Sharing Tags*

2.4 Describe

Refs: git-describe

`git-describe` show the the most recent tag that is reachable from a commit. By default it uses only annotated tags, but you can use any tag with `--tags` option.

Exemple:

```
$ git tag
v0.2
v0.90
$ git describe
v0.90-3-g8a8e4de
```

You are 3 commits after the version v0.90 at the commit `8a8e4de`. The prefix `g` is added to indicate a git managed version.

2.5 Logs

Refs: git-log(1), git-show(1), git-diff(1), gitrevisions(7).

git tutorial: Exploring history.

Git User Manual: Browsing revisions, Understanding Commits,

Review changes in the whole repository.

```
$ git log --name-status
$ git log --summary
$ git log --stat
$ git log --patch # abbrev -p
```

Changes on some file/directory

```
$ git log  --stat   -- Muttrc
$ gitk -- Muttrc
$ gitk --all -- Muttrc
```

All the commits which add or remove any file data matching the string `'foo()'`:

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

To follow among renames, even crossing directories, use for a single file:

```
$ git log -p --follow Muttrc
```

Changes in a commit range:

```
$ git log v2.6.15..v2.6.16  # ...in v2.6.16, not in v2.6.15
$ git log master..test      # ...in branch test, not in branch master
$ git log test..master      # ...in branch master, but not in test
$ git log test...master     # ...in one branch, not in both
$ git log --since="2 weeks ago"
```

Changes introduced by the last commit:

```
$ git log -1 --stat
$ git log -1 -p
```

Changes introduced by some commit: You need only the initial part of the commit sha.

```
$ git log -1 --stat 20b0f6e1961d5da
$ git log -1 --stat -p  20b0f6e1961d5da
$ git show 20b0f6e1961d5da
$ git show HEAD
$ git show devel # the tip of the "devel" branch or tag
$ git show HEAD^  # to see the parent of HEAD
$ git show HEAD~4 # 4 commits before HEAD
```

If the commit is a merge commit `git show <commit>` give only the difference between <commit> and its first parent. To get both:

```
$ git show <commit>^1
$ git show <commit>^2
```

You can also use git-diff but by suffixing the commit with `^!` to mean the commit and nothing in the ancestors (see gitrevisions)

```
$ git diff 20b0f6e1961d5da^!
$ git diff HEAD^!
```

2.6 Reflogs

Refs: git-reflog(1), git-log(1), git-show(1), user-manual: recovering lost changes, git-notes: reflog

The reflog records each position of HEAD in the last 30 days (or configuration `gc.reflogExpireUnreachable`). The reflog history is local to your repository not shared, or cloned.

To show the reflog use:

```
$ git reflog show --date=relative
$ git log --walk-reflogs
$ git show-branch --reflog
```

`--walk-reflogs` and `--reflog` are abridged in `-g`. If the rebase and amend don't appear in a simple log without `-g`, when you use the reflog you can see and recover commits that have been amended or let away by a rebase.

You can attain any past commit not yet pruned by:

```
$ git log master@{1}
$ git show HEAD@{"1 week ago"}
```

2.7 git diff

git diff show differences introduced by commits

Refs: git-diff(1), git-difftool(1), gitrevisions(7), git-format-patch(1).

Diff and index:

```
# Changes beetween the index and the working tree;
# i.e change in the working tree not yet staged for the next commit.
$ git diff
# Changes between  your last commit and the index;
# what you would be committing if you run "git commit" without "-a" option.
$ git diff --cached
# Changes beetween your last commit and the working tree;
# what you would be committing if you run "git commit -a"
$ git diff HEAD
```

diffs between two branches:

```
$ git diff master..test
```

You can also use a *difftool*, if you want to see the diff with *meld*:

```
$ git difftool --tool=meld  master..test
```

To know the list of available tools:

```
$ git difftool --tool-help
```

To define a new tool you set in your `.gitconfig`:

```
[difftool "ediff"]
    cmd = emacs --eval \"(ediff-files \\\"$LOCAL\\\" \\\"$REMOTE\\\")\"
```

You use a triple dot to get the diff between the common ancestor of *master* and *test* and the tip of *test*. *Warning: The semantic of the triple dot is different with* `git log`:

```
$ git diff master...test
```

Patch to apply to *master* to obtain *test*:

```
$ git format-patch master..test
```

2.8 Commit tree

Refs: gitk(1), tig-manual, git-log(1),

View source commit tree, you can use many GUIs, gitk is provided with git, and tig is a ncurses front-end. .

```
$ gitk --all
$ tig --all
$ gitg
```

You can also use git-log, with the option `--graph`:

```
$ git log --graph --pretty=oneline --abbrev-commit --decorate --all --color
```

2.9 Looking file content in the tree

Refs: git-grep(1), git-log(1)

```
$ git grep "foo()"                  # search working directory for "foo()"
$ git grep 'defun.*init *\(.*\)'    # search working directory for pattern
$ git grep -E 'defun.*init *(.*)'   # use extended regexp (default basic)
$ git grep "foo()" v2.6.15          # search old tree for "foo()"
$ git grep init 6874caeedb3c -- *.el # search "init" in .el files at some commit
```

To search "`foo()`" in all files in all commit history:

```
$ git rev-list --all | xargs git grep "foo()"
```

To look for the commits that **introduced** or **removed** "`foo()`":

```
$ git log -p -S "foo()"
```

To search for commit that **introduced** or **removed** an extended regex:

```
$ git log -p -S pickaxe-regex 'defun.*init *\(.*\)'
```

To search for commit whose patch text contains added/removed lines that match a regex:

```
$  git log -p -G 'defun.*init *\(.*\)'
```

`log -G` will show a commit that just **moved** the regexp, without changing its number of occurences, while `log -p -S pickaxe-regex` will not retain it.

2.10 Viewing other versions of a file

Refs: git-show(1),

You can use a tag, a branch, or a commit sha.

```
$ git show devel:src/prog.py
$ git show v2.5:src/prog.py
$ git show e05db0fd4f3:src/prog.py
```

CHAPTER 3

Developping

Ref: User-manual: Developing with git

3.1 Adding files to the index

To add file to the index do:

```
$ git add file1 file2 file3
```

To undo it:

```
$ git reset -- file1 file2 file3
```

the previous command do not work before the first commit because you have no *HEAD*, but you can remove the files from the index *while keeping it in the working tree* with:

```
git rm --cached file1 file2 file3
```

3.1.1 glob expansion

There is a difference in the way the patterns are interpreted by git:

```
$ git add *.txt
```

is a shell glob expansion it will add `file.txt` but not `subdir/other.txt` while:

```
$ git add \*.txt
```

is expanded by git and it will add both.

3.2 Adding a part of a file

Ref: git-add(1), git book: Interactive Staging

For a registered file, that has changed since last commit you can use `git add --interactive` or `-i` to select the patch to add to the index:

```
$ git add -i
  1:        +0/-0       +65/-0 source/developping.rst
  2:    unchanged        +1/-1 source/error_fix.rst

  *** Commands ***
  1: status   2: update       3: revert       4: add untracked
  5: patch    6: diff         7: quit         8: help
```

```
What now> p
........
Stage this hunk [y,n,q,a,d,/,e,?]? e
Waiting for Emacs...
....
What now> s
           staged     unstaged path
1:         +7/-0       +58/-0 source/developping.rst
2:    unchanged        +1/-1 source/error_fix.rst
.....
What now> q
Bye.
```

Here the patch has be selected by editing the patch in emacs.

You can also use tools like git-gui(1), magit, or one of the many guis availables to select the hunks you want to stage.

