

Git : définitions

Git permet d'enregistrer l'évolution d'une arborescence de fichiers (le **répertoire de travail**) au sein d'un **dépôt** (le sous-répertoire `.git`). Développé en 2005 pour répondre aux besoins du développement du noyau Linux, il est puissant et versatile.

DÉFINITION Système de gestion de versions distribué

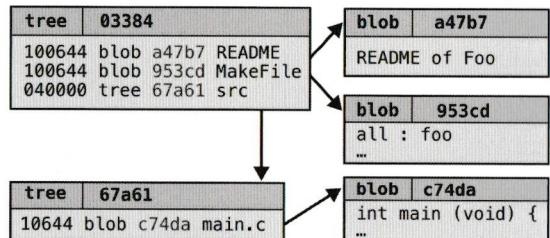
Dans les systèmes distribués, il n'est pas nécessaire d'avoir un dépôt central. Chaque intervenant dispose d'une copie complète du dépôt avec tout l'historique et peut effectuer autant de changements locaux qu'il le souhaite, sans devoir en référer au « serveur ». En pratique, on conserve des dépôts centraux pour faciliter les échanges et servir de référence commune. Mais les développeurs sont maîtres de ce qu'ils envoient sur ces dépôts, car le partage des changements est une opération totalement découpée de leur enregistrement.

DÉFINITION Système de suivi de contenu

Plus qu'un système de gestion de versions, Git est un système de gestion de contenus (*content tracker*). Son fonctionnement interne reflète ce choix. Nous présentons rapidement les concepts associés, leur connaissance facilitant la compréhension de l'outil.

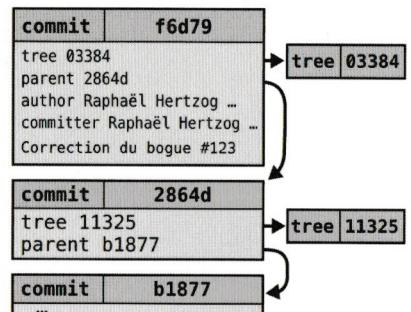
Un **dépôt Git** peut être vu comme une collection d'objets liés entre eux. Chaque objet est identifié par une chaîne de 40 caractères hexadécimaux, la somme de contrôle (*checksum*) SHA-1 de son contenu. Seuls les 5 premiers caractères figurent dans les exemples de ce mémento.

La représentation que Git se fait du répertoire de travail s'appuie sur deux types d'objets : des **données** (*blob*) et des **arborescences** (*tree*). Une arborescence n'est rien d'autre que la représentation d'un répertoire, c'est-à-dire une liste de noms dont chacun est associé à des permissions et à un identifiant d'objet (ce dernier étant soit un fichier – objet de type « données », soit un répertoire – objet de type « arborescence »). Le répertoire de travail est donc lui-même un objet de type « arborescence ». S'il contenait les fichiers `README`, `Makefile` et `src/main.c`, il serait vu de la sorte :



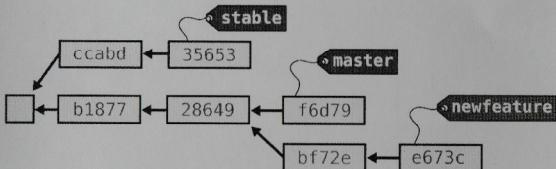
Un **historique de dépôt Git** n'est rien d'autre qu'une succession de versions du répertoire du travail (ce répertoire dans lequel vous travaillez et effectuez des modifications). Pour

représenter cela, Git s'appuie sur un objet de type **commit** associant des informations telles que l'auteur, la date et le message descriptif, à une version du répertoire de travail, mais aussi et surtout à un commit « parent » représentant la version précédente (un commit fusionnant plusieurs branches aura donc plusieurs parents).



Git : définitions

En plus des **données**, **arborescences** et **versions**, Git manipule des **références**, qui sont autant de pointeurs vers des commits, des labels, qui permettent de nommer des branches, des tags, etc. C'est ce qui permet à un dépôt Git de gérer en parallèle plusieurs **branches**, et donc plusieurs historiques. Une branche dans Git n'est pas un objet mais une référence qui pointe vers le dernier commit effectué dans la branche.



