git
Basic concepts and usage

Francis Colas
What is git?

git concepts

Standard commands

Feature branch workflow

Conclusion
What is git?
Revision control

Source code:
- everything is source code,
- usually written in an iterative process,
- sometimes by several persons;

Revision control:
- management of versions of documents,
- tracking of changes,
- restoration of previous state,
- similar concepts to a database;

Some generic concepts:
- atomicity: ensuring consistency of state,
- distributed/centralized: communication model of changes,
- branches and tags: development direction and milestones,
- locking/merging: handling of concurrent changes.
Timeline:

before hand-numbered or date-stamped copies,

1990 CVS: centralized and non-atomic,

2000 Subversion: centralized and atomic,

2000s GNU Arch, Monotone, Darcs: decentralized and atomic,

2005 Linux cannot use BitKeeper anymore,

2005 answer: mercurial and git.
02

git concepts
General characteristics

Repositories:
- monolithic and self-sufficient,
- everybody has a full clone,
- in particular: no need for a connexion to commit.

Concepts:
- **git** handles files (no empty directory),
- central objects are **commits**:
  - set of file changes,
  - applied on top of a given [list of] commit[s],
- **tags** and **branches** just point to a commit,
- **remotes** are pointers to other clones (**github**, **gitlab**, other clone somewhere...).

---

1actually, they are more a snapshot of the state
Staging area

Three (conceptual) places:
▶ working directory: the files in the filesystem,
▶ staging area: the files as how they’ll be committed,
▶ git repository: committed files.

What it means:
▶ changes happen in working directory,
▶ need for preparation of the commit using the staging area,
▶ actual commit is a frozen snapshot of the staging area.
Standard commands
Basic commands

Initialization:

- `git init <name>`: initialize an empty repository,
- `git clone <url>`: clone a repository,
Basic commands

Initialization:
▶ `git init <name>`: initialize an empty repository,
▶ `git clone <url>`: clone a repository,

Manipulating the staging area:
▶ `git add <files>`: add current state of files to next commit,
▶ `git rm <files>`: delete files,
▶ `git mv <oldname> <newname>`: move/rename a file.
Basic commands

Initialization:

▶ `git init <name>`: initialize an empty repository,
▶ `git clone <url>`: clone a repository,

Manipulating the staging area:

▶ `git add <files>`: add current state of files to next commit,
▶ `git rm <files>`: delete files,
▶ `git mv <oldname> <newname>`: move/rename a file.

Looking at status:

▶ `git status`: list state of working copy with respect to last commit.
Basic commands

Initialization:
▶ `git init <name>`: initialize an empty repository,
▶ `git clone <url>`: clone a repository,

Manipulating the staging area:
▶ `git add <files>`: add current state of files to next commit,
▶ `git rm <files>`: delete files,
▶ `git mv <oldname> <newname>`: move/rename a file.

Looking at status:
▶ `git status`: list state of working copy with respect to last commit.

Commit and transmission:
▶ `git commit -m "commit message"`: record changes into a commit,
▶ `git pull [<url|remote>]`: get changes and merge them,
▶ `git push [<url|remote>]`: sends commit to remote/url.
Branching commands

Handling branches (because it’s cool):

▶ `git branch <branch>`: create branch,
▶ `git branch -d <branch>`: delete a branch,
▶ `git checkout <branch>`: change to a different branch,
▶ `git checkout -b <branch>`: create and change to branch,
▶ `git push -u <remote> <branch>`: create and push branch to remote,
▶ `git merge <branch>`: merge changes of the given branch into the current one (used by pull).
Additional information

Temporary changes:

▶ **git stash**: save current changes aside,
▶ **git stash pop**: restore saved changes.

Seeing branches and commits:

▶ **gitk [--all]**: good tool when lost in branches,
▶ **git gui**: makes it easy to pick individual changes in a file.

Ignore file:

▶ **.gitignore**: list of patterns of filenames that will be ignored.

Fixing the last commit before sending it:

▶ **git commit --amend**: replaces commit with current staging area.
Feature branch workflow
Feature branch workflow

General idea:

- features get developed in individual branches,
- when ready: merge into master branch,
Feature branch workflow

General idea:
▶ features get developed in individual branches,
▶ when ready: merge into master branch,

Advantages:
▶ master always consistent and (hopefully) working,
▶ less merges while concurrent development,
▶ easier to manage,
▶ interesting commit tree. :-)
Feature branch workflow

General idea:
>
features get developed in individual branches,
>
when ready: merge into master branch,

Advantages:
>
master always consistent and (hopefully) working,
>
less merges while concurrent development,
>
easier to manage,
>
interesting commit tree. :-)  

Merging:
>
open merge request,
>
have somebody review changes,
>
ideally review and test code before accepting merge.
Practically

In practice:

- `git checkout -b new_feature`
- `code, test, code more, test even more...`
- `git status`
- `git add <files>`
- `git status`
- `git commit -m "new feature"`
- `git pull`
- `git push -u origin new_feature`
- `merge request`
- `get it accepted`
- `git checkout master`
- `git pull`
- `git branch -d new_feature`
- `git remote prune origin`
From the command line:

▶ git stash
▶ git checkout master
▶ git checkout -b merging_new_feature
▶ git fetch
▶ git merge new_feature
▶ code review
▶ test (compilation, unit tests, etc.)
▶ git checkout master
▶ git merge new_feature
▶ git push
▶ git checkout your_branch
▶ git stash pop
Conclusion
Conclusion

**git:**
- not complex to use,
- good for one or many people,
- good for connected or offline work;

**Feature branch model:**
- convenient in big projects,
- state-of-the-art use of git.
Thanks for your attention.