A practical git introduction

Introduction

In just a few years, git has become the dominant version control system in the software industry. Despite its widespread use, it often still appears as either magical or cumbersome when its core concepts are not fully grasped. This post is a walkthrough of practical git usage that will detail how git internally handles things.

1. Introduction
    1. Why a version control system?
    2. Why git?
    3. Setup
2. git basics
    1. First commit!
    2. Staging area
    3. Data store
        - Objects
        - Physical storage
    4. Immutability
    5. Reflog
    6. tl;dr
3. Branches and remotes
    1. Branches
    2. Comparing
        - Referencing parent commits
        - Comparing content
        - Comparing commits
        - Comparing branches
    3. Merging
        - Three-way merge
        - Fast forward
        - Resolving conflicts
    4. Rebasing
        - Rewriting history
    5. Rebase caveats
    6. Remotes
        - Push

        - Fetch
        - Pull
        - Force push
    7. tl;dr
4. Good practices
    1. Workflow
        - Branching flow
        - Merge or rebase?
    2. What's a good commit?
        - Atomicity
        - Hiding sausage making
        - Commented hunks of code
    3. What's a good commit message?
        - Formatting
        - Content
        - Documentation
5. Going further
    1. Options
    2. Searching
    3. Reset
    4. Cherry pick
    5. Stash
    6. Bisect
6. Environment
    1. Command-line completion
    2. Configuration
        - Local/global/system
        - Basics
        - Aliases
    3. Hooks
    4. Custom command-line prompt
    5. Protocols
    6. GUIs and plugins
    7. Jargon
        - upstream/downstream
        - bare and non-bare repositor
        - fork
        - pull request
        - porcelain/plumbing
        - gist
        - hunk
7. References

Why a version control system?

People not already using a version control system (vcs) often perform some manual operations to keep incremental revisions of some work. Keeping iterative versions of a document or a collection of documents may be done through naming schemes like `[filename]_v{0-9}+.doc` or `[timestamp]_[filename]_[comment].zip` (where e.g. using the ISO 8601 format for dates will sort version). However it becomes quickly cumbersome to easily see compare versions, to undo some modification or to work in parallel on the same document and not mentionning the possibility of making a mistake when freezing a version.

This is where a vcs becomes handy if not mandatory to use. A vcs will store version of a collection of documents without modifying their apparent filenames, allow to undo/redo some modification and keep a context in which modification were performed (e.g. an author, timestamp, comment for the modification).

Why git?

Versioning code is not a new problem. Alternatives to git, such as subversion (svn), concurrent versions system (cvs), perforce are "old" vcs softwares. Wether git is better than those softwares or not will not be discussed here. However let's list some of the attractive features provided by git:

- free/open source/binaries available for all major platforms
- decentralized
    - users keep a local clone, work locally at their own pace and deliver their changes when ready
    - every local clone is basically a (potentially partial) backup
- handle very large codebase and/or long history (git was built to handle the Linux kernel)
    - fast (most operations are performed locally)
    - reliable (data is mostly immutable)
- flexible regarding workflows
- with the help of github, it is now becoming *the* standard decentralized vcs
- git philosophy is to perform simple operations and let complex ones — that actually occur not so frequently — to the user so it does no black magic

> *One of the things that makes Git a pleasure to use for me is that I actually trust what Git does, because what Git does in the end is very, very stupid.*
>
> — Linus Torvald

Setup

Before running git commands, we need:

- git installed (preferably ≥ 2.0.0 as the default behavior for some commands has been greatly improved in late versions; see git 2.0.0 changelog)
- a git user i.e.:
  - a name `git config --global user.name "My Name"`
  - an email `git config --global user.email "me@mail.org"`

Let's create a dummy repository

```
$ git init $HOME/bonjour
Initialized empty Git repository in /Users/marc/bonjour/.git/

$ cd $HOME/bonjour

$ tree -a -I hooks
.
└── .git
    ├── HEAD
    ├── config
    ├── description
    ├── info
    │   └── exclude
    ├── objects
    │   ├── info
    │   └── pack
    └── refs
        ├── heads
        └── tags
```

git just created a hidden repository to contain internal data that we'll describe later and that's it, we've got a git repository!

git basics

First commit!

Before diving quickly into git internals, let's create our first commit:

```
$ git status
On branch master

Initial commit
```

```
nothing to commit (create/copy files and use "git add" to track)

$ echo "A dummy app listing ways to just say 'hello'" > README.md

$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.md

$ git add README.md

$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   README.md

$ git commit -m "First commit"
[master (root-commit) 45de2f7] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md

$ git status
On branch master
nothing to commit, working directory clean
```

Note that at each step, the `git status` command provides helpful overview of the current repository state and a *contextual* help.

Staging area

git uses a two-phase commit:

1. add desired changes to the staging area
2. commit changes.

After having changed some files, we may select the changes that should be committed using:

- `git add` to add a new file or some modification in an already tracked file; the `--patch` option allows to select hunks only
- `git rm` to remove a file from tracking
- `git mv` to rename a file

The staging area is an important aspect of git as it is what connects the local file system to the internal git storage. It is described by `git status` as *"changes to be committed"* and thus it is important to think of it as a commit draft.

```
$ mkdir {fr,en}

$ echo 'bonjour' > fr/data

$ echo 'hello' > en/data

$ git add fr/data

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  new file:   fr/data

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  en/
```

Notice that we do not have to stage all local modifications. Once we are done selecting changes to be committed, we may actually commit those changes.

```
$ git commit -m "Create french data file"
[master 13d1b4b] Create french data file
 1 file changed, 1 insertion(+)
 create mode 100644 fr/data
```

This example is really simple; a more realistic workflow would be:

1. make local changes

2. stage meaningful changes/unstage undesired ones (using `git reset [--patch] HEAD file`)
3. go to 1. until staged changes perform the desired goal
4. commit the changes.

Now that we have two commits, we may look at how git handles our data internally to better understand git mechanics.

Data store

Objects

After creating a commit, `git` displays a unique id for the newly created commit. We may use some git commands to inspect this data

```
$ git cat-file -t 13d1b4b
commit

$ git cat-file commit 13d1b4b
tree 7efc3caa79efbab80f45335d4d5f8d2885daff29
parent 45de2f713305a9dcd3e82833653153c19081f36e
author Marc Helbling <marc.d.helbling@gmail.com> 1413120925 +0200
committer Marc Helbling <marc.d.helbling@gmail.com> 1413120925 +0200

Create french data file
```

From this, we see that a commit references:

- a "tree": git internal description of the filesystem
- a parent: git is fundamentally a **direct acyclic graph** (DAG) in which nodes are commits that references their parent commit(s)
- an author: the person who originally wrote the current commit *content*
- an author date
- a committer: the person who created the git commit on behalf of the author (which, as in our case, may be the same person as the author)
- a committer date
- a message: the description for the changes contained in the commit.