Mise en route

Première étape : indiquer à Git quelle identité sera enregistrée dans chaque commit (l'adresse par défaut `login@nom-machine` étant rarement la bonne).

```
git config --global user.email jean.dupont@example.com
git config --global user.name 'Jean Dupont'
```

L'option `--global` diffuse cette configuration à tous les dépôts : Git l'enregistre dans `$HOME/.gitconfig` afin qu'elle soit partagée entre tous vos dépôts. Il sera toujours possible d'utiliser une identité différente pour un dépôt particulier en omettant cette option au sein du dépôt concerné.

Création d'un dépôt

N'importe quel répertoire peut devenir un dépôt Git : il suffit d'exécuter la commande `git init` dans le répertoire concerné. Ceci crée un répertoire `.git` avec les métainformations nécessaires à Git. Peu importe que le répertoire de travail soit initialement vide ou non, le dépôt sera vierge et sans historique. L'inclusion de fichiers préexistants se fait séparément (cf. section « Modifications »). Un dépôt Git créé de la sorte combine dépôt et répertoire de travail. Pour créer un dépôt central dans lequel personne ne travaille mais utilisé par tous pour partager des modifications, il convient d'utiliser `git init --bare depot.git`. Cette commande crée un répertoire `depot.git` ne contenant que les métainformations Git, habituellement présentes dans le sous-répertoire `.git`.

REMARQUE L'extension `.git` du répertoire créé signale par convention un dépôt central sans répertoire de travail associé.

Téléchargement d'un dépôt existant

La plupart du temps, les dépôts Git sont préexistants et il convient de les télécharger pour pouvoir travailler dessus. La commande *ad hoc* est `git clone url`. Le dépôt ainsi téléchargé est une copie complète, avec tout l'historique. L'URL à employer qui vous a été communiquée peut être de plusieurs formes :

<code>ssh://login@git.example.com/chemin/depot.git</code>	Accès par SSH (méthode par défaut si le protocole est omis). Le préfixe <code>login@</code> est optionnel.
<code>login@git.example.com:/chemin/depot.git</code>	
<code>git://git.example.com/chemin/depot.git</code>	Accès par le serveur Git anonyme sur le port 9418 (<code>git-server</code>).
<code>http://git.example.com/chemin/depot.git</code>	Accès par HTTP.
<code>ftp://git.example.com/chemin/depot.git</code>	Accès par FTP.
<code>rsync://git.example.com/chemin/depot.git</code>	Accès par rsync.

REMARQUE On peut passer en dernier argument le nom du (nouveau) répertoire dans lequel le dépôt va être téléchargé, par défaut le dernier élément de l'URL, privé du suffixe `.git`.

Modifications (commandes de base)

Contrairement à d'autres systèmes de gestion de versions, lorsque vous exécutez `git commit`, Git n'enregistre pas les modifications depuis le répertoire de travail mais depuis « l'index », une zone tampon servant à préparer le prochain *commit*. Les commandes `git add` et `git rm` permettent d'intégrer certains changements du répertoire de travail dans l'index.

`git add fichier` Ajoute un nouveau fichier à l'index, et/ou met à jour le contenu du fichier dans l'index.

REMARQUE `git stage` est synonyme de `git add` ; « stage » exprime mieux l'opération de mise à jour de l'index.

`git add -p fichier` Ajoute à l'index un sous-ensemble des changements présents dans le fichier. Les changements individuels sont extraits du patch global (cf. `git diff`) et sont proposés de manière interactive.

`git rm fichier` Supprime un fichier du répertoire de travail et de l'index.

`git rm --cached fichier` Supprime un fichier de l'index tout en le conservant dans le répertoire de travail.

`git rm -r répertoire` Supprime du répertoire de travail et de l'index un répertoire et tous les fichiers qu'il contient, de manière récursive. `-r` fonctionne aussi avec `--cached`.

