Tutoriel Git

Érik Martin-Dorel

Version 5.4.1 - 20 Novembre 2019

Table des matières

1 Introduction

Git est un système de contrôle de version de code source, initialement créé par Linus Torvalds en 2005 pour gérer le développement du noyau Linux. C'est un système *décentralisé*, c'est-à-dire que chaque utilisateur travaille sur une copie *locale* du dépôt du projet, sans avoir besoin de connexion réseau pour faire ses *commits*. Contrairement à **Subversion** qui génère un numéro incrémenté à chaque commit, avec **Git** l'historique des commits n'est pas forcément linéaire, et chacun d'entre eux est repéré par une signature *SHA-1* (40 caractères hexadécimaux, les 7 premiers étant en pratique suffisants pour identifier le commit). Enfin, c'est un logiciel libre distribué sous licence GNU GPL v2.

2 Installation de Git

2.1 Installation

Si vous êtes auto-administrateur de votre poste :
— Pour installer **Git** sous GNU/Linux Ubuntu :
`$ sudo apt-get install git gitk git-gui git-doc`
— Pour installer **Git** sous Windows, vous pouvez consulter le site : `https://git-for-windows.github.io/` et/ou cet article de blog : `http://itekblog.com/beginners-guide-to-gitgithub-with-windows/#Installing_Git`

Vous pouvez exécuter la commande `git` dans un terminal pour vérifier que **Git** est bien installé, ce qui devrait afficher le texte d'aide suivant :

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
```

```
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty Git repository or reinitialise an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

2.2 Configuration obligatoire

Chaque commit **Git** comporte le nom et l'adresse e-mail de son auteur, il est donc indispensable de définir ces informations au préalable :

```
$ git config --global user.name "Prénom Nom"
$ git config --global user.email prenom.nom@irit.fr
$ git config --global -l  # pour vérifier
```

2.3 Configuration fortement recommandée

Pour une utilisation plus aisée de « `git push` » :

```
$ git config --global push.default upstream
```

Si emacs (ou vim, nano, gedit, ...) est votre éditeur de texte préféré :

```
$ git config --global core.editor emacs
```

ou alors si c'est Sublime Text :

```
$ git config --global core.editor "subl -n -w"
```

2.4 Configuration facultative

Pour colorer la sortie de « `git diff` » et des commandes similaires :

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

Pour que la sortie de « `git pull -r` » soit aussi détaillée que « `git pull` » :

```
$ git config --global rebase.stat true
```

Pour avoir un "shortlog" des commits dans le message par défaut des "merge" :

```
$ git config --global merge.log true
```

Pour vérifier la configuration :

```
$ git config --global -l
```

3 Introduction aux concepts de Git

Dans un projet **Git**, il y a 3 espaces différents :
- l'espace de travail (ou *working directory*) : l'ensemble des fichiers pouvant être directement édités par le développeur
- l'aire d'embarquement (ou *staging area* ou *index*) : l'ensemble des fichiers créés/modifiés/supprimés qui ont été sélectionnés en vue d'un commit
- le dépôt (ou *repository*) : l'ensemble des commits constituant l'historique du projet (stocké dans le sous-dossier `.git`).

Une particularité de **Git** : on ne peut pas commiter un dossier vide. En effet, **Git** effectue uniquement le suivi de version des fichiers et de leur contenu, mais pas des dossiers. Et lorsqu'on ajoute un dossier (non vide) dans l'aire d'embarquement, cela revient juste à y ajouter l'ensemble des fichiers de ce dossier.

Contrairement aux systèmes de gestion de version centralisés comme **Subversion**, avec **Git** chaque dépôt contient tout l'historique du projet. Ainsi, les communications qui peuvent intervenir entre les différents dépôts d'un même projet sont essentiellement des opérations de synchronisation (`push` pour envoyer, `pull` (= `fetch` + `merge`) pour recevoir).

Dans la terminologie **Git**, les dépôts distants pouvant intervenir dans ces opérations de `push` et `pull` s'appellent des *remotes*. Un *remote* est un alias vers un dépôt distant, constitué d'un identificateur (souvent `origin`) et d'un URL.

L'un des atouts de **Git** est la grande facilité avec laquelle on peut créer, renommer ou supprimer des branches (comme les branches sont juste des "pointeurs" vers un commit particulier, leur création ne coûte rien). On parlera de branche locale (resp. de branche distante) si la branche a été créée sur le dépôt local (resp. sur un *remote*).