This is the high level definition of a commit in git. We need to go one more step into internals and inspect the "tree" object to have a better picture of how git structures data.

```
$ git cat-file tree 7efc3caa79efbab80f45335d4d5f8d2885daff29
```

```
100644 README.mdF??Bd???"?C-%D?J?x40000 fr\bbS?"}nC??WJ!

$ git ls-tree 7efc3caa79efbab80f45335d4d5f8d2885daff29
100644 blob 4695a64264e4d7ea22d9432d25449f1e4aeb781e    README.md
040000 tree 5c7f626253bb14227d6e074382ee91574a180f21    fr

$ git cat-file blob 4695a64264e4d7ea22d9432d25449f1e4aeb781e
Hello

$ git ls-tree 5c7f626253bb14227d6e074382ee91574a180f21
100644 blob 1cd909e05d33f0f6bc4ea1caf19b5749b434ceb3    data

$ git cat-file -p 1cd909e05d33f0f6bc4ea1caf19b5749b434ceb3
bonjour
```

So we can see that:

- a "tree" contains pointers to "blobs" and other trees and a name for each pointer
- a "blob" is bunch of bytes representing user content (text, images etc.)
- both trees and blobs store a file mode (i.e. a `chmod`); note however that the file owernship (`chown`) will depend on the user that performs the git commands and is up to the final user
- git performs deduplication based on content: if file `foo` and file `bar` are a copy of each other
    - they will be represented by the same blob
    - the tree will point to two blobs with the same id but refering different names and possibly different file modes

There are 4 git objects (listed from "low" to "high" level) that can be described as:

- blob: content
- tree: file tree description
- commit: project snapshot with contextual metadata
- tag: frozen commit name (we will not cover tags and refer to the documentation for more details)

Physical storage

As git handles history of files, we may ask ourselves how does git stores incremental differences for our data. To test this, let's add some content in an existing file

```
$ echo "salut" >> fr/data
```

```
$ git add -u

$ git commit -m "Add more frensh data"
[master 456a082] Add more frensh data
 1 file changed, 1 insertion(+)
```

and inspect the resulting blob object

```
$ git ls-tree 456a082
100644 blob 4695a64264e4d7ea22d9432d25449f1e4aeb781e    README.md
040000 tree 3d9ba4d12442602bd81928438f80810622b9fd56    fr

$ git ls-tree 3d9ba4d12442602bd81928438f80810622b9fd56
100644 blob bd61b2ccb39197cc3a66b43f52a6fed66a237a29    data

$ git cat-file blob bd61b2c
bonjour
salut
```

The conclusion is that git stores pointers to "full" blobs which means that a blob is useful by itself, independantly of the history file it represents. Practically, this means that even a shallow repository is usable (especially for git≥1.9.0). This could seem inefficient as for each file, git will keep a copy of the full content after each commit. However, git may also create "packfiles" that represent content 'delta's to optimize disk usage. Time to examine how the data is actually stored.

```
$ cat .git/objects/bd/61b2ccb39197cc3a66b43f52a6fed66a237a29
xK??OR04aH????/-?*N?)-?S?a

$ python -c """
import zlib, sys;
print repr(zlib.decompress(sys.stdin.read()))
""" < .git/objects/bd/61b2ccb39197cc3a66b43f52a6fed66a237a29
'blob 14\x00bonjour\nsalut\n'

$ wc -c fr/data
      14 fr/data
```

git physically stores content using zlib-compressed files containing:

1. the object type (blob, tree, commit or tag)
2. the uncompressed data size

3. a null byte
4. the actual content

The commit files follow the same construction. Tree storage is slightly different. The zlib compression will however prevent from using too much disk space for storing commits (especially if blobs represent large files).

The last important point for this quest is to understand how git names his internal files. Those names correspond to a cryptographic hash of the object. The hash function used is SHA-1 and it may serve as a signature to assert data integrity (i.e. the decompressed object sha1 signature should match its filename and the content size should be the same as the size stored in the object). sha1 produces 160-bit hash value that git represents as a 40 digits long hexadecimal value. You may have noticed that when referencing git objects, we did not always used a 40 digits long value every time. git allows to use a shorter sub-sha1 *prefix*, provided that it is not ambiguous (i.e. that it enables to reference an object uniquely). It basically means that the bigger your repository (in terms of git objects), the longer the sha1 prefix you will have to use.

```
$ cat fr/data | git hash-object --stdin
bd61b2ccb39197cc3a66b43f52a6fed66a237a29

$ tree .git/objects/ --matchdirs -P bd
.git/objects/
├── 13
├── 1c
├── 3d
├── 45
├── 46
├── 5c
├── 77
├── 7e
├── bd
│   └── 61b2ccb39197cc3a66b43f52a6fed66a237a29
├── cf
├── info
└── pack
```

git shards objects into 16×16=256 subfolders to grant a faster access to a given sha1. This matters as all manipulations in git involve a sha1 (sometimes through aliases). In the end, we have spent a bit of time looking at how git stores *loose* objects i.e. how git stores its objects in individual files. This helped us getting the big picture of the internal storage.

We will not dig the packfile format which is an optimization to avoid cluttering the disk by regrouping objects together (typically by invoking `git gc`).

Immutability

When creating our last commit, we made an horrible typo. git allows to amend the last commit using `git commit --amend`. If the staging area contains any modification, they will be added to the commit. In our case, we just want to fix our typo so we do not add anything to the staging area:

```
$ git commit --amend
```

opens our favorite editor (as defined by the `$EDITOR` environment variable) where we may edit the commit message

```
Add more french data

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Oct 12 16:19:08 2014 +0200
#
# On branch master
#
# Changes to be committed:
# modified:   fr/data
#
```

and simply save our modification.

```
[master dd0f5d6] Add more french data
 Date: Sun Oct 12 16:19:08 2014 +0200
 1 file changed, 1 insertion(+)
```

We see that git creates a *new* commit object `dd0f5d6` (remember that the sha1 involves the commit message and timestamps).

```
$ git cat-file commit dd0f5d6
tree 77a832b508bd5d2fb7c1eb8999e6e0a9f926434d
```

```
parent 13d1b4b062b7a7308553bc504dda2d43d32525ba
author Marc Helbling <marc.d.helbling@gmail.com> 1413123548 +0200
committer Marc Helbling <marc.d.helbling@gmail.com> 1413725760 +0200

Add more french data
```

This means that a commit is immutable; any modification creates a new commit instead of modifying the existing one thus making git a functional DAG. We will later see that this is an important property to have in mind when multiple people are working on the same repository. The benefit is that every object is uniquely defined and when you manipulate a sha1 you do not have to worry about not having the correct content.