`git mv ancien nouveau` Renomme le fichier `ancien` en `nouveau`. Commande totalement équivalente à la séquence ci-dessous (Git ne conserve pas les informations de renommage en interne).

`git commit` Enregistre le contenu de l'index dans un nouveau commit.
`git commit -a` Enregistre le contenu du répertoire de travail dans un nouveau commit. Cette commande appelle implicitement `git add` sur tous les fichiers modifiés (mais pas sur les nouveaux fichiers encore inconnus de Git !) et `git rm` sur tous les fichiers supprimés.

`git commit fichier` Crée un nouveau commit contenant uniquement les changements concernant le fichier indiqué. C'est le contenu du répertoire de travail qui est utilisé. L'index est mis à jour pour intégrer ces changements.

`git commit --amend` Enregistre le contenu de l'index dans le dernier commit (HEAD). Cette opération remplace le dernier commit et ne doit pas être employée si ce commit a déjà été partagé avec d'autres utilisateurs. Elle permet simplement de corriger le dernier commit d'une branche privée.

`git citool` Ouvre une application graphique (`git gui`) qui permet de préparer puis d'enregistrer un commit.

`git status` Affiche l'état des fichiers du répertoire de travail et de l'index. Voir ci-dessous.

Avec Git on manipule couramment trois arborescences de fichiers : **HEAD** qui est la dernière version enregistrée de la branche courante, **l'index** et le **répertoire de travail** (*working directory* en anglais).

Pour suivre l'état des fichiers dans le répertoire de travail et dans l'index, Git fournit `git status`, qui classe les fichiers en différentes catégories :

- **non suivis** (*untracked files*) pour les fichiers inconnus de Git, qui ne font partie ni du dépôt, ni de l'index (on peut les masquer en les listant dans `.gitignore`);
- **modifiés** (*changes not staged for commit*) pour les fichiers du dépôt Git qui présentent des modifications dans le répertoire de travail par rapport à l'index ;
- **enregistrés dans l'index** (*changes to be committed*) pour les fichiers du dépôt pour lesquels l'index contient des changements par rapport au dépôt.

Un fichier peut être à la fois modifié et enregistré dans l'index, lorsque des changements interviennent après qu'on a appellé `git add` une première fois.

REMARQUE Pensez à `git stash nom` et `git stash apply` pour conserver votre travail en cours sans le committer, notamment si vous avez besoin de changer de branche.

Exploration (changements, historique)

Explorer des changements : git diff

git diff Affiche la différence entre le contenu de l'index et celui du répertoire de travail. Ces changements ne seront pas commis par `git commit` sauf si on prend soin de les ajouter à l'index avec `git add` sur les fichiers listés.

git diff --cached Affiche la différence entre le contenu du dernier `commit` et celui de l'index. Ces changements seront commis par `git commit`.

git diff HEAD Affiche la différence entre le contenu du dernier commit et celui du répertoire de travail. Cela correspond à ce qui serait commis par `git commit -a`.

git diff A B Affiche la différence entre le contenu pointé par `A` et celui pointé par `B`. `A` et `B` peuvent être des commits, tags, branches, objets de type « tree » ou « blob ».

git diff A...B Affiche la différence entre le contenu de l'ancêtre commun à `A` et `B`, et celui de `B`.

REMARQUE Il est toujours possible de restreindre la génération des différences à un sous-ensemble des fichiers en les passant sur la ligne de commande après un double tiret, par exemple : `git diff A B -- src/ Makefile`

Explorer l'historique : git log, git blame, gitk

git log [A] Affiche l'historique des changements (c'est à dire une liste de `commits`) accessibles depuis le `commit` pointé par `A`. Si `A` est omis, c'est `HEAD` qui est employé.

git log [A] -- fichier Idem en restreignant l'affichage aux commits qui modifient le `fichier` indiqué. Très pratique pour voir les derniers changements ayant affecté un fichier !

git log A..B Affiche l'historique entre les deux commits pointés par `A` et `B`.

git log --graph --oneline [A] Affiche un représentation textuelle de l'historique accessible à partir de `A` (`HEAD` si omis).