Enfin, il est possible de créer des *tags* pour marquer un commit comme étant très important, ou correspondant à une nouvelle *release* du projet. Voir par exemple `http://git.github.io/git-reference/branching/#tag`

4 Création d'un nouveau dépôt Git

4.1 Création d'un dépôt local à partir de rien

(Cette procédure serait surtout utile si vous souhaitez créer un dépôt personnel, pour garder localement l'historique de certains fichiers dont vous êtes le seul développeur. Dans le cas d'un projet de recherche collaborative, voir la section suivante.)

— Création d'un dossier qui contiendra le nouveau dépôt :

```
$ mkdir -p ~/forge/git/nouveau-depot
$ cd ~/forge/git/nouveau-depot
```

— Création d'un nouveau dépôt **Git**, stocké dans le sous-dossier `.git` du dossier courant :

```
$ git init
```

4.2 Clonage d'un dépôt distant

4.2.1 Clonage sans authentification (dépôt GitHub public)

Il suffit d'ouvrir un terminal (ou **Git-Bash** sous Windows), d'aller dans un dossier dédié, par exemple en faisant :

```
$ mkdir -p ~/forge/git/ && cd ~/forge/git/
```

et d'utiliser la commande « `git clone` » :

```
$ git clone https://github.com/ocaml/ocaml.git
```

4.2.2 Clonage avec authentification SSH (dépôt GitHub privé)

Dans cette section on suppose que l'utilisateur `username` héberge sur GitHub un dépôt privé `repo` destiné à un développement collaboratif.

Création d'un compte GitHub Chaque membre du projet doit déjà avoir un compte GitHub. Sinon, s'inscrire sur `https://github.com/join` et choisir son nom d'utilisateur et l'offre gratuite (*free plan*). Puis, communiquer son nom d'utilisateur au responsable du projet.

Création d'un couple de clés SSH Chaque membre du projet doit avoir une **clé privée SSH**. Sous Windows il est possible d'en créer une directement avec **Git GUI** (voir par exemple cet article de blog [1]). Autrement, pour créer une clé SSH en ligne de commande il suffit d'utiliser la commande suivante, qui vous demandera de choisir un mot de passe (facultatif, mais conseillé) pour protéger votre clé privée :

```
$ ssh-keygen -t rsa -b 4096
```

Ensuite, il faut associer sa **clé publique SSH** avec son compte GitHub, en allant sur `https://github.com/settings/keys`, en cliquant sur « *New SSH key* » et en collant sa clé publique SSH (qui devrait se trouver dans le fichier `~/.ssh/id_rsa.pub`). Pour vérifier la connexion SSH, on peut faire :

```
$ ssh -T git@github.com
```

Si cela ne marche pas, c'est peut-être dû à un problème de déverrouillage de votre clé privée. Dans ce cas vous pouvez essayer les commandes suivantes :

— Sous GNU/Linux :

1. `http://code.tutsplus.com/tutorials/git-on-windows-for-newbs--net-25847`

```
$ ssh-add
$ ssh -T git@github.com
```

— Sous Windows avec **Git-Bash** :

```
$ eval $(ssh-agent -s)
$ ssh-add
$ ssh -T git@github.com
```

Clonage du dépôt privé GitHub Pour cloner le dépôt privé `repo` de l'utilisateur `username`, il suffit d'utiliser la commande « `git clone` » :

```
$ mkdir -p ~/forge/git/ && cd ~/forge/git/
$ git clone git@github.com:username/repo.git
```

Remarque 1 : L'un des avantages du protocole SSH par rapport à HTTPS (préfixe `https://`) pour accéder à vos dépôts privés **Git** est que grâce à l'utilitaire `ssh-add`, il suffit de déverrouiller sa clé privée une fois par session, plutôt que de devoir retaper son mot de passe à chaque fois pour chaque communication HTTPS avec le serveur. En revanche, le protocole HTTPS serait davantage pratique pour des dépôts **Git** accessibles sans authentification (comme par exemple des projets publics hébergés sur GitHub, auxquels vous n'avez qu'un accès en lecture).