Reflog

If we look at our commit tree

```
$ git log --graph --oneline
* dd0f5d6 Add more french data
* 13d1b4b Create french data file
* 45de2f7 First commit
```

we see that we created 3 commits until now. Amending our last commit did not add a new commit in the tree, it only replaced one commit by an other commit. We could fear that any git command have a direct and impactful consequence.

However git keeps a reflog which is a record of all commits that were referenced at some point.

```
$ git reflog
dd0f5d6 HEAD@{0}: commit (amend): Add more french data
456a082 HEAD@{1}: commit: Add more frensh data
13d1b4b HEAD@{2}: commit: Create french data file
45de2f7 HEAD@{3}: commit (initial): First commit
```

Any action that has been **committed** can be retrieved later, even if it is no longer referenced in the commit tree. This allows to undo bad commands very easily and may serve as a safety net.

Note however that git has garbage collection commands that will remove unreachable objects. By default those commands will not remove objects that were

created in the last 2 weeks.

git basics: tl;dr

- git is a functional DAG where nodes represents filetrees with metadata and keep a link to their parents
- staging area is the bridge between local file tree and git data store
- git internal data stored efficiently and safely in the `.git` folder
- git commands performed by passing sha1 prefixes that identify objects uniquely
- when lost, run `git status`
- `git reflog` records a reference to all created commits even when no longer reachable
- don't mess with the `.git` folder!

Branches and remotes

Branches

As we have seen in the previous section, git is a graph and you may have noticed that `git status` names branches:

```
$ git status
On branch master
nothing to commit, working directory clean

$ git branch
* master
```

By default, git creates a branch called `master`. If we inspect this object

```
$ git cat-file -t master
commit

$ git show --oneline master
dd0f5d6 Add more french data
diff --git a/fr/data b/fr/data
index 1cd909e..bd61b2c 100644
--- a/fr/data
+++ b/fr/data
@@ -1 +1,2 @@
 bonjour
```

```
+salut
```

we realize that a branch simply is a pointer to a "leaf" commit also called "tip" commit. The binding name/commit is stored in

```
$ tree .git/refs/heads/
.git/refs/heads/
└── master

$ cat .git/refs/heads/master
dd0f5d6500d72d54747dec1dc4139f13b5fdb8f2
```

git also keeps an alias for current branch last commit as `HEAD` (with a special case for `detached HEAD`).

```
$ git show --oneline HEAD
dd0f5d6 Add more french data
diff --git a/fr/data b/fr/data
index 1cd909e..bd61b2c 100644
--- a/fr/data
+++ b/fr/data
@@ -1 +1,2 @@
 bonjour
+salut

$ cat .git/HEAD
ref: refs/heads/master
```

Creating new branch is very easy with git:

```
$ git branch structure-data

$ git branch
  structure-data
* master

$ git checkout structure-data
Switched to branch 'structure-data'

$ git branch
* structure-data
```

```
  master

$ git show --oneline HEAD
dd0f5d6 Add more french data
diff --git a/fr/data b/fr/data
index 1cd909e..bd61b2c 100644
--- a/fr/data
+++ b/fr/data
@@ -1 +1,2 @@
 bonjour
+salut
```

We see that our brand new branch points exactly to the same commit as our `master` branch. This is the default behavior when creating a new branch; it is assumed that the new branch will start from `HEAD` and may be changed by passing the desired branching commit sha1 as a second argument i.e. `git branch new_branch new_branch_HEAD_commit`.

```
$ echo -e "# old\n\nbonjour\nsalut" > fr/data

$ git add fr/data && git commit -m "add 'old' header to french data"
[structure-data dd0109a] add 'old' header to french data
 1 file changed, 2 insertions(+)


$ echo -e "\n# modern\n\nyo" >> fr/data

$ git add -u && git commit -m "add modern french data"
[structure-data 5a40d5a] add modern french data
 1 file changed, 4 insertions(+)
```

Comparing

Now that we have two distinct branches, we should make sure that the changes introduced by our new branch match the intended specification. We therefore need to see the differences in content and commits.

Referencing parent commits

- `abc123^` parent commit of commit `abc123`
- `abc123^^` grandparent commit of commit `abc123`
- more generally `abc123~n`
    - `abc123~1` ⇔ `abc123^`

    - `abc123~2` ⇔ `abc123^^`

Comparing content

- `git diff --cached`: changes that have been staged
- `git diff A B`: changes (computed using the longest common subsequence algorithm algorithm)
- `git diff A...B`: changes from common ancester of `A` and `B` to `B`

Comparing commits

- `git branch --contains sha1`: list all branches containing the commit `sha1`
- `git log A --not B`: list commits contained in branch `A` that are not in branch `B`

Comparing branches

- `git branch --merged`: list branches that are reachable in the current branch history
- `git branch --no-merged`: list branches that are not reachable in current branch history

If we compare our branch content with the `master` branch:

```
$ git diff master structure-data
diff --git a/fr/data b/fr/data
index bd61b2c..ea35d1f 100644
--- a/fr/data
+++ b/fr/data
@@ -1,2 +1,8 @@
+# old
+
 bonjour
 salut
+
+# modern
+
+yo

$ git log --oneline structure-data --not master
5a40d5a add modern french data
dd0109a add 'old' header to french data

$ git checkout master && git branch --merged
* master
```

we see that we have been structuring our data and adding new content in two commits that are ready to be merged.

Merging

As we are happy with the changes introduced in our new branch, we may now make our work available in the repository trunk. There are multiple strategies to perform this, the most basic one being a merge commit. In its simplest — yet most common — form, a merge will take 2 branches and create a commit having the branches respective HEADs for parents through a merge.

Three-way merge

Three-way merge is a central algorithm in git. As its name suggests, it involves 3 distinct commits:

- the `MERGE_HEAD` commit i.e. the modification that we want to merge
- the `HEAD` commit i.e. the branch in which the `MERGE_HEAD` will be merged i.e. the branch on which the `git merge` command is called
- the `ORIG_HEAD` commit i.e. the *best common ancester* of `MERGE_HEAD` and `HEAD` that will serve as the reference.