3.3 Adding a part of a new file

Now if the file is yet unregistered you cannot add only a part with `git add -i`.

But you can use the sequence:

```
$ git add -N new_file
$ git add -i
```

`git add -N` create an entry for *new-file* in the index with no content.

Then the interactive add is done like above.

3.4 Well formed commits

Summary of A Note About Git Commit Messages :

1. Short (50 chars or less) summary
2. Blank line.
3. More detailed explanatory text wrapped to about 72 cols.
4. Further paragraphs come after blank lines.
5. You can use bulleted text with a hyphen or asterisk preceded by a single space
6. Use the present tense.

3.5 Adding an empty commit at root of a branch.

Refs: git symbolic-ref, git checkout, git clean

```
$ #store inexistent ref: newroot in HEAD
$ git symbolic-ref HEAD refs/heads/newroot
$ # wipe the index
$ git rm --cached -r .
$ # clean the worktree
$ git clean -df
$ #create the branch newroot with an empty commit
$ git commit --allow-empty -m 'root commit'
```

```
$ # rebase everything over newroot
$ git rebase newroot master
```

- repeat for other branches you want to rebase on the same newroot
- You can then move to some branch and remove *newroot* with `git branch -d newroot`.

Recent git have the `--orphan` option to `checkout` to create a new branch starting from nowhere. You can also do:

```
$ git checkout --orphan newroot
$ clear the index and the working tree
$ git rm -rf .
$ git commit --allow-empty -m 'root commit'
$ git rebase newroot master
```

CHAPTER 4

Objects names

4.1 Naming commits

See git-rev-parse, gitrevisions(7), git-name-rev(1), git-describe(1), git-reflog(1)

- To inspect your repository (and in scripts)

```
$ pwd
/home/marc/bash/lib
$ git rev-parse  --is-inside-git-dir
false
$ git rev-parse  --is-inside-work-tree
true
$ git rev-parse --git-dir
/shared/home/marc/bash/.git
$ git rev-parse --show-cdup
../
$ git rev-parse --show-prefix
lib/
```

- translating symbolic <-> sha

```
$ git rev-parse --symbolic-full-name HEAD
refs/heads/master
$ git symbolic-ref HEAD
refs/heads/master
$ git name-rev --name-only HEAD
master
$ git rev-parse  HEAD~3
25f4b1d58e20f2026a36d80073654f52b055537b
$ git name-rev --name-only 25f4b1d58e20
master~3
```

- find the hash of some file
  - Look at Pro Git: Git Objects for details.
    - Discussion by Linus Torvald

```
git hash-object <file>
cat <file> | git hash-object --stdin
(/usr/bin/stat  --printf "blob %s\0" calculette.c; cat calculette.c) | sha1sum
```

- to see remotes

```
$ git remote
lighttpd
nx_server
ssh_server
```

```
....
$ git rev-parse  --symbolic --remotes
lighttpd/master
nx_server/distribution
nx_server/kernoel
nx_server/master
ssh_server/distribution
ssh_server/kernoel
ssh_server/master
ssh_server/tubuntu
....
```

- remote details

```
$ git remote show ssh_server
* remote ssh_server
  URL: ../ssh_server
  Tracked remote branches
  kernoel master tubuntu
  $ git config --get-regexp remote\\.ssh_server\\..\*
  remote.ssh_server.url ../ssh_server
  remote.ssh_server.fetch +refs/heads/*:refs/remotes/ssh_server/*
```

- version/most recent tag

```
$ git describe HEAD
init-1.0-29-gcb97cd9
$ git name-rev --name-only cb97cd9
master
$ git describe HEAD~14
init-1.0-15-g84aeca4
$ git name-rev --name-only 84aeca4
master~14
$ git describe HEAD~29
init-1.0
$ git describe --long HEAD~29
init-1.0-0-ge23c217
```

- past tips of branches

  We use the reflog, be careful that the reflog is local to your repository, and is pruned by `git reflog expire` or by `git gc` `HEAD@{25}` is the 25th older head of branch, this is not always the same than `HEAD~25` which is the 25th ancestor of the actual head.

```
$ git name-rev HEAD@{25}
HEAD@{25} b3distrib~11
$ git rev-parse HEAD@{25}
2518dd006de12f8357e9694bf51a27bbd5bb5c7a
$ git rev-parse HEAD~11
2518dd006de12f8357e9694bf51a27bbd5bb5c7a
$ git name-rev 2518dd0
2518dd0 b3distrib~11
$ git rev-parse HEAD@{18}
0c4c8c0ea9ab54b92a2a6d2fed51d19c50cd3d76
$ git name-rev HEAD@{18}
HEAD@{18} undefined
$ git rev-parse HEAD@{14}~4
0c4c8c0ea9ab54b92a2a6d2fed51d19c50cd3d76
$ git rev-parse HEAD@{13}~5
24c85381f6d7420366e7a5e305c544a44f34fb0f
git log -1 -g --oneline HEAD@{13}
a1b9b5c HEAD@{13}: checkout: moving from b3distrib to a1b9b5c
```

  In the previous example The 13th ancestor from the `HEAD` is a checkout at the beginning of a rebase so `HEAD@{14}` is now dangling, and `HEAD@{18}` the fourth predecessor (`HEAD@{14}~4`) of `HEAD@{14}`

is unreachable from a ref.

Nevertheless `HEAD@{25}` has been rebased as `HEAD~11` and can be reached.

4.2 Finding the sha of a file

Refs: git ls-files(1), git ls-tree(1), git-rev-parse(1), gitrevisions(7), git hash-object(1).

To show the blog sha associated with a file **in the index**:

```
$ git ls-files --stage somefile
100644 a8ca07da52ba219e2c76685b7e59b34da435a007 0   somefile
```

This is **not** the *sha1 sum* of the raw content, but you can get it from any file *even unknown in your repository* with:

```
$ git hash-object somefile
a8ca07da52ba219e2c76685b7e59b34da435a007
$ cat somefile | git hash-object --stdin
a8ca07da52ba219e2c76685b7e59b34da435a007
```

The sha is derived from the content, and the size of the file, you can get it from the sha1sum command with:

```
$ (/usr/bin/stat  --printf "blob %s\0" somefile; cat somefile) | \
  sha1sum
a8ca07da52ba219e2c76685b7e59b34da435a007
```

While git ls-file use by default the cached content, by using plumbing commands, you can also look at any object.

To show the blog sha of the object associated with a relative path in the *HEAD*:

```
$ git ls-tree HEAD <path>
```

You can also use path starting from the git worktree directory. If the root of your are in a directory *subdir* you get the same result with:

```
$ git ls-tree HEAD somefile
100644 blob 1a8bedab89a0689886cad63812fca9918d194a98        somefile
$ git ls-tree HEAD :somefile
100644 blob 1a8bedab89a0689886cad63812fca9918d194a98        somefile
$ git ls-tree HEAD :./somefile
100644 blob 1a8bedab89a0689886cad63812fca9918d194a98        somefile
git ls-tree HEAD :/subdir/file #note initial slash
100644 blob 1a8bedab89a0689886cad63812fca9918d194a98        somefile
```

you can also use git rev-parse with:

```
$ git rev-parse HEAD:subdir/somefile # no leading slash
1a8bedab89a0689886cad63812fca9918d194a98
$ git rev-parse HEAD:./somefile
1a8bedab89a0689886cad63812fca9918d194a98
$ git rev-parse :./somefile # index cached content
a8ca07da52ba219e2c76685b7e59b34da435a007
$ git rev-parse :0:./somefile
a8ca07da52ba219e2c76685b7e59b34da435a007
$ git hash-object somefile # the unregisterd worktree version
67a21c581328157099e8eac97b063cff2fb1a807  somefile
```

4.3 Finding the top level directory

Ref: git-rev-parse(1)

To show the absolute path of the top-level directory.:

```
$git rev-parse --show-toplevel
```

To show the *relative* path of the top-level repository:

```
$git rev-parse --show-cdup
```

or to show the path of the current directory relative to the top-level:

```
$git rev-parse --show-prefix
```

I use it to have a default message showing paths relative to top-level with:

```
$git commit :/$(git rev-parse --show-prefix)<relative-name>
```

To show the git directory:

```
$git rev-parse --git-dir
```

If `$GIT_DIR` is defined it is returned otherwise when we are in Git directory return the `.git` directory, if not exit with nonzero status after printing an error message.