gitk [A] Démarrer un outil graphique de visualisation de l'historique accessible à partir de `A` (`HEAD` si omis).

git blame [A] [-- fichier] Affiche le contenu du `fichier` tel que présent dans l'arborescence de `A` (`HEAD` si omis) en indiquant pour chaque ligne le dernier `commit` à avoir modifié (ou introduit) celle-ci.

git log est une commande très puissante et dispose de plusieurs dizaines d'options. Les plus importantes sont reprises ci-dessous. En outre, à la place de la référence optionnelle notée `[A]`, aussi bien `git log` que `gitk` acceptent toutes les options et syntaxes supportées par `git rev-list`, les plus importantes étant listées ci-dessous. De plus ces commandes attendent un intervalle de commits en paramètres.

Options de `git rev-list` prises en charge par `git log` et `gitk`

<code>-n X</code>	Restreint l'affichage aux <code>X</code> derniers commits.
<code>--since=date</code>	Calcule un sous-ensemble des commits en fonction de leur ancienneté.
<code>--until=date</code>	
<code>--all</code>	Étend l'ensemble des commits à afficher à tous ceux accessibles par une référence connue (branche locale, tag, branche distante).
<code>--branches[=modèle]</code>	Étend l'ensemble des commits à afficher à tous ceux accessibles par une branche locale dont le nom respecte le <code>modèle</code> indiqué (toutes les branches locales si <code>modèle</code> est omis).
<code>--remotes[=modèle]</code>	Similaire à <code>--branches</code> mais pour les branches distantes.
<code>--tags[=modèle]</code>	Similaire à <code>--branches</code> mais pour les tags.
<code>-S texte</code>	Limite la recherche à des commits qui introduisent/suppriment des lignes contenant ce texte.
<code>intervalles</code>	De nombreuses syntaxes permettent de désigner des ensembles de commits (voir la section « Aide syntaxique »).

Exploration (changements, historique)

Options de `git log`

<code>--decorate</code>	Affiche les noms des branches et tags à côté du <code>commit</code> correspondant.
<code>--oneline</code>	Affiche chaque commit sur une seule ligne (en utilisant l'identifiant de commit abrégé et la première ligne du message de log).
<code>-p</code>	Affiche le patch correspondant à chaque <code>commit</code> .
<code>--stat</code>	Affiche un <code>diffstat</code> correspondant aux changements du <code>commit</code> : la liste des fichiers modifiés avec des statistiques sur le nombre de lignes ajoutées et supprimées.

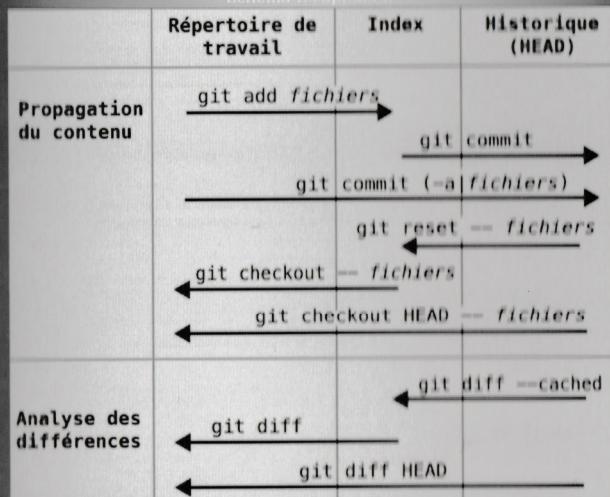
EXEMPLE `git blame`

```
* git blame Makefile.am
[...]
d41a6301 (Scott James Remnant 2005-03-11 09:00:14 +0000 23) EXTRA_DIST = \
cbf95b51 (Guillem Jover 2009-10-25 23:06:46 +0100 24) .mailmap \
1a0bade (Guillem Jover 2009-03-02 06:56:25 +0200 25) ChangeLog.old \
fd3aaecf (Raphael Hertzog 2007-12-27 15:22:38 +0100 26) README.translators \
971e675a (Guillem Jover 2009-10-25 23:56:30 +0100 27) get-version \
c14a29ae (Guillem Jover 2009-09-23 03:21:16 +0200 28) doc/README.api \
b3db5f47 (Sven Joachim 2010-07-25 23:34:03 +0200 31) doc/icon-epilog \
b3db5f47 (Sven Joachim 2010-07-25 23:34:03 +0200 32) doc/icon-prolog \
10440009 (Ian Jackson 2008-03-30 09:47:15 +0300 33) doc/triggers.txt \
[...]
```