The result of a three-way merge will look like: `HEAD + (MERGE_HEAD - ORIG_HEAD)`

HEAD	+ MERGE_HEAD	- ORIG_HEAD	→ result
foo	foo	foo	foo
foo			foo
	foo		foo
	foo	foo	
foo		foo	
foo	bar	baz	*conflict*

This short overview of how the three-way merge works assume that `ORIG_HEAD` is unique which can be wrong when merge commits are involved too. By default, git uses the `merge-recursive` strategy that is a three-way merge where the `ORIG_HEAD` is a (virtual) merge commits for all common ancestors.

So, after running

```
$ git merge --no-ff structure-data
> Merge branch 'structure-data'
```

```
>
> # Please enter a commit message to explain why this merge is necessary
> # especially if it merges an updated upstream into a topic branch.
> #
> # Lines starting with '#' will be ignored, and an empty message aborts
> # the commit.
Merge made by the 'recursive' strategy.
 fr/data | 6 ++++++
 1 file changed, 6 insertions(+)

$ cat fr/data
# old

bonjour
salut

# modern

yo
```

the commit tree now looks like

```
$ git log --graph --oneline
* 62cbf27 Merge branch 'structure-data'
|\
| * 5a40d5a add modern french data
| * dd0109a add 'old' header to french data
|/
* dd0f5d6 Add more french data
* 13d1b4b Create french data file
* 45de2f7 First commit
```

Fast forward

When merging, we explicitely asked git to create a merge commit using the `--no-ff` flag. However, looking at the commit graph, we see that it is (almost) equivalent to the simplified one

```
* yyyyyyy add modern french data
* xxxxxxx add 'old' header to french data
|
* dd0f5d6 Add more french data
```

```
* 13d1b4b Create french data file
* 45de2f7 First commit
```

Indeed, in this case, `HEAD` and `ORIG_HEAD` pointed to the same commits hence `HEAD` may just be updated to `MERGE_HEAD`. Wether one should prevent fast-forward merges or not is a matter of workflow and we will discuss this point a bit later.

Resolving conflicts

We have seen that in some situations, the three-way merge results in a situation where the merge content may not be automatically deduced. This is called a merge conflict. Let's create a conflict:

```
$ git checkout -b modern-french
Switched to a new branch 'modern-french'

$ echo "jourbon" >> fr/data

$ git add -u && git commit -m "add new modern data"
[modern-french 81defc8] add new modern data
 1 file changed, 1 insertion(+)

$ echo -e "\n# slang\n\nchenu reluit" >> fr/data

$ git add -u && git commit -m "add french slang"
[modern-french e6defd6] add french slang
 1 file changed, 5 insertions(+)

$ git checkout master

$ echo "wesh" >> fr/data

$ git add -u && git commit -m "add more modern data"
[master 075ce36] add more modern data
 1 file changed, 1 insertion(+)
```

Let's see what happens when we try to merge:

```
$ git merge modern-french
Auto-merging fr/data
CONFLICT (content): Merge conflict in fr/data
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   fr/data

$ git diff
diff --cc fr/data
index cd1a280,5a932d1..0000000
--- a/fr/data
+++ b/fr/data
@@@ -6,4 -6,9 +6,13 @@@ salu
  # modern

  yo
++<<<<<<< HEAD
 +wesh
++=======
+ jourbon
+
+
+ # slang
+
+ chenu reluit
++>>>>>>> modern-french
```

We see the content from the `master` branch materialized in a block delimited by `<<<<<<<` and `=======` and the content of `modern-french` is delimited by `=======` and `>>>>>>>`. By default, git only shows `HEAD` (on the top of a conflict) and `MERGE_HEAD` (on the bottom of a conflict). Visualizing the `ORIG_HEAD` content is a matter of configuration and may be achieved by setting `git config --local merge.conflictstyle diff3`:

```
$ git merge --abort

$ git merge modern-french
Auto-merging fr/data
CONFLICT (content): Merge conflict in fr/data
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git diff
diff --cc fr/data
index cd1a280,5a932d1..0000000
--- a/fr/data
+++ b/fr/data
@@@ -6,4 -6,9 +6,14 @@@ salu
  # modern

  yo
++<<<<<<< HEAD
 +wesh
++||||||| merged common ancestors
++=======
+ jourbon
+
+
+ # slang
+
+ chenu reluit
++>>>>>>> modern-french
```

We now have the full picture:

- there was no content before attempting to merge;
- the `master` branch wants to add "wesh" in the modern section;
- the `modern-french` branch wants to add a "slang" section.

This conflicts is easy to solve by editing the file and keeping both changes and thus having the following

```
$  git diff
 +wesh
+ jourbon
+
+
+ # slang
+
+ chenu reluit
```

To finalize the resolution, we may now stage our changes and commit them

```
$ git add -u && git commit
> Merge branch 'modern-french'
```

```
>
> Conflicts:
>   fr/data
[master b20ab97] Merge branch 'modern-french'
```

and look at our commit tree

```
$ git log --graph --oneline
*   b20ab97 Merge branch 'modern-french'
|\
| * e6defd6 add french slang
| * 81defc8 add new modern data
* | 075ce36 add more modern data
|/
*   62cbf27 Merge branch 'structure-data'
|\
| * 5a40d5a add modern french data
| * dd0109a add 'old' header to french data
|/
* dd0f5d6 Add more french data
* 13d1b4b Create french data file
* 45de2f7 First commit
```

It is important to note that when not sure about the resolution of a conflict:

- reflog keeps previous states so it will always be possible to undo things
- the conflict resolution can be aborted using `git merge --abort`.

Rebasing

Once again, it feels like the graph

```
*   b20ab97 Merge branch 'modern-french'
|\
| * e6defd6 add french slang
| * 81defc8 add new modern data
* | 075ce36 add more modern data
|/
```

could make more sense as

```
|
* yyyyyyy add french slang
* xxxxxxx add new modern data
|
*  075ce36 add more modern data
|
```

Indeed, the purpose of the `modern-french` branch was just to add new content; we could debate about the reason why we would create a new branch as we are continuing previous work. So let's rewind our repository to where it was before merging

```
$ git reset --hard 075ce36

$ git checkout modern-french && git log --graph --oneline
* e6defd6 add french slang
* 81defc8 add new modern data
*   62cbf27 Merge branch 'structure-data'
|\
| * 5a40d5a add modern french data
| * dd0109a add 'old' header to french data
|/
* dd0f5d6 Add more french data
* 13d1b4b Create french data file
* 45de2f7 First commit
```

What we would like to achieve is to change the parent commit of 81defc8 to `master`'s HEAD (075ce36). git allows to move a series of commit (or branch) and replay them on another commit (or branch) with the `git rebase` command.