To know if you are in a work-tree:

```
$git rev-parse --is-inside-work-tree
```

Note also that an alias expansion prefixed with an exclamation point will be executed from the top-level directory of a repository i.e. from `git rev-parse --show-toplevel`.

CHAPTER 5

Sharing development

Refs: git manual:Sharing development with others

5.1 Git Pull

Refs: git pull(1), git fetch(1)

To pull your master branch with the remote tracking branch in *origin*:

```
$ git pull origin master
```

It is equivalent t:

```
$ git fetch
$ git merge origin/master
```

5.2 Git pull –rebase

Refs: gitolite: git-pull –rebase

Instead of merging the remote repository after fetching, `git pull --rebase` rebase the current branch on top of the upstream branch. But it uses relog information to avoid to rebase commits previously pulled from the remote.

5.3 Git push

Refs: git push(1), git manual: Pushing changes

To push your master branch to the *origin* repo:

```
git push origin master
```

The origin should be ref:*configured as remote <remote_config>*.

5.4 Sharing Tags

5.4.1 Listing remote tags

Refs: git ls-remote(1), git fetch(1), git show-ref(1)

We have seen how to *List local tags*, but the remote repository can have a different set of tags. Usually we want to have the tags of a remote origin bare repository, but if we include also in our remotes the repo of a fellow developper it is usually inapropriate to import all his tags.

To list the remote tags we use:

```
$ git ls-remote --tags somerepo
da4412bf6edd0d99c8149a205d78b6a0a6f8f091    refs/tags/torepair
4a7f903017e22d0effb4b233f99548fd3abdac11    refs/tags/torepair^{}
17b3e9b93faf30e59fe9910de2da208d018bba7a    refs/tags/v1
4a7f903017e22d0effb4b233f99548fd3abdac11    refs/tags/v1^{}
```

Here the objects `da4412b` and `17b3e9b` are the tags object, and `4a7f9030` is the commit pointed to by the two tags. The notation `<rev>^{}` dereference the tag recursively until a non-tag object is found (see gitrevisions)

The lightweight tags are also shown by this command. But lightweight tags are not object, but only an alias for a commit, so only the commit appear in the list.

To differenciate between lightweight and anotated tags you can `git-cat-file -t <tag>` it output tag for a rag object, but commit for a lightweight tag.

Tags are fetched by default, unless you spectify `--no-tags` or have set the option `remote.<name>.tagopt`. If you don't change defaults you get the remote tags from the repository you fetch from; but they are not pushed by default, that allow to have tags for local use in your repo.

You can also use:

```
$  git ls-remote --tags .
```

to get the *local* tags.

It is equivallent to:

```
$ git show-ref --tags --dereference
```

5.4.2 Fetching remote tags

Refs: git fetch(1), git show(1).

To fetch an individual remote tag:

```
$ git fetch somerepo tags/torepair
```

Then you can examine it with *git tag*, or with:

```
$ git show torepair
tag torepair
Tagger: Some Body <some.body@git.org>
Date:   Sun Oct 19 11:45:13 2014 +0200

defective commit

commit 4a7f903017e22d0effb4b233f99548fd3abdac11
........
```

5.4.3 Pushing Tags to remote

Refs: git push(1).

To push an individual tag:

```
$ git push origin tags/v1
To git@github.com:me/testrepo.git
 * [new tag]         v1 -> v1
```

To push and include *all* tags:

```
$ git push --tags origin
Counting objects ...
...
[new tag]         v1 -> v1
```

5.4.4 Changing the tag message

Modifying a shared tag is strongly discouraged by git-tag(1). But changing only the message while keeping an unchanged date and commit reference is not too harmfull, but you have to know that your change will not be automatically propagated to people that pull from you.

If you want to keep the original date use:

```
GIT_COMMITTER_DATE="2014-09-28 11:52" git tag -a -f \
-m "new description" tag v0.90 v0.90
```

CHAPTER 6

Merge and Patch

6.1 Merging

- The two parents and their common base form the three stages of the merge:
  - `git show :1:file` is the `base` we have the difference as `git diff -1` or `git diff --base`
  - `git show :2:file` is `ours` we have the difference as `git diff -2` or `git diff --ours`
  - `git show :3:file` is `theirs` we have the difference as `git diff -3` or `git diff --theirs`
- In a failed merge an unmerged path *file* contains the combined unmerged file, and *git diff* will show a *combined diff* which show differences with the two parents.
- `git log --merge -p <path>` will show diffs first for the HEAD version and then the MERGE_HEAD version.
- `git log ..otherbranch` show the changes that will be merged in the current branch.
- `git diff ...otherbranch` is the diff from common ancestor (merge base) to graphical representation of the branches since they were merged last time.
- `git mergetool` launch a graphical mergetool which will work you through the merge.

6.1.1 Keeping one branch

If in your repo you have a conflict in a file, but want to adopt either *ours* or *theirs* version of the file, you don't need to manually edit the merge, you can checkout the chosen file:

```
$ git checkout --theirs <file>
$ git add <file>
```

Refs:

- git doc: git-merge.html

6.2 Submitting patches

Use git format-patch

```
$ git format-patch origin
```

Or with a commit range:

```
$ git format-patch before..end
```

Produces in the current directory a sequence of patches, with names from each commit log.

You can apply them one by one or as a whole with:

```
$ git am patches.mbox
```

git-am will fail if the patch does not apply, you can instead use a 3 ways merge with:

```
$ git am -3 patches.mbox
```

You can fix conflicts, add the fixed files to the index and commit with:

```
$ git am --resolved
```

Refs:

- Applying / Merging Changes From One Git Repository To Another
- git – applying patches
- Pro Git: Maintaining a Project
- gitready: pick out individual commits compare `git cherry-pick`, `git format-patch` with `git am`, or with `git apply`, and `git merge`.

6.3 Subtree-merge

Subtree merge is a very useful strategy to import a subtree from an other repository in a branch of our repository.

It is presented in a very concise way in Git howto: How to use the subtree merge strategy from which I extract the following code that illustrates the merging of a project **B** in the subdirectory `dir-B` of our project.

```
$ git remote add -f Bproject /path/to/B
$ git merge -s ours --no-commit Bproject/master
$ git read-tree --prefix=dir-B/ -u Bproject/master
$ git commit -m "Merge B project as our subdirectory"
```

To follow the changes in the B project you use:

```
$ git pull -s subtree Bproject master
```

This strategy ia applied to a bigger example in GitHub Help: Working with subtree merge

A different technique, doing the merge after the read-tree is in Scott Chacon Pro Git: Subtree Merging and used in this Git Subtree Workflow by David S Anderson and in Git Subtree Merge –The Quick Version by John Atten.