EXEMPLE `git log`

```
* git log --oneline --graph --decorate
* aee5795 (HEAD, origin/master, origin/HEAD, master) dpkg: Add restrictions for
* d00d440 Merge commit '1.16.1.2'
| \
| * 0003dd4 (tag: 1.16.1.2, origin/sid, sid) Release 1.16.1.2
| * 274f813 dpkg-buildflags(l): -D_FORTIFY_SOURCE=2 goes in CPPFLAGS
| * 67ff656 dpkg: Refactor disappear code into its own pkg_disappear() function
| * c38afe2 libdpkg: Add new fd_fd_copy_and_md5()
| * d76bb53 Merge commit '1.16.1.1'
| \
| * 9babd2d (tag: 1.16.1.1) Release 1.16.1.1
| * 343c493 dpkg-buildflags: Disable bindnow if retro is not used
| * 669e0ff dpkg-source: do not ignore the automatic patch when checking for un...
| * 2ac198b Clarify README instructions
| * 155c307 libdpkg: Add missing symbols to the version script
| \
| * 437fb05 (tag: 1.16.1) Release 1.16.1
* 7af0fb2 dpkg-buildflags: Disable bind now by default
```

Schéma récapitulatif



Annulation, restauration et nettoyage

Effacer des changements non commis

<code>git reset</code>	Efface les changements en attente de commit (autrement dit réinitialise l'index avec le contenu de <code>HEAD</code>). Les fichiers du répertoire de travail sont inchangés.
<code>git reset --hard</code>	Efface tous les changements présents dans l'index et dans le répertoire de travail (autrement dit, réinitialise l'index et le répertoire de travail avec le contenu de <code>HEAD</code>). Attention, cette commande est destructive ! Il est impossible de récupérer les changements qui étaient présents dans le répertoire de travail.
<code>git reset [-] fichier...</code>	Efface uniquement les changements en attente de commit qui affectaient le <code>fichier</code> (autrement dit restaure l'état du <code>fichier</code> dans l'index, à son état dans <code>HEAD</code>). Cette commande est donc la réciproque de <code>git add</code> .
<code>git checkout [-] fichier...</code>	Efface les changements du <code>fichier</code> du répertoire de travail par rapport à sa copie dans l'index (autrement dit restaure le <code>fichier</code> du répertoire de travail à son état dans l'index).
<code>git checkout HEAD [-] fichier...</code>	Efface les changements du <code>fichier</code> du répertoire de travail et de l'index (autrement dit restaure le <code>fichier</code> du répertoire de travail et de l'index à son état dans <code>HEAD</code>). Équivalent de <code>git reset -- fichier & git checkout -- fichier</code> .

REMARQUE Pour ces trois dernières commandes, l'option `-p` permet de choisir interactivement le sous-ensemble des changements à effacer.

Restaurer l'état d'un fichier donné

<code>git reset A [-] fichier...</code>	Restaure l'état du <code>fichier</code> dans l'index, à son état dans le commit <code>A</code> .
<code>git checkout A [-] fichier...</code>	Restaure le <code>fichier</code> du répertoire de travail (et de l'index) à son état dans le commit <code>A</code> .

REMARQUE Pour ces deux commandes, l'option `-p` permet de choisir interactivement le sous-ensemble des changements à effacer.

Revenir sur des changements commis

Lorsque `git reset` prend un paramètre un `commit` (et aucun fichier), il fait pointer `HEAD` sur le `commit` indiqué. Selon les options, il modifie ou non l'index et le répertoire de travail. Cette commande permet donc de revenir en arrière dans l'historique en laissant ou non la possibilité de recréer les derniers `commits` à partir des changements encore présents dans l'index et/ou le répertoire de travail.