5 Utilisation minimale de Git

5.1 Interface standard (ligne de commande)

Il y a 7 commandes de base à connaître absolument :

— `git status`
— `git add`
— `git commit`
— `git mv`
— `git rm`
— `git pull`
— `git push`

et 2 autres commandes très utiles en pratique :

— `git reset`
— `gitk`

À noter : il n'existe pas de commande « `git cp` » (donc si jamais on doit copier un fichier il suffit d'utiliser « `cp` » et « `git add` »).

5.1.1 Les 7 commandes de bases

— Pour afficher un résumé de l'état du dépôt :
`$ git status`

— Pour ajouter des fichiers dans l'aire d'embarquement (ou index) :
`$ git add Nom des fichiers à commiter`
ou bien pour y ajouter/supprimer automatiquement tous les fichiers ayant été créés/édités/supprimés :
`$ git add -v -A .`
où l'option « `-v` » (verbose) demande à **Git** de résumer ce qu'il a fait. (Si trop de fichiers ont été ajoutés : voir ci-dessous la commande « `git reset` »)

— Pour ajouter des fichiers dans l'historique :
`$ git commit -m "Résumé du commit"`
ou juste
`$ git commit`
pour ouvrir l'éditeur par défaut des messages de commit.
Remarque 2 : seuls les fichiers précédemment ajoutés dans l'aire d'embarquement seront commités. Il ne faut donc pas oublier d'utiliser « `git add` » avant « `git commit` » (à moins de suivre la *Remarque 3*)
Remarque 3 : si les fichiers devant être commités sont tous déjà connus de **Git**, on peut combiner le « `git add` » et le « `git commit` » en une seule commande en utilisant l'option « `-a` » :
`$ git commit -a -m "Edit files."`

— Pour déplacer un fichier déjà présent dans l'historique :
`$ git mv nom_du_fichier nouveau_nom_du_fichier`
`$ git commit -m "Rename file."`

— Pour supprimer un fichier déjà présent dans l'historique :
`$ git rm nom_du_fichier`
`$ git commit -m "Remove nom_du_fichier."`

— Pour récupérer les changements faits par les autres développeurs sur la branche `master` du remote `origin` et les fusionner *avec la branche actuelle* :
`$ git pull origin master`
ou plus simplement :
`$ git pull`
si la branche actuelle a déjà été associée à `origin/master`.
Remarque 4 : si vous avez suivi le début de ce tutoriel et fait un « `git clone` », `origin/master` pointera déjà vers le dépôt **Git** distant.
Remarque 5 : pour être complet au sujet de « `git pull` », la commande « `git pull origin master` » correspond à la succession de 2

commandes : « `git fetch origin && git merge origin/master` ». (C'est-à-dire, une récupération des commits disponibles sur le dépôt distant, suivie d'une fusion de la branche `origin/master` avec la branche actuelle.)

— Pour envoyer les changements de la branche `master` sur (la branche portant le même nom sur) le dépôt distant :
```
$ git push origin master
```
ou plus simplement :
```
$ git push
```
si `master` est la branche actuellement sélectionnée. Pour plus de précisions sur les branches et `git push`, voir la section 6.1.

5.1.2 Les 2 commandes à connaître aussi

— Si jamais un « `git add -A .` » a ajouté trop de fichiers, pour ôter de l'index certains fichiers que l'on ne veut pas commiter, faire :
```
$ git reset -- nom_du_fichier_en_trop
```
et idéalement, créer un fichier `.gitignore` à la racine du dépôt pour ignorer systématiquement ces fichiers (voir la section Recommandations ci-dessous).
Cette commande « `git reset` » a d'autres utilisations possibles avec une syntaxe particulière, que nous ne détaillerons pas ici (cf. la documentation : « `git help reset` »)

— Enfin, la commande
```
$ gitk
```
peut être utilisée à tout moment pour afficher l'historique des commits de la branche actuelle, ainsi qu'un résumé des changements sur le point d'être commités. (Pour quitter, faire `Ctrl+Q`.)
Et pour afficher l'historique de toutes les branches, faire :
```
$ gitk --all
```

5.2 Interface graphique

On peut aussi travailler avec une interface graphique en faisant :

```
$ git gui
```

Voici 3 articles de blog illustrant (l'installation et) l'utilisation de `git gui` :

— `http://itekblog.com/beginners-guide-to-gitgithub-with-windows/`
— `http://www.thegeekstuff.com/2012/02/git-for-windows/`
— `http://code.tutsplus.com/tutorials/git-on-windows-for-newbs--net-25847` (qui explique également la création des clés SSH avec `git gui`)