CHAPTER 7

Remote repositories

7.1 Cloning a remote repository

Ref: git-clone(1).

To clone a remote repository:

```
$ git clone git://github.com/cramz/git-memo.git
```

This create a local `git-memo` repository with a copy of the remote and check-out the current branch.

All the remotes branches are copied in the local repository, and the local branch are ref:*tracking the remote ones <remote_tracking>*

If you add the `--mirror` option you create a *bare* repository and maps all remote refs in the local repository.

7.2 Adding a new remote

In my local `git-memo` repository:

```
git remote add myserver git://myserver.com/cramz/git-memo.git
```

add a new remote called `myserver`, this does not fetch or push, or checkout anything, but the next:

```
git fetch
```

Will fetch `refs/remotes/myserver/master`. This remote tracking branch is updated also by:

```
git remote update myserver
```

By default all the remote branches are tracked, but you can use `-track <branch>` (abbrev `-t`) to select only some branches.

7.3 Fetching from Remote

To fetch all remotes:

```
$ git remote update
```

7.4 Remote configuration

To get remotes configuration we can edit .git/config or

```
$ git config --get-regexp 'remote'
remote.etc.url /shared/home/marc/crypt/gits/etc_shared.git
remote.etc.fetch +refs/remotes/etc/*:refs/remotes/etc/*
remote.etc.tagopt --no-tags
```

Options are added with commands like:

```
$ git config --add remote.etc.tagopt --no-tags
```

7.5 creating a bare remote repository

1. copy local bare repo

   ```
   $ git clone --bare ~/proj proj.git
   $ scp proj.git myserver.com/var/git/proj.git
   $ cd proj
   $ git remote add origin ssh://myserver.com/var/git/proj.git
   ```

2. push a branch to an empty repository

   We can also create it empty and push on the remote server

   - on the remote server:

     ```
     $ mkdir /var/git/proj.git && cd /var/git/proj.git
     $ git --bare init
     ```

   - on the client:

     ```
     $ cd proj
     $ git remote add origin ssh://myserver.com/var/git/proj.git
     $ git push origin master
     ```

7.6 push and delete remote branches

To push a new branch:

```
$ git push origin newfeature
```

To delete the branch on the remote:

```
$ git push origin :newfeature
```

It means push an empty branch to newfeature

7.7 References

- github: remotes
- git-remote(1)
- Setting up a new remote git repository

CHAPTER 8

Remote tracking branches

8.1 Tracking or not tracking

When you *clone a remote repository* all the remote branches are tracked and set as *upstream branch* for the new checked out master, then git-pull(1) will appropriately merge from the starting point branch.

But it is not special to cloning, when a local branch is started off a remote-tracking branch, the remote branch is tracked, with the default value of the global `branch.autosetupmerge` configuration flag.

If you want to override this global setting, you can use the option `--track` or `--no-track`.

To start a local branch from *origin/mywork* but not track the origin, you issue:

```
git branch --no-track origin/mywork
```

Note that for two local branches the default, is no tracking, so with:

```
git branch develop master
```

or:

```
git checkout -b develop master
```

develop will not track master, unless you have used:

```
git branch --track  develop master
```

or:

```
git checkout -b --track  develop master
```

You can add a tracking of an upstream branch with:

```
git branch --set-upstream-to=origin/mywork mywork
```

This is specially usefull whan you first created *mywork* and then pushed it to *origin* as:

```
git push origin mywork
```

will not set *origin/mywork* as remote tracking branch for *mywork*, except if you explicitly issue:

```
git push --set-upstream origin mywork
```

or have set `branch.autosetupmerge` to `always`.

`--set-upstream` is abridged in `-u`.

8.2 Configuration of upstream branches

A branch is registered as *upstream* for another one by setting the two configuration variables `branch.<name>.remote` and `branch.<name>.merge`.

The previous tracking branch will result in a configuration including:

```
[branch "mywork"]
remote = origin
merge = refs/heads/mywork

[remote "origin"]
url = <url>
fetch = +refs/heads/*:refs/remotes/origin/*
```

see the documentation of these two configuration options in git-config(1) to learn the configuration setting a local branch as upstream for an other local branch.

CHAPTER 9

Rebase

Reference: git-rebase(1)

The introduction is a condensed presentation of examples in git-rebase(1)

9.1 Rebase a topic

Suppose you have this situation:

```
     A---B---C---D---E topic
    /
D---E---A---F master
```

And you are on `topic`.

Then the command:

```
git rebase master
```

will apply on `master` all commit in `topic` that where not yet there, *A will be skipped,* and the result is:

```
             B'--C'---D'---E' topic
            /
D---E---A---F master
```

If you where not on `topic` but on `master` then you should have used instead:

```
git rebase master topic
```

That will do the successive operations:

```
git checkout topic
git rebase master
```

If you want only to use the commits starting with `D` you use:

```
git rebase --onto master C topic
```

to get:

```
             D'---E' topic
            /
D---E---A---F master
```

In all these example the rebase let you on `topic`.

9.2 Interactive Rebase

Reference: git-rebase interactive mode

The `--interactive` switch is used to reorder or merge patches.

```
$ git rebase --interactive HEAD~8
pick f08daa2 modified:   moin/farmconfig.py configured for moinmoin notebook and marcwiki
pick 802071d moin/notebook.py: added config for notebook
pick 65802dc moin/farmconfig.py added  mail_smarthost
pick ee35e7d changed fstab and hosts
pick 9913667 /etc/fstab: fixed cifs and nfs shares
pick 54055e3 fstab: crypt cannot be fscked at boot, disabled fsck
pick 1470a45 fstab: changed mountpoint
pick afbb0b8 passwd group mailcap state of etc/kernoel/master

# Rebase 15b369f..afbb0b8 onto 15b369f
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit,
#              and edit the sum of commit messages
# If you remove a line here THAT COMMIT WILL BE LOST.

The option are:
```

delete: if you delete the commit line, it will be omitted from the *rebase*.

reorder: You can change the orders of the commit *pick* lines, they will be processed in the new order.

pick, or reword (shortcuts `c` and `r`): Include the commit silently, *reword* is similar to *pick*, but *rebase* will open the commit message in an editor to allow you to fix it.

edit (shortcut `e`): For each *edit* the commit is applied, then the *rebase* pause to allow you to use `git commit --amend` to change the commit message, or change the commit, or *split it in many smaller commits*.

squash and fixup (shortcuts `s` and `f`): *squash* merge the commit in the previous one, then the *rebase* pause to let you edit the merged commits. If you instead use *fixup*, the second commit message is discarded and the first one is used.

exec (shortcut `x`): *exec* command launches the command in a shell spawn from the root of the working tree. The rebase will continue if the shell exit with a 0 status, and pause when the command fail, to let you fix teh error and `git rebase --continue` or `git rebase --abort`.

reorder + *squash* + *delete* is a very powerful tool to clean a suite of patches.

For each *edit*, *squash*, failed *exec* or conflict *rebase* will stop until you edit or merge comments (in case of a squash), or fix the conflict, then you just need to:

```
$ git rebase --continue
```

or:

```
$ git rebase --abort
```

9.2.1 *Interactive rebase* example

You have made a small error in the file SmtplibExample.py, and corrected it, You don't want to make a new commit for this tiny fix, but make it part of your previous commit.