`git rebase` works by

1. going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto),
2. getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files,
3. resetting the current branch to the same commit as the branch you are rebasing onto,
4. and finally applying each change in turn.

There are 3 main differences with `git merge`:

1. `git rebase` does *not* create any additional commit object
2. when running `git rebase other` from the `current` branch, git will checkout the `other` branch before (re)applying the commits of the `current` branch. Practically this means that the `other` branch will stand for `ORIG_HEAD` and `current` branch for `MERGE_HEAD`
3. git will drop a commit for which the computed diff is now empty.

Let's try to rebase `modern-french` onto `master`:

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: add new modern data
Using index info to reconstruct a base tree...
M   fr/data
Falling back to patching base and 3-way merge...
Auto-merging fr/data
CONFLICT (content): Merge conflict in fr/data
Failed to merge in the changes.
Patch failed at 0001 add new modern data
The copy of the patch that failed is found in:
   /private/tmp/bonjour/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --a

$ git diff
diff --cc fr/data
index cd1a280,89bb164..0000000
--- a/fr/data
+++ b/fr/data
@@@ -6,4 -6,4 +6,9 @@@ salu
  # modern

  yo
++<<<<<<< HEAD
 +wesh
++||||||| merged common ancestors
++=======
+ jourbon
++>>>>>>> add new modern data
```

When applying the commits, git uses the three-way merge algorithm which explains that we have a conflict similar to the one we had with a merge. We edit the conflict as

previously:

```
$ git diff
diff --cc fr/data
index cd1a280,89bb164..0000000
--- a/fr/data
+++ b/fr/data
@@@ -6,4 -6,4 +6,5 @@@ salu
  # modern

  yo
 +wesh
+ jourbon
```

and then stage our modification to continue the rebase process

```
$ git add -u && git rebase --continue
Applying: add new modern data
Applying: add french slang
Using index info to reconstruct a base tree...
M   fr/data
Falling back to patching base and 3-way merge...
Auto-merging fr/data

$ git log --graph --oneline
* b588260 add french slang
* 7a48ea2 add new modern data
* 075ce36 add more modern data
*   62cbf27 Merge branch 'structure-data'
|\
| * 5a40d5a add modern french data
| * dd0109a add 'old' header to french data
|/
* dd0f5d6 Add more french data
* 13d1b4b Create french data file
* 45de2f7 First commit
```

We have thus linearized our commit history by rebasing the `modern-french` branch onto the `master` branch.

As for the `merge` command

- `git reflog` would enable to undo a faulty rebase if needed

- at any conflict resolution, `git rebase --abort` would stop the rebase process and put the branch back at its original state (all rebase information is stored in the `.git/rebase-merge/` folder)
- also, if a commit no longer makes sense due to changes in the upstream branch, `git rebase --skip` will skip the now obsolete commit.

Rewriting history

By rebasing our branch, we have avoided a merge commit. However, we now have two successive commits that seem to bring similar changes

```
* 7a48ea2 add new modern data
* 075ce36 add more modern data
```

Those changes probably deserve to belong to the same commit. `git rebase --interactive` enables to rewrite a set of commits interactively e.g.

- reword a commit message
- squash commits together
- edit commits
- remove commits
- swap commits.

In our case we want to squash commits `075ce36` and `113d5ab`:

```
$ git rebase --interactive 075ce36^
```

will present all child commits that may be rewritten:

```
pick 075ce36 add more modern data
pick 7a48ea2 add new modern data
pick b588260 add french slang

# Rebase 62cbf27..b588260 onto 62cbf27
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
```

```
#  x, exec = run command (the rest of the line) using bash
```

We may notice that commits order is reversed compared to the output of `git log` command and older commits are first in the list.

We just need to change the line

```
pick 7a48ea2 add new modern data
```

into (note that changing anything else that the verb at the beginning of a line will have no effect)

```
fixup 7a48ea2 add new modern data
```

and save the change

```
$ git rebase --interactive 075ce36^
[detached HEAD df55562] add more modern data
 Date: Wed Oct 22 22:45:05 2014 +0200
 1 file changed, 2 insertions(+)
Successfully rebased and updated refs/heads/modern-french.
```

We are done rewriting history and now have a clean commit tree:

```
$ git log --graph --oneline
* cc4b70a add french slang
* df55562 add more modern data
*   62cbf27 Merge branch 'structure-data'
|\
| * 5a40d5a add modern french data
| * dd0109a add 'old' header to french data
|/
* dd0f5d6 Add more french data
* 13d1b4b Create french data file
* 45de2f7 First commit
```

Now the `modern-french` branch should be renamed into `master` which can be done in multiple ways e.g.

```
$ git checkout master && git reset --hard modern-french
HEAD is now at cc4b70a add french slang

$ git branch --delete modern-french
Deleted branch modern-french (was cc4b70a).
```

or

```
$ git branch -D master
Deleted branch master (was 075ce36).

$ git branch --move modern-french master
```

Rebase caveats

Rewriting history can lead to predictible yet unexpected results.

Suppose we create a new file with some lines and a commit for each line.

```
$ git checkout -b rewriting-history

$ echo "one" > dummy && git add dummy && git commit -m "first"
[rewriting-history 9041c15] first
 1 file changed, 1 insertion(+)
 create mode 100644 dummy


$ echo "two" >> dummy && git add dummy && git commit -m "second"
[rewriting-history 50c1ff1] second
 1 file changed, 1 insertion(+)

$ echo "three" >> dummy && git add dummy && git commit -m "third"
[rewriting-history 631a301] third
 1 file changed, 1 insertion(+)
```

Now let's say we want to swap commits

```
$ git rebase --interactive HEAD~3
```

and set

```
pick 9041c15 first
pick 631a301 third
pick 50c1ff1 second
```

This will give the following conflict diff

```
++<<<<<<< HEAD
++||||||| parent of 631a301... third
++two
++=======
+ two
+ three
++>>>>>>> 631a301... third
```

This is fully predictible: each commit stores full file snapshots however we tend to think in incremental delta, simply looking at the diff induced by changes from commit `A` to commit `B` (i.e. `git diff A B`). However when three-way merge is involved (be it for merge or rebase) is the diff with respect to the common ancestor (i.e. `git diff A...B`).

Let's say that we resolved the conflict with the following

```
$ git diff
diff --cc dummy
index 5626abf,4cb29ea..0000000
--- a/dummy
+++ b/dummy
@@@ -1,1 -1,3 +1,2 @@@
  one
 -two
+ three
```

When we continue the rebase, we will again hit a conflict:

```
++<<<<<<< HEAD
 +three
++||||||| parent of 50c1ff1... second
++=======
```

```
+ two
++>>>>>>>> 50c1ff1... second
```

We see that we have moved the issue from the common ancestor to the `MERGE_HEAD`, which in a rebase, is the parent commit and created a conflict cascade.

Remotes

Until now, we have been working locally. As the repository is now clean, we are now ready to publish our work to the world. git servers can be interacted with through the `git remote` command. By default no remote server is defined. We will use a repository declared on GitHub as our remote called `origin`:

```
$ git remote add origin git@github.com:mrchlblng/bonjour.git

$ git remote show origin
* remote origin
  Fetch URL: git@github.com:mrchlblng/bonjour.git
  Push  URL: git@github.com:mrchlblng/bonjour.git
  HEAD branch: (unknown)
```

We see two new verbs, "Fetch" to retrieve modification *from* the remote repository and "Push" to publish our local modification *to* a remote repository. This local/remote binding is callled the refspec. By default, the "Fetch" URL is the same as the "Push" URL but this may be easily configured if needed.

As we did not interact (to fetch or push) with the `origin` remote yet, the `HEAD` branch is unknown.

Push

`git push origin local:remote` will push the `local` branch as the `remote` branch onto the `origin` remote (where `remote` is equal to `local` by default).

git might be configured to shorten the command line to push changes upstream. However care should be taken that this configuration depends a lot on the git version being used (and you might depend on multiple versions when working with distinct servers) so as a safe rule of thumb, always invoke `git push origin local` to push the `local` branch to the `origin` remote. Actually pushing changes is not the action that you perform most of the time and it will keep you out of any embarrasing mistake.

Note that if `local` is empty (i.e. `git push origin :remote`), the `remote` branch will be deleted from the `origin` remote.

We may push our `master` branch to our `origin` remote

```
$ git push origin master
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (8/8), 738 bytes | 0 bytes/s, done.
Total 8 (delta 0), reused 0 (delta 0)
To git@github.com:mrchlblng/bonjour.git
 * [new branch]      master -> master
```

and everyone with an access to the remote can now see our work.

Fetch

Until now, we have been working on our own on the repository. As we created a public repository, some changes may have been pushed to our remote. git does not automatically try to get modification from the remote so it is the user responsibility to make sure his repository is up to date. The command to retrieve remote changes is `git fetch`:

```
$ git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From github.com:mrchlblng/bonjour
 * [new branch]      english    -> origin/english
```

We can see that a new branch `english` has been pushed. We see that locally, git refers to it as `origin/english`. Indeed, git keep remote object references in an eponym namespace

```
$ tree .git/refs/ --matchdirs -P remotes/origin
.git/refs/
├── heads
├── remotes
│   └── origin
│       ├── english
```

```
│       └── master
└── tags
```

It is important to understand that fetching a remote will *not* modify the working tree. Also s a consequence of the remote namespacing, to reference a remote (thus hopefully up to date) branch we should prefix it with the remote name

```
$ git checkout -b english origin/english
Branch english set up to track remote branch english from origin.
Switched to a new branch 'english'
```

Pull

Often times, we fetch a remote to check for update on our working branch meaning that we want to update both git objects and our working tree. This means that we would actually like to fetch changes and apply them. This is what `git pull` does:

1. it fetches changes from the remote defined by the local branch refspec
2. it updates our working tree
    - by merging our local branch with the remote version
    - or by rebasing local branch on the remote branch when invoking `git pull --rebase`.

Force push

Let's say we would like to fix the commit message from the branch we fetched

```
$ git commit --amend -m "add english/american data"
[english ea761bc] add english/american data
 Author: Linux Tor <tor@linux.org>
 Date: Sun Nov 2 20:19:35 2014 +0100
 1 file changed, 20 insertions(+)
 create mode 100644 en/data

$ git push origin english
 ! [rejected]        english -> english (non-fast-forward)
error: failed to push some refs to 'git@github.com:mrchlblng/bonjour.gi
hint: Updates were rejected because the tip of your current branch is b
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for detai
```

By amending, we have just replaced the branch tip commit and whenever a user tries to push a local branch to an existing remote branch, git will check that the remote branch `HEAD` is still reachable in the branch being pushed and will fail if it is unreachable. This may happen in 2 cases:

- local branch is out-of-sync due to changes pushed to the remote; the reflex should be immediately to fetch the remote and update the branch
- history has been rewritten and the sha1 is no longer reachable; we need to force push our local branch to rewrite the remote history:

```
$ git push --force origin english
Counting objects: 1, done.
Writing objects: 100% (1/1), 192 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:mrchlblng/bonjour.git
 + 0ffd492...ea761bc english -> english (forced update)
```

We here see that pushing a rewritten history (`git commit --amend` or `git rebase`) should be done with care and we will discuss workflows in the next section.

Branches and remotes: tl;dr

- branches are simply references to commits
- `HEAD` references the tip commit for current branch
- `git merge` (and most "merging-like" commands) relies on three-way merge by default
- `git rebase` enables to rewrite history
  - `git rebase foo` will move current branch commits onto `foo` branch (note that in case of conflict `ours` and `theirs` can feel inverted as `git rebase foo` checkouts the `foo` branch under the hood)
  - `git rebase --interactive sha1^` will allow to edit/squash/remove/swap all commits from `sha1` to `HEAD`
  - rebasing rewrites history and creates new commit objects
- comparing branches for merge/rebase should be done with `git diff A...B` to take into account the common ancestor
- retrieving remote updates is the user responsibility
- merge/rebase commands should usually reference branches from remote namespace
- never force push without
  - having checked if changes were pushed to the remote
  - fully specifying the remote/branch (to avoid force pushing wrong

branches)

Good practices

Workflow

Branching flow

Creating branches with git is very cheap and should therefore be used without fear. The question is then: how should the branches relate to one another?

There is no single answer to that question. Mostly because the answer depends on type nature of the project: a web app runs a single ever up-to-date version when a desktop app may have multiple versions supported at a given time. The former may however be seen as a simple particular case of the latter.

> *Don't merge upstream code at random points.*
> *Don't merge downstream code at random points either.*
>
> — Linus Torvald

Most worflows maintain different contexts with careful synchronization points and the typical synchronization for a web app will look like:

- the `master` branch should always be stable and reflect the code running in production
- the `develop` branch hosts developments between 2 releases
- new features should branch off/get integrated in the `develop` branch
- hotfixes should branch off `master` and be integrated in both `master` and `develop` branches.

The question now becomes: how should branches be synchronized?

Merge or rebase?

The "merge or rebase" question is one of the most debated one.

The arguments mostly fall back to

- `git merge` keeps original context but create a clumsy history that can be difficult to read and makes `git bisect` more difficult to use
- `git rebase` ends up in a linear history but rewrites history by computing incremental patches and thus modifies the original authored commits.

It used to be about only merge or only rebase but usage evolving with time. Nowadays, rewriting a "private" branch is seen as a cleanup and therefore mostly considered a good practice. Private does not necesarily means that the branch was not pushed on a remote yet; it rather means that you are mostly working on the branch alone. You may then push *your* branch on a remote, either to keep a backup or to help discuss a matter with team mates. Hence privacy should be seen as responsibility:

- a private branch is your own responsibility and its history may be altered to meet the project quality standards
- a public branch is a collective responsibility and thus history should be taken with care as changing it may offend people.

The question is now: how should a private branch be integrated in the public history? There is no really better solution; however it seems that people are tending to use a merge in order to keep an identifiable view of the former private branch. Also some tools like GitHub make opiniated choice and when using a Pull Request workflow with the service interface, branches may only be merged.

To summarize what seems to become the dominant branch workflow in git:

> *Rebases are how changes should pass from the top of hierarchy downwards and merges are how they flow back upwards.*

What's a good commit?

Atomicity

A good commit is about atomicity. Not unlike Unix philosophy, a commit should do one thing and on thing well. The reason is that it will allow to undo a precise change very easilyit will provide a kind of documentation about how the system works in its whole. A commit message where the conjunction 'and' appears is probably not very atomic. Nonetheless, doing only one thing does *not* necessarily mean that a good commit should impact only a single file. Typically making a change in the code base will require a test suite update.

Hiding sausage making

However, unlike the maxim, the destination is more important than the journey when it comes to making good commits. It means that even though you took dead ends (like thinking the cause of a bug is X when X is only a consequence of Y) when trying to solve a hard bug, it is very valuable to keep those peregrination in the project history. It first makes it harder to review the changes in the first place and once in the

main trunk it will be harder to understand what the actual fix is. git is flexible and allows different workflows:

- keeping all changes unstaged and crafting the commits once everything is implemented (using `git add --patch`)
- creating commits along the way and finally rewriting the history to clean things up. Note that it is sometimes easier to reset to a state where all changes are unstaged/uncommitted (see Reset).

Commented hunks of code

Last but not least, a good commit should not contain code that has been commented:

- it will only get people to be confused about the commented hunks
- it goes against the main usage of a vcs i.e. history control that allow to retrieve past hunks that are no longer in the trunk.

What's a good commit message?

Formatting

The structure of a commit message is important. It shoud look like:

```
Short commit summary (below 50 chars)

Detailed description of the changes introduced by the commit
(using at most 72 chars wide columns and markdown style)
```

Following this format will produce readable `git log` outputs and display nicely with git tools and services.

Content

The commit message is like a (technical) book cover: it should give a good insight of what the commit is actually about. The description is like the backcover, providing more in-depth detail about the commit itself (and possibly listing dead ends encountered). It should answer the following questions:

- Why is this change necessary?
- How does it address the issue?
- What side effects does this change have?

Documentation

As every line of code in repository comes from a unique commit, writing good commit messages will provide a documentation covering all the code:

- `git whatchanged path/to/file` lists all commits that changed a particular file
- `git blame path/to/file` shows what revision and author last modified each line of a file
- `git log --topo-order --graph -u -L$line,$line:path/to/file` will show all commits modifying the line `$line` of a file.

The advantage of this is that messages cannot go out-of-sync. Whenever a line will be updated, the associated commit message will provide an up-to-date version. And those 3 commands provide fine documentation granularity from a project level (`git whatchanged`), to a file level (`git blame`) and finally a line level (`git log --topo-order`).

Going further

Options

Here is a short list of useful options that applies to many git commands

- `--patch`: work on hunks instead of full files
- `--cached`: apply command to the staging area instead of repository objects
- `--stat`: display a diffstat only i.e. for each file modified it shows the number of deletions and additions
- `git [command] [options] -- file/to/path`: using `-- file/to/path/` will only apply `git [command] [options]` to `file/to/path`. Note that a regexp might be used e.g. `-- '*.[ch]'` will perform the command on all C files and headers.
- some commands accept date/time filters such as `--since`, `--until`, `--before` or `--after`; the "approxidate" parser will accept absolute date (e.g. `"2014-01-01"` or `"Jan 01 10:00:00 2014 +01"`) or relative ones (e.g. `"3.weeks.ago"` or `"last monday"`).
- commands that do not directly accept date/time filters can still be used with dates through the `branch@{approxidate}` construct e.g. `git diff --stat master@{1.week.ago} master` will display a diffstat of changes committed on the `master` during the last week.

Searching

We have already seen commands to compare branches and list commit messages. git allows to quickly search through the repository too.

- searching content

    - `git grep -e 'pattern' [revision]`: search for `'pattern'` in a given revision (`HEAD` by default) full content
    - `git rev-list --all | xargs git grep -e 'pattern'`: search for `'pattern'` in all reachable commits
- searching diff:
    - `git log -S"pattern"`: search for commits having `"pattern"` in their diff (be it addition or deletion)
- searching commit messages:
    - `git log --grep='pattern'`: search for `'pattern'` in commit messages in the current branch history
    - `git log --all --grep='pattern'`: search for `'pattern'` in commit messages of all branches
- searching authors:
    - `git log --author='pattern'`: search for commit made by an author matching `'pattern'`
    - `git log --committer='pattern'`: search for commits committed by a committer matching `'pattern'`

Reset

As we've seen at the beginning, git involves both a local file tree and a commit tree. When one needs to reset something in git, it should first analyze what exactly should be reset:

- local file tree only: `git checkout sha1 -- path/to/content`
- git commit tree only: `git reset sha1`
- both local file tree and git commit tree: `git reset --hard sha1`

Cherry pick

It is sometimes handy to be able to apply a single commit from another branch e.g. to report a patch. `git cherry-pick` will do just that and apply the `sha1` commit content in a new commit on the current branch. The commit object will be distinct. However this is not considered as rewriting history since the commit did previsouly not exist in the branch.

Stash

It sometimes happen that you need to quickly change context (e.g. to fix some bug in production). To avoid creating a dummy commit with all current staged and unstaged diff, you may use `git stash`. This will push a new stash in the stash stack.

When stashing, you should always save a message (`git stash -m "..."`) to keep some context to what you were previously doing. The risk is that you might

completely forget that you stashed some modification.

Bisect

When some bug or regression is found in a repository, we used to perform dichotomy thourgh history to find when *it* was introduced. `git bisect` automates the process.

We first need to start the session by setting the interval that should be tested, flagging the last known commit that is known to not have the bug as good and a bad commit