6 Utilisation avancée de Git

6.1 Création d'une branche

— Pour expérimenter en toute tranquillité une nouvelle fonctionnalité qui pourrait briser la stabilité de la branche principale, vous pouvez créer très facilement une nouvelle branche et basculer vers celle-ci en faisant :
`$ git checkout -b experiment-new-feature`
ce qui équivaut à la succession de commandes :
```
$ git branch experiment-new-feature && \
  git checkout experiment-new-feature
```
Comme en **Git** les branches sont juste des "pointeurs" vers un commit particulier, leur création ne coûte rien (contrairement à d'autres systèmes comme **Subversion** par exemple).

— On peut afficher la liste des branches locales en faisant :
`$ git branch`
ou
`$ git branch -a`
pour afficher également les branches distantes.

— Après avoir travaillé dans une nouvelle branche `experiment-new-feature`, on peut utiliser les commandes « `git add` » et « `git commit` » comme d'habitude, puis pousser cette branche sur le dépot distant en faisant :
`$ git push -u origin experiment-new-feature`
ou de manière plus concise :
`$ git push -u origin HEAD`
ce qui suppose que `origin` est bien le nom qui identifie le dépôt distant (c'est toujours le cas si le dépôt a été initalement *cloné*). Sinon, vous pouvez afficher cet identificateur en faisant :
`$ git remote -v`
Remarque 6 : l'option « `-u` » ci-dessus permet de mémoriser l'association de la branche distante `origin/experiment-new-feature` à la branche locale `experiment-new-feature`, afin de pouvoir écrire seulement « `git push` » par la suite pour pousser la branche, au lieu de « `git push origin experiment-new-feature` ».
Remarque 7 : « `git push -u origin experiment-new-feature` » *ne tient pas compte de la branche actuellement ouverte* (cela pousse la branche locale `experiment-new-feature` vers le dépôt `origin`), tandis que « `git push -u origin HEAD` » *pousse la branche actuellement ouverte* vers le dépôt `origin`. (Le nom de la branche distante sera par défaut le même nom que la branche locale, ici,

`origin/experiment-new-feature`, mais il est possible de personnaliser ceci : « `git push -u origin feat:staging` » pousserait la branche `feat` vers la branche `origin/staging`).

6.2 Changement de branche

— Pour afficher la branche où l'on se trouve :
```
$ git branch
```
— Pour changer vers une branche existante :
```
$ git checkout nom-de-la-branche
```

6.3 Fusion d'une branche

— Pour fusionner les modifications d'une branche (nommée par exemple `experiment-new-feature`) avec la branche principale (par défaut la branche `master`), il suffit de faire :
```
$ git checkout master
$ git merge experiment-new-feature
```
S'il n'y a pas de conflit, le commit correspondant au merge est créé.
S'il y a un conflit, le résoudre puis commiter le résultat :
```
$ git diff
... Éditer le fichier concerné ...
$ git add nom_du_fichier_édité
$ git commit
```
— Pour supprimer la branche locale devenue superflue, on peut faire :
```
$ git branch -d experiment-new-feature
```

6.4 Amender un ou plusieurs commits

Les deux techniques décrites ci-dessous ne sont pas possibles si les commits ont déjà été publiés sur le serveur avec « `git push` ». Dans ce cas, il suffit de trouver le numéro du commit (en utilisant « `gitk` » voire « `git log -p` ») puis d'utiliser la commande « `git revert identifiant_du_commit` » pour créer un nouveau commit qui annule le commit spécifié.

6.4.1 `git commit --amend`

Si le *dernier commit* doit être *légèrement rectifié* vis-à-vis du message de commit ou d'une des lignes de code sauvegardées, on peut effectuer la modification du code souhaitée, puis exécuter :

```
$ git add nom_du_fichier_rectifié
$ git commit --amend
```

6.4.2 `git rebase --interactive`

Il est possible de réordonner ou de fusionner plusieurs commits pour avoir un "historique plus propre", à condition que ces commits n'aient pas déjà été publiés/poussés avec « `git push` ». Il suffit de faire :

```
$ git rebase -i
```

puis de suivre les instructions qui s'affichent :
- remplacer `pick` par `reword` pour éditer un message de commit
- ... par `squash` pour fusionner avec le commit du dessus
- ... par `fixup` pour fusionner avec le commit du dessus, mais ignorer le message du commit courant.