You stash your current work

```
$ git stash --quiet
```

You look at the last commit for the file

```
$ git log -n 1 --pretty=oneline --abbrev-commit  TD/Smtp/SmtplibExample.py
9c091e6 SmtplibExample.py: refactored to a function and a main.
```

You rebase from the previous commit:

```
$ git rebase --interactive 9c091e6^
```

You get the rebase list to edit:

```
pick 9c091e6 SmtplibExample.py: refactored to a function and a main.
pick 3d3f53e SmtplibExample2.py: 2to3, switched to argparse, minor fixes
pick 0c4f2cf Cours/SocketTcp.mdn: sockets lectures now in markdown
pick aa34250 index.mdn: added sockets
....
```

You change the first *pick* to *edit* valid it, then rebase pause at:

```
Stopped at 9c091e6... SmtplibExample.py: refactored to a function and a main.
You can amend the commit now, with
git commit --amend
Once you are satisfied with your changes, run
git rebase --continue
```

You checkout your amended file from the stash:

```
git checkout stash@{0} -- : TD/Smtp/SmtplibExample.py
```

You add it and amend the commit:

```
git add TD/Smtp/SmtplibExample.py
git commit --amend
```

You continue the rebase:

```
[detached HEAD eae8d29] SmtplibExample.py: refactored to a function and a main.
1 files changed, 22 insertions(+), 14 deletions(-)
Successfully rebased and updated refs/heads/master.
```

See also the Interactive rebase help at github

9.3 Checking your rebase and undoing it

The rebase can be a dangerous operation, sometime I lost a file by deleting a commit that add a file within an interactive rebase. The head *before* a rebase is stored in ORIG_HEAD. All dangerous operations like *rebase*, *merge*, *pull*, *am* modify this reference, so you can only use it to refer to the HEAD *before* the last dangerous operation (but a simple commit don't change it).

To see what you have changed in the repository since last dangerous operation:

```
git diff ORIG_HEAD HEAD
```

If it was an interactive rebase to clean your history you expect that you preserved the global state of your repository, and to have an empty answer.

To see what commits are in HEAD and not in ORIG_HEAD:

```
git log ORIG_HEAD..HEAD
```

```
..  index:: gitk
```

You can also use visualization tools like *tig* ou *gitk*:

```
gitk ORIG_HEAD HEAD
gitk ORIG_HEAD --not --all
tig ORIG_HEAD..HEAD
```

Or:

```
tig ORIG_HEAD...HEAD
```

and you may want to toggle revision graph visualization with *g* key.

After an interactive rebase you may want to check the commits since the begining of the rebase in both branches. You will use:

```
git log --boundary --pretty=oneline --left-right  ORIG_HEAD...HEAD
```

And if your rebase went wrong you restore the previous state with:

```
git reset --hard ORIG_HEAD
```

If you have lost your ORIG_HEAD after a rebase because you did an other operation that reset it, you can still find the previous head which is now a dangling ref, unless you have garbage collected it.

You need to inspect your reflog and find the first commit before the rebase, in an interactive rebase the process begin with a checkout of the commit on which you rebase, so the previous commit was the head before the rebase:

```
git reflog
```

```
....
95512de HEAD@{7}:  rebase -i (pick): fixin typos
a1b9b5c HEAD@{8}: checkout: moving from master to a1b9b5c
c819a90 HEAD@{9}: commit: adding myfile.txt
```

In this example the previous head was the ninth older commit HEAD@{9} with an abbreviated commit c819a90.

9.4 dangling objects

The main section is the *garbage collection section*

After rebasing the old branch head is no longer in a branch and so it is dangling, it will be garbage collected when it will be no more referenced.

As explained in the previous section it is used in the reflog, so it will be garbage collected after expiring the reflog.

Sometime, when we are certain our rebase is correct and we will never want to come back to previous state, we want to clean these dangling objects. We use:

```
$ git prune
```

If we want to do the opposite, i.e. preventing this dangling commit to be lost some next garbage collection away we can point a new branch at it:

```
$ git branch <recovery-branch> <dangling-commit-sha>
```

CHAPTER 10

Fixing errors

10.1 Amend a commit

To amend the **last** commit:

```
$ #correct the errors, then
$ git add file
$ git commit --amend
$ # fix the commit message if necessary
```

To amend a past commit on a clean working directory, you need to come back to the erroneous commit, fix it and *rebase:* the latter commits on the the new fixed state.

An example adapted from user-manual: Rewriting a single commit

```
$ git log --stat -- badfile
$ git tag bad mywork~5
$ git checkout bad
$ # make changes here and update the index
$ git commit --amend
$ git rebase --onto HEAD bad mywork
$ git tag -d bad
```

The use of the tag is optional you can also do

```
$ BAD="7c2c66b71b"
$ git checkout $BAD
$ # make changes here and update the index
$ git commit --amend
$ git rebase --onto HEAD $BAD mywork
```

When you want to amend past commits with changes in the working tree, you cannot checkout the past commit because the worktree is dirty.

You can use git stash to come back to the original worktree or work in a linked temporary working directory:

```
$ git-new-workdir <repository> <temporary_workdir>
$ cd <temporary_workdir>
# find the commit to amend
$ git log --stat
$ BAD="<commit>"
$ git reset --hard $BAD
# edit the wrong file may be copying from <repository>
$ git add <changed file>
$ git commit --amend
$ git rebase --onto HEAD $BAD mywork
```

You can also use `git rebase --interactive` as indicated *in the rebase section*. where you find an *example of fixing an old error with interactive rebase*.

10.2 Commit a fixup and squash.

The *interactive rebase* can be made a lot simpler for fixing errors with the `git commit --fixup` or `git commit --squash command`.

On a clean worktree, or cleaned by a `git stash`, you change your erroneous file(s) and commit it (them) with

```
$ git commit --fixup=a0b1c2d3
```

Where you give the erroneous commit number, then you *fixup* the error with:

```
$ git rebase --interactive --autosquash a0b1c2d3^
```

The first command just use the original message prefixed by `fixup!`, the second one squash the original and next commit discarding the message of the fixup commit.

You can also do a simple commit and begin your message by `fixup!` followed by an initial section of the original commit message.

If instead of *fixup* you use *squash* the process is similar but the commit message for the folded commit is the concatenation of the messages of the first commit and of those with the *squash* command.

10.3 splitting a commit

ref: git-rebase: splitting commits, git gui

To split a commit, you first rebase interactively to the commit or one of its ancestor

```
$ git rebase -i <commit>
```

Then you mark the commit with the action *edit*, and when it comes to editing:

```
$ git reset HEAD^
```

Then you can examine the status with:

```
$ git status
```

and add some files and stage the appropriate hunks. It can be easy to use:

```
$ git gui
```

to commit the appropriate hunks in individual commits

Then you can as usual do:

```
$ git rebase --continue.
```

CHAPTER 11

Filter branch

This chapter describe some uses of `git filter-branch` before using it you should be aware that this command is destructive, and even the untouched commits end up with different object names so your new branch is separate from the original one. If ever you repository was shared anyone downstream is forced to manually fix their history, by rebasing all their topic branches over the new HEAD. More details in git-rebase(1) - recovering from upstream rebase

When filtering branches the original refs, are stored in the namespace `refs/original/`, you can always recover your work from there, but if you want to delete the previous state, after checking the new one is coherent, you need to delete these refs otherwise the original object will not be garbage collected. If you want to make experiments without the trouble to recovering from `refs/original` you should get a copy of your repository with:

```
git clone path_of_origin path_of_copy
cd path_of_copy
git branch --unset-upstream
git reset --hard
```

11.1 References

- The main reference is git documentation: filter-branch
- It is introduced in S. Chacon Pro-Git Rewriting History chapter and in Maintenance and Data Recovery - removing objects.