Commande	Commit	HEAD	Index	Répertoire de travail
<code>git reset --soft A</code>	A	A	inchangé	inchangé
<code>git reset --medium A</code>	A	A	A	inchangé
<code>git reset A</code>				
<code>git reset --hard A</code>	A	A	A	A

Nettoyage

`git clean` sert à supprimer du répertoire de travail les fichiers non suivis (et non ignorés) par Git. Il faut systématiquement passer l'option `-f` pour réaliser cette opération destructrice (sauf si l'option `clean.requireForce` est à `false`). On peut aussi employer l'option `-n` pour simplement afficher ce qui aurait été supprimé (si l'on avait mis `-f` au lieu de `-n`). L'option `-x` inclut également les fichiers ignorés (listés dans `.gitignore`) et l'option `-d` inclut les répertoires en plus des fichiers.

Modifications (commandes avancées)

git am boîte-aux-lettres
git am fichiers-patch

Applique une série de patchs reçus par courriel et crée un commit pour chaque patch. On peut lui passer une *boîte-aux-lettres* au format Mailbox ou Maildir, ou bien directement un ensemble de *patchs* générés avec `git format-patch`. Les informations du commit sont directement extraites du message électronique (remarquez que la sortie de `git format-patch` génère effectivement un commit).

REMARQUE Si le processus s'interrompt à cause d'un patch qui ne s'applique pas, vous pouvez appliquer ce dernier à la main et l'enregistrer dans l'index avant de relancer le processus avec `git am --continue`. Pour ignorer le patch problématique, on fera `git am --skip`. Pour abandonner l'opération entière : `git am --abort`.

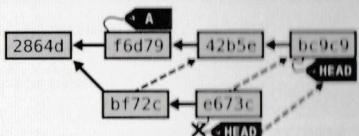
git cherry-pick A.. Pour chaque commit indiqué (*A..*), crée un nouveau commit dans la branche courante et duplique le contenu du commit indiqué. Alors qu'une fusion intègre tous les changements d'une branche, cette commande permet de choisir exactement quels changements intégrer dans la branche courante.

REMARQUE L'opération peut générer des conflits qu'il convient de régler en enregistrant le résultat désiré dans l'index, puis en appelant `git cherry-pick --continue`. Si on veut annuler l'opération, effectuer `git cherry-pick --abort`.

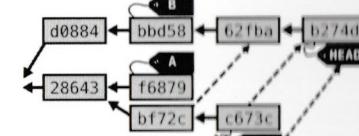
git revert A.. Pour chaque commit indiqué (*A..*), crée un nouveau commit dans la branche courante qui annule les changements apportés par le commit indiqué.

REMARQUE `git revert` est l'opération inverse de `git cherry-pick`. Il est implémenté avec la même fonction que `git cherry-pick`, mais c'est le patch inversé du commit indiqué qui est employé (un ajout devient une suppression et vice-versa).

git rebase A Réécrit la branche courante de telle manière que tous ses commits soient des descendants de *A*. Pour cela, tous les commits de la branche courante (depuis le premier descendant du premier ancêtre commun à *A* jusqu'à HEAD) sont recréés (comme avec `git cherry-pick`). Le plus ancien commit devient le descendant direct de *A*.



git rebase --onto B A Réécrit la branche courante (dérivée de *A*) de telle manière que tous ses commits soient des descendants de *B*. Pour cela, tous les commits de la branche courante (depuis le premier descendant du premier ancêtre commun à *A* jusqu'à HEAD) sont recréés (comme avec `git cherry-pick`). Le plus ancien commit devient le descendant direct de *B*.



git rebase -i A L'option `-i` permet de modifier de manière interactive l'historique des changements en réordonnant les commits, en fusionnant certains commits dans d'autres (`squash`, `fixup`), en modifiant ou découplant des commits (`edit`), voire en en supprimant certains. `