```
$ git bisect start

$ git bisect bad 456xyz # by default HEAD is assumed

$ git bisect good abc123 # last commit known to be bug-free
```

git then iterates over the range of commits using binary search and will wait for a good/bad flag for each checkouted commit. When all revisions have been bisected, the faulty commit is stored in `refs/bisect/bad`.

To automate things further, an executable script or a command checking wether a revision is good or bad may be supplied by

```
$ git bisect run command
```

Environment

Command line completion

One of the most helpful tool you will need is git bash completion. This enables tab-completion on most git commands. Downloading the script and sourcing it in its bash profile is all that is needed.

Configuration

Local/global/system

git configuration may be done at three levels (listed by order of *descending* precedence)

- `--local`ly i.e. at a repository level in `repo_path/.git/config`
- `--global`ly for a given user in `$HOME/.gitconfig`; note here that if the `$HOME` environment variable changes, you may experience behaviors that you did not expect
- `--system`-wide in `/etc/gitconfig`

by invoking

```
git config [--level] --(set|add) section[.subsection]=value
```

Most of the time configuration is set at the `--global` level but it really depends on your setup.

All configurations can either be done through the `git config` command or by directly editing the `gitconfig` file if you know what you are doing.

Basics

We have already seen some configuration as git requires a user to be properly used:

```
$ git config --global --set user.name="My Name"

$ git config --global --set user.email="me@mail.org"
```

We have also been changing the merge conflict output to show the `ORIG_HEAD`:

```
$ git config --global merge.conflictstyle diff3
```

One of the most useful configuration is colors that will make reading any `git diff` ouputs easier to read (note that this is the default for git≥1.8.4):

```
$ git config --global ui.color=auto
```

Finally, git may automatically correct mistyped commands when the error can be unambiguously fixed in the specified number of deci-seconds (0.1 second):

```
$ git config --global --set help.autocorrect=5
```

Aliases

Aliases offer a way to

- use shorter names for commands frequently typed; a common example is to alias `checkout` with `co`

```
$ git config --global alias.co checkout
```

- introduce a personnalized command using constant arguments; a typical usage is a custom display of commit history:

```
$ git config --global alias.hist "log --graph --pretty=format:'%Cred%
```

- invoke custom commands with possible arguments (git≥1.5.3) and possibly piping with non-git commands
  - listing recent commits using the bash `head` command (arguments will apply on the `head` command i.e. `git head -5` will list the last 5 commits)
  - fixuping a commit by commiting and rebasing automatically

```
$ git config--global alias.head "git log --oneline --pretty=format:'%Cr

$ git config --global alias.fixup "!sh -c '(git diff-files --quiet || (
```

Listing all defined aliases may be done with the following command

```
$ git config --get-regexp ^alias\. | sed -e s/^alias\.//
```

and/or defined as an alias!

Hooks

git hooks offer the possibility to add custom behaviors on top of some git actions:

- user hooks:

    - `applypatch-msg`
    - `pre-applypatch`
    - `post-applypatch`
    - `pre-commit`
    - `prepare-commit-msg`
    - `commit-msg`
    - `post-commit`
    - `pre-rebase`
    - `post-checkout`
    - `post-merge`
    - `pre-push`
- server hooks:
    - `pre-receive`
    - `update`
    - `post-receive`
    - `post-update`
    - `pre-auto-gc`
    - `post-rewrite`

Hooks have to be *executable* and should be placed in the `.git/hooks` folder. Hooks thus can *not* be enforced within a repository. Some typical usage for hooks is:

- check commit message format (see e.g. a hook for pivotal enforcing branch name convention in the `pre-commit` hook and referencing pivotal story id in the `prepare-commit-msg` hook)
- automatically run test suite locally; depending on the project policy, this could be done for every commit or simply before pushing code to a remote
- automatically triggering project build/deployment when the remote is updated.

Custom command-line prompt

When interacting from the command-line, having a prompt displaying information such as the branch or brief status helps to improve productivity by avoiding unnecessary intermediate commands. Implementing a robust bash customization is cumbersome and there already exist lots of customizable projects:

- powerline
- bash powerline
- bash git prompt
- git prompt
- zsh git prompt

Protocols

Interaction with a remote rely on a transfer protocol. Depending on the server configuration, the following protocols may be available:

- git: fast but with no authentication
- ssh: secured & efficient; requires to set up keys
- http(s): efficient; authentication might be more cumbersome than with ssh as no session is kept and credentials might have to be provided multiple times.

Other technical considerations such as corporate firewall preventing traffic through some ports might make the https protocol useful in most cases.

Major commercial git hosting solutions support at least ssh and https protocols.

GUIs and plugins

Here is a list of higher level GUIs or plugins:

- GitHub GUI for mac/windows
- sourcetree for mac/windows
- tortoisegit for windows
- git tower for mac
- fugitive, vim-airline for vim
- magit for emacs (tutorial)
- sublime-text git for sublime-text

Jargon

upstream/downstream

This definition is relative and depends on how data flows:

- downstream expresses that content is being fetched
- upstream stands for the original source of data.

bare and non-bare repository

A non-bare repository is what we have been manipulating so far: a repository containing both `.git` folder and a local file tree.

A bare repository has no local file tree but just the `.git` folder. It is typically used for host host git repositories on a server and is instanciated using `git init --bare`.

fork

A fork is a copy of a repository.

pull request

A pull request is a request to merge a branch (typically hosted on a fork) into an other branch.

porcelain/plumbing

This terminology refers to toilets:

- porcelain is the material from which toilets are made; this stands for git high level commands, the one the user actually see and interact with
- plumbing is what happens behind the scene and what carries "stuff"; plumbing are the low level commands to which porcelain commands transfer data.

gist

A gist is a lightweight repository typically versioning a single or a small number of files.

hunk

A hunk is a section of diff e.g.

```
@@ -1,1 -1,3 +1,2 @@
  one
 -two
+ three
```

- the triple `@` indicates where are using `diff3`
- the three pairs of numbers indicates common ancestor, "from file" and "to file" lines span
- a line span pair represent `first line, number of lines` for the diff patch.

References

- https://try.github.io/levels/1/challenges/1
- http://git-scm.com/book/en/v2
- https://speakerdeck.com/schacon/introduction-to-git
- http://onlywei.github.io/explain-git-with-d3/
- http://wildlyinaccurate.com/a-hackers-guide-to-git
- http://eagain.net/articles/git-for-computer-scientists/
- http://justinhileman.info/article/git-pretty/