11.2 Removing an object or a directory

This can be done with `--tree-filter` or `-index-filter` as the second one does not check out the tree, it is a lot quicker.

When filtering branches you may remove all the changes introduced by some commit and ends up with empty commit. Some of these emty commits are useful because they have many parents, i.e. they record a merge.

To avoid such situation you can use `--prune-empty` (but it is incompatible with `--commit-filter`.

Your command will be:

```
git filter-branch --prune-empty --index-filter  \
    'git rm --cached --ignore-unmatch badfile' HEAD
```

Here the `git rm` command has the option `--cached` since we are working on the index and `--ignore-unmatch` because the file can be absent in the index for some commits, like those anterior to the first occurrence of the file.

If you rather want to delete a full directory content, you will add the `-r` option to make the remove recursive.:

```
git filter-branch --prune-empty --index-filter  \
    'git rm -r --cached --ignore-unmatch baddir' HEAD
```

If your object or directory is in many branch, cleaning HEAD will not get read of it, you should in this case clean all refs and filter all tags with:

```
git filter-branch --prune-empty --index-filter  \
    'git rm  --cached --ignore-unmatch badfile' \
    -tag-name-filter cat -- --all
```

If your unwanted blob has changed name along the history, it will still be kept with the olders name, but if you take care to find them with:

```
git log --name-only --follow --all -- badfile
```

After that your history no longer contains a reference to `badfile` but all the `refs/original/branch` and the reflog still do. You have to options, if you have no backup you should do:

```
git clone file:///path/to/cleanrepo
```

It is quick since done with hardlinks and the clone will not have the removed objects.

If you have yet done a backup as proposed *above* you can clean before repacking. After a filter-branch git keep *original* refs, that prevent the previously referenced object to become loose and be cleaned by garbage collection. If you want to get rid of them you delete these refs, on the other side if you want to keep them longer, you better rename them to prevent them to be overrode by some next operation (even if you can also control the original namespace with `--original` option).

```
git for-each-ref --format='%(refname)' refs/original | \
    xargs -n 1 git update-ref -d
```

Then your logs:

```
git reflog expire --expire=now --all
```

And you garbage collect all unreferenced objects with:

```
git gc --prune=now
```

More details in the section garbage collection.

Note: Many collaborative hosted repositories like GitHub, BitBucket and others, will not let you push back your deletes, so if you really want to be sure nobody can get your old file, you will have to delete these repos an push new ones.

CHAPTER 12

Git Subtree

Reference git-subtree(1) in contrib directory

Git subtree is an alternative from submodules, but while submodules are mainly aimed at putting in a directory of your repository an other project maintained somewhere else, and keeping the foreign repo in sync; git-subtree allow to keep separate a subproject and allow bidirectional collaboration between your main repo and the subprojects.

12.1 Subtree split

12.1.1 Extracting the branch

First you split a new branch from your history containing only the subtree rooted at <prefix>. The new history includes only the commits (including merges) that affected <prefix>. The commit in which where previously rooted in the subdirectory <prefix> are now at the root of the project.

```
git subtree split -P <prefix>  -b <name-of-new-branch>
```

12.1.2 Importing the branch in another repository

This is the simpler alternative, in the repository you pull your new branch:

```
git pull </path/to/old-repo> <name-of-new-branch>
```

12.1.3 Using the branch as master branch of a new repo

Then you create a new repo <new-repo> for the splitted branch:

```
cd <new-repo>
git init
```

You fetch the original branch and it is referred as a detached `FETCH_HEAD`.

```
git fetch </path/to/old-repo> <name-of-new-branch>
```

If you want to get it as master branch:

```
git checkout -b master FETCH_HEAD
```

12.1.4 Rebasing the branch in the new repo.

I show the example of rebasing on the top of the master, you can easily adapt it to an other rebase.

In this example I have an old repo *unix-memo* and I want to extract some part split as *git-extract* and include it on the top of the repo *git-memo*.

In *git-memo* I fetch the split branch:

```
$ git fetch ../unix-memo/ git-extract
....
From ../unix-memo
* branch            git-extract  -> FETCH_HEAD
```

I can examine the situation:

```
$ git log FETCH_HEAD
$ gitk master FETCH_HEAD
```

I want to rebase, but as some of my commits refer to the old repo *unix-memo*, I also want to amend them.

```
$ git rebase --interactive --onto HEAD --root FETCH_HEAD
```

I then mark as *edit* the commit I want to amend, then when the rebase is complete:

```
$ git rebase --continue
Successfully rebased and updated detached HEAD.
```

Now this HEAD is detached I can show it with:

```
$ git rev-parse HEAD
bb7d9a2a871e39e7f2f262f805221a41a4354f98
```

But it is easier to work on it with a temporary tag, then fast forward above *master*.

```
$ git tag detached
$ git checkout master
Previous HEAD position was bb7d9a2...
Switched to branch 'master'
$ git merge --ff-only detached
Updating 755786a..bb7d9a2
Fast-forward
....

$ git tag -d detached
Deleted tag 'detached' (was bb7d9a2)
```

12.1.5 Getting rid of the split branch.

You probably don't want anymore the split branch in your original directory, if you still want to keep the history (because it has influenced the past history of other files) you just remove in the original repo.:

```
git rm -rf <prefix>
git commit -m'moved new development of <name-of-new-branch>
    in a new repository'
```

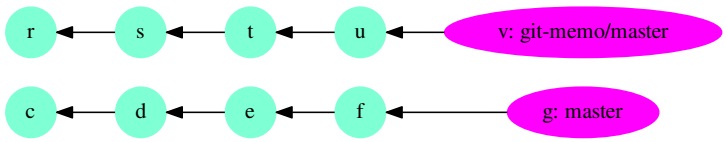
If you truly want to extract from the root of history of the original repository the split branch, and you can afford the cost of recovering from upstream rebase

12.2 adding a subtree to a project

We have a main project, with some commits in the master branch, and a remote project that we add to our repo with:

```
git remote add -f git-memo https://github.com/marczz/git-memo.git
```

Now our repository look like this.



We want to add the remote in a subtree of our main repo rooted at `./gitmemo`. We have to choose how we will mix the commits of the two branches, either they can be intermixed, or we can squash all the commits in one, before adding them. As the first option would mixe in the commit timeline completely foreign works, wechoose to squash with:

```
git subtree add --prefix gitmemo --squash git-memo master
```

And we obtain:

Later you can pull change from the remote repository with:

```
git subtree pull --prefix=git --squash git-memo
```

If you have one more commit on the remote, your commit tree is now like this:

CHAPTER 13

Repository Backup

13.1 file-system copy

A git repository can be safely copied to an other directory, or put in an archive file. This is a very basic way of keeping a backup of your repo.

```
cp -r myrepo backup_copy
tar -czf backup_copy.tgz myrepo
```

But such a frozen copy can not be updated.

13.2 git bundle

Git bundle let you pack the references of your repository as a single file, but unlike the tar command above the bundle is a recognized git source. You can fetch, pull clone from it.

```
$ git bundle create /path/to/mybundle master branch2 branch3
```

or to get all refs

```
$ git bundle create /path/to/mybundle --all
```

You can then check what is in your bundle:

```
$ git bundle list-heads /path/to/mybundle
```

After moving the bundle on any location you can get a new repository by:

```
$ git clone -b master /path/to/mybundle newrepo
```

If you want to continue your development in the first repository you can do as follow:

```
$ git tag lastsent master
#.... do some commits
$ git bundle create /path/to/newcommitbundle lastsent..master
```

Then send it by email or any dumb transport, and in the repository on the other side:

```
$ git bundle verify /path/to/newcommitbundle
$ git pull /path/to/newcommitbundle master
```

More details in git-bundle(1) Manual Page and Git's Little Bundle of Joy

A discussion is in Stackoverflow: Backup a Local Git Repository

13.3 gcrypt remote

Joey Hess gcrypt remote offer a great solution to backup to an unsecure location.