Enfin, sauvegarder le fichier (normalement nommé `git-rebase-todo`) et quitter votre éditeur.

7 Recommandations

7.1 Création d'un fichier .gitignore

En général, le tout premier commit d'un dépôt **Git** est l'ajout d'un fichier `.gitignore` à la racine du dépôt. Il contient une liste de fichiers que l'on ne souhaite pas commiter, par exemple :

```
# ignore backup files
*~
# ignore pdf files
*.pdf
# ignore latex-generated files
*.acn
*.acr
*.alg
*.aux
*.bbl
*.bcf
*.blg
*.dvi
*.fdb_latexmk
*.fls
```

```
*.glg
*.glo
*.gls
*.idx
*.ilg
*.ind
*.ist
*.lof
*.log
*.lot
*.maf
*.mtc
*.mtc0
*.nav
*.nlo
*.out
*.pdfsync
*.ps
*.run.xml
*.snm
*.synctex.gz
*.toc
*.vrb
*.xdy
*.tdo
# ignore temp file for algorithm2e latex package
*.loa
# ignore temp file for comment latex package
comment.cut
```

Il existe un site web permettant de générer des fichiers `.gitignore` pour tous types de projets : `https://www.gitignore.io/`

Enfin, il est toujours possible de commiter un fichier qui serait normalement ignoré à cause du `.gitignore`, en faisant :

```
$ git add -f fichier_normalement_ignoré
$ git commit
```

7.2 Concernant les commits

— Si possible, toujours vérifier qu'il n'y a pas d'erreur de compilation avant de commiter des fichiers source.

— Éviter de commiter des fichiers binaires/temporaires générés à la compilation (en pratique, on créera au tout début un fichier ".gitignore" à la racine du dépôt).

7.3 Concernant les messages de commit

— Choisir des messages de commit courts et explicites (une ligne de moins de 80 caractères, suivie éventuellement d'une ligne blanche et d'autres lignes de commentaires).
— Une convention assez répandue consiste à commencer par un verbe à la première personne du singulier au présent avec une majuscule (ce qui donnerait par exemple des messages de commits comme "Add a README" ou "Fix a minor bug" en anglais, et "Ajoute un README" ou "Corrige un bug mineur" en français).
— Voir par exemple l'article suivant : `http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html`

7.4 Concernant les branches

Il est conseillé de vérifier sur quelle branche on se trouve (`git branch`) avant de pousser vers le serveur (`git push`) :

```
$ gitk
$ git branch
$ git push
```

Ou bien, utiliser l'option « `-n` » (c.-à-d. `--dry-run`) de `git push` :

```
$ gitk
$ git push -n
$ git push
```

8 Documentation Git

Voici d'autres ressources en ligne sur **Git** :
— La documentation officielle : `http://git-scm.com/doc`
— Une introduction pour les connaisseurs de **Subversion** : `https://git.wiki.kernel.org/index.php/GitSvnCrashCourse`
— Un résumé synthétique des principales commandes (conçu par GitHub) : `http://git.github.io/git-reference/`
— Un ensemble de tutoriels illustrés (conçus par Atlassian) : `https://fr.atlassian.com/git/tutorials/`

Le service informatique de l'IRIT a créé une liste de diffusion dédiée aux discussions sur l'utilisation de **Git** : git-users@irit.fr ; vous pouvez vous y abonner en allant sur `https://listes.irit.fr/sympa/info/git-users`

9 Comparaison avec Mercurial

Git (écrit en C) et **Mercurial** (écrit principalement en Python) sont tous les deux des systèmes de gestion de version *décentralisés* et partagent de nombreux points communs, mais certaines caractéristiques de **Git** peuvent s'avérer cruciales :

- sa détection implicite des renommages
- sa gestion efficace des *branches* (locales et distantes)
- son support des fusions multiples (*octopus merges*)
- son support natif du *rebase*
- son architecture visant la sûreté de l'historique (SHA-1, *reflog*)
- ses performances.

Quelques articles de blog comparant les deux systèmes :

- `http://blogs.atlassian.com/2012/03/git-vs-mercurial-why-git/`
- `https://felipec.wordpress.com/2011/01/16/mercurial-vs-git-its-all-in-the-b`
- `https://importantshock.wordpress.com/2008/08/07/git-vs-mercurial/`

10 Licence de ce document