You should first install `git-remote-gcrypt` from his repository https://github.com/joeyh/git-remote-gcrypt.git

Then create the encrypted remote by pushing to it:

```
git remote add cryptremote gcrypt::rsync://example.com:repo
git push cryptremote master
```

More information in the GitHub repository

and in Joey Hess: fully encrypted git repositories with gcrypt and git-annex special remote: gcrypt

13.4 git mirror

It is more interesting to use git do do backups.

You can mirror your repository to a bare repository with:

    git clone –mirror myrepo repo_backup.git

repo_backup.git has the same branches than *myrepo* an an "origin" remote pointing to *myrepo*. You can update the copy from repo_backup.git by pulling *origin*.

13.5 mirror remote

In the previous solution the backup repository track its origin, but the original repository knows nothing about its backup. It ca be solved by using instead a mirror remote.

You first create an empty bare repository:

```
git init --bare /path/to/reposave.git
```

Then in *myrepo*:

```
git remote add --mirror=push backup file:///path/to/reposave.git
git push backup
```

The backup repository will be updated by repeating the last command.

If you want to be able to recover in the backup from a bad push, you may want the backup to keep a reflog. As reflog are disabled by default in bare repository you shoul set it in *reposave.git*:

```
git config core.logAllRefUpdates true
```

13.6 live mirror

Sometime you may want a mirror to experiment a dangerous operation. In this case the mirror is not a backup we can better think as the origin as the backup. The `clone --mirror` is not the solution since it creates a bare repository. And a simple clone is not either even if it as all reference, because it has only one branch, the other branch are just registered as `remotes/origin/name`. *

As you what to make experiments in this repository you probably don't even want to track the origin, if you succeed you will replace the original one, if you fail you will remove the repository.

You first do a simple clone:

```
git clone myrepo /path/to/experimental
```

Then create all missing branches:

```
cd /path/to/experimental
git branch --no-track branch2 remotes/origin/branch2
git branch --no-track branch3 remotes/origin/branch3
```

Delete the refs to origin:

```
git branch -d remotes/origin/master remotes/origin/branch2 \
    remotes/origin/branch3 remotes/origin/HEAD
git remote remove origin
```

And play with *experimental*

Git for site deployment

On the remote:

```
$ mkdir website.git && cd website.git
$ git init --bare
Initialized empty Git repository in /home/user/website.git/
```

Define a post-receive hook:

```
$ cat > hooks/post-receive
#!/bin/sh
GIT_WORK_TREE=/var/www/www.example.org git checkout --force
$ chmod +x hooks/post-receive
```

On the main site define a remote:

```
$ git remote add web ssh://server.example.org/home/user/website.git
$ git config remote.web.push "+master:refs/heads/master"
$ git push web
```

Reference: Using Git to manage a web site by Abhijit Menon-Sen

Ian Bicking does not like this strategy and prefer to:

> copy the working directory to a new location (minus .git/), commit that, and push the result to your deployment remote. You can send modified and untracked files, and you can run a build script before committing and push the result of the build script, all without sullying your "real" source control.
>
> —Ian Bicking in this blog

He writed git-sync automating this task

CHAPTER 15

Git keyword expansion

We can archive the files using keywords defined in gitattributes by using the command git archive.

We archive files by:

```
git archive --format=tgz --worktree-attributes HEAD file1 file2 ...
```

The attributes are taken from the tree, and with the option `--worktree-attributes` in the checked out worktree.

Git also look at the repository local file `$GIT_DIR/info/attributes`

If the attribute `export-subst` is set for a file then git will expand several placeholders with the syntax `$Format:PLACEHOLDERS$`.

The place holders are defined in git log pretty formats

I use in my C source files:

```
/* @date   $Format: %ci$
 * @version Commit: $Format: %h$
 * Tree: $Format: %t$
 */
```

To exports the committer date `%ci`, the abbreviated tree hash `<%t`, and the abbreviated commit hash `%h`

CHAPTER 16

Git Hooks

Hooks are little scripts you can place in *$GIT_DIR/hooks* directory

refs: githooks, Pro Git: Git Hooks

16.1 prepare_commit_msg

Ref: gitdoc: prepare_commit_msg

This hook is invoked by git commit after preparing the default message, and before the editor is started. It takes one to three parameters.

- The name of the file that contains the prepared message.
- The second is `message` if a `-m` or `-F` option was given, `template` with a `-t` option or configuration `commit.template`, `commit` if a `-c`, `-C` or `--amend` option was given, `merge` for a merge commit or with a `.git/MERGE_MSG` file; and `squash` if a `.git/SQUASH_MSG` file exists.
- The third is only for a `commit` and is the commit SHA-1.

A non-zero exit means a failure of the hook and aborts the commit.

```
#!/bin/sh
num_messages=5
format="# %h %s [%an]"
log="$(git log --pretty="${format}" -${num_messages})"
header="#
# last ${num_messages} messages
# ----------------------"
template() {
    echo "${header}"
    echo "${log}"
}
case "$2" in
merge|template|squash) ;;
""|commit) template >> $1;;
*)  echo "error in prepare-commit-msg hook" >&2
    exit 1
    ;;
esac
```

CHAPTER 17

Garbage Collecting

Refs: Git user manual: dangling-objects

git-gc(1), git-fsck(1) git-prune(1), git-pack-refs(1), git-repack(1), git-prune-packed(1), git-reflog(1).

Maintenance and Data Recovery has some examples of repository cleaning.

In a repository, some object become unreachable by any refs, during some operations, like deleting a branch, deleting an unreachable tag, rebasing, expiring entries in the reflog ... These unreachable objects can be reported by git-fsck.

17.1 git fsck

If you use the default options:

```
$ git fsck
dangling tree 4df800e6a1c6ba57821e4e20680566492bbb5e81
```

report only the dangling objects.

You can add the object unreachable by any reference with:

```
$ git fsck --unreachable
unreachable tree 6e946512b1b4841dff713bb65e78a8ddbf0171d3
unreachable tree 4df800e6a1c6ba57821e4e20680566492bbb5e81
unreachable tree 125a284a7a428d91e199a3c9d7f7a834613d1d13
```

Both commands above use also the reflog. If you want to look at the object which are dangling except from the reflog or object unreachable except the reflog you do:

```
$ git fsck --dangling --no-reflogs
dangling commit 2b21a6a8e775a43eafbe1b9e8b1fb5debe77b2ee
dangling commit c3e1ecc9f0e7a67c9a05db92d6c6548a1c965830
dangling commit 1702677e8ca31c0536745b36280397457bf20002
dangling commit b746f71fa14416e22920f38b8439bbb1972481a3
dangling commit 408c05cafd2bfdb861676dd54a0ed083f7fdfcaa
dangling commit 6c92fe1d7f47e397c3ccd2713ddd9c3b8239d1c8
...
$ git fsck --unreachable --no-reflogs
unreachable commit 2b21a6a8e775a43eafbe1b9e8b1fb5debe77b2ee
unreachable tree 39610cc0e1126a2bb73cb3345b9228f1ccb374d9
unreachable commit c3e1ecc9f0e7a67c9a05db92d6c6548a1c965830
unreachable tree e981cec3e50cf10b59b544d86be681a7030cb0a6
...
```

17.2 Automated garbage collection with `gc --auto`.

Git stores the objects either one by one as *loose* objects, or with a very efficient method in *packs*. But if the size of packs is a lot lesser than the cumulated loose objects, the access time in a pack is longer. So git does not pack the objects after each operations, but only check the state of the repository with `gc --auto`.

`gc --auto` look if the number of loose objects exceeds `gc.auto` (default 6700) and then run `git repack -d -l` which in turn run git-prune-packed. setting `gc.auto` to 0 disables repacking. When the numbers of packs is greater than `gc.autopacklimit` (default 50, 0 disable it) `git gc --auto` consolidates them into one larger pack.

When doing a `git gc --aggressive` the efficiency of git-repack depends of gc.aggressiveWindow (default 250).

`git gc --auto` also pack refs when `gc.packrefs` has its default value of `true`, the refs are then placed in a single file `$GIT_DIR/packed-refs`, each modification of a ref again create a new ref in `GIT_DIR/refs` hierarchy that override the corresponding packed ref.

The `gc` command also run by default with the `--prune` option, which clean unreachable loose objects that are older than `gc.pruneExpire` (default 14 days). If you want to use an other date you have to add `--prune=<date>`, `--prune=all` prunes loose objects regardless of their age. *Note that it may not be what you want on a shared repository, where an other operation could be run concurrently.*

An unreachable commit is never pruned as long it is in a reflog(1), but `gc --auto` run `git reflog expire` to prune reflog entries that are older than `gc.reflogExpire` (default 90 days) or unreachable entries older than `gc.reflogExpireUnreachable` (default 30 days). These values can be also configured for each individual ref see git-config(1).

Records of conflicted merge are also kept `gc.rerereresolved` (default 60 days) or `gc.rerereunresolved` (default 15 days) for unresolved merges.

17.3 Forced garbage collection.

You can have an idea of the state of your repository by issuing `git count-objects -vH`

After some operation that creates a lot of unreachables objects, like rebasing or filtering branches you may want to run `git gc` without waiting the three months of expirability. This is also a necessity if you have to delete an object, now unreachable, but that contains some sensible data, or a very big object that was added and then deleted from the history (see the *filter branch section*).

As the operation is recorded in the reflog, you expire it with:

```
git reflog expire --expire=now --all
```

And you garbage collect all unreferenced objects with:

```
git gc --aggressive --prune=now
```

CHAPTER 18

Miscellaneous operations

18.1 switching branches without doing a checkout

Refs: git symbolic-ref, git reset,

```
$ git symbolic-ref HEAD refs/heads/otherbranch
```

For every work on the branch you must get a fresh index with:

```
$ git reset
```

The script `git-new-workdir` in the contrib directory creates a symlink to a repository, optionally with a new checked out branch:

```
$ git-new-workdir <repository> <newworkdir> [<branch>]
```

18.2 Transparent encryption

There are two different use of encryption, the first is when your un-encrypted repository tree is convenient on your own server but you don't want to push un-encrypted data on remote. You have to alternative, either you encrypt the sensible files in the repository, and then you can push freely the repository content, or you don't push to unsecure remote but with a backend like the *gcrypt encrypted remote* or even you store encrypted *bundles*.

If your data is truly sensible, you should not let it un-encrypted even on your preferred server, you can then either encrypt it in the repository tree, and nothing else is necessary, or have the repository and working tree in an encrypted filesystem, and apply one of the previous solution to backup your repo, or to encrypt the data in the repository in the unencrypted filesystem, and decrypt it on-the-fly in the working-tree which is itself in an encrypted filesystem.

You have to be aware that encrypted files can lead to a very inefficient storing because close versions of a file when encrypted do not have a *delta* allowing packing. See git-pack-objects(1) Manual Page for details

I show an elementary example of transparent encryption.

In a new repository I put a big text, and look at the repo size:

```
$ git add DavidCopperfield.txt
$ git commit -m'added DavidCopperfield.txt'
$ du -sk DavidCopperfield.txt
1968        DavidCopperfield.txt
$ du -sk .git
1052        .git
```

Now I change a sentence in the text, and check in my work:

```
$ git diff --numstat
1       0       DavidCopperfield.txt
```

Now I look at the size of my repo:

```
$ du -sk DavidCopperfield.txt
1968        DavidCopperfield.txt
$ du -sk .git
1960        .git
```

My repo is nearly twice as big as previously, not very nice. But I remember that git use *loose* object until told to pack, so I do:

```
$ git gc
Counting objects: 8, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (8/8), done.
Total 8 (delta 1), reused 0 (delta 0)
$ du -sk .git
892 .git
```

A lot, better.

Now I do the same experiment with an encrypted file.

I first create some script ~/gitencrypt/passphrase:

```
pass="my secret passphrase"
```

~/gitencrypt/clean_filter:

```
#!/bin/sh

salt=82acb021e056fc9e8f75a5fe # 24 or less hex characters
. ${0%/*}/passphrase
openssl enc -base64 -aes-256-cbc -S $salt -k "$pass"
```

~/gitencrypt/smudge_filter:

```
#!/bin/sh

. ${0%/*}/passphrase
openssl enc -d -base64 -aes-256-cbc -k "$salt"
```

~/gitencrypt/diff_filter:

```
!/bin/sh

. ${0%/*}/passphrase
openssl enc -d -base64 -aes-256-ecb -k "$pass" -in "$1"
```

and in my `.git/config` I add:

```
[filter "openssl"]
    smudge = /tmp/gitencrypt/smudge_filter
    clean = /tmp/gitencrypt/clean_filter
[diff "openssl"]
    textconv = /tmp/gitencrypt/diff_filter
[merge]
    renormalize = true
```

Then I add the same file I did previously:

```
$ git add DavidCopperfield.txt
$ du -sk .git
1036        .git
```

I do the same sentence change than previously, then:

```
$ git diff --numstats
1       0       DavidCopperfield.txt
$ git add DavidCopperfield.txt
$ du -sk .git
1928        .git
$ git gc --prune
Counting objects: 2, done.
Writing objects: 100% (2/2), done.
Total 2 (delta 0), reused 0 (delta 0)
$ du -sk .git
1928        .git
```

Git cannot pack gpg encoded files because even if only a sentence differ every block of encrypted file is completely different.

The approach used here is almost identical to the one proposed by Woody Gilk in Transparent Git Encryption accompanied with a set of scripts git-encrypt

A similar, more polished approach is Andrew Ayer git-crypt.

The inability to pack an encrypted directory was signaled by Junio Hamano in the article: Re: Transparently encrypt repository contents with GPG or look at the article thread.

18.3 Using git-wip

Refs: git-wip repository and README

git-wip is a script that will manage Work In Progress branches.

To show the log of the commits in wip/master and not in master:

```
$ git log master..wip/master
```

You can add `-p` to see what is added:

```
$ git log -p master..wip/master
```

Here as usual for a git revision range `master..wip/master` means all the commit in `wip/master` which are not in `master`.

To see the what is in wip and not committed to master you do:

```
$ git diff master...wip/master
```

This shows the diff between the common ancestor of master and wip/master and master.

```
$ git diff master..wip/master
```

is the same than:

```
$ git diff master wip/master
```

And represent the difference beetween master and wip/master, this is probably **not what you want** because if you have committed something since the last *wip*, master is not an ancestor of wip/master, so this diff will also undo whatever is committed since the common ancestor.

CHAPTER 19

Indices and tables

- *genindex*
- *search*

R

S

T

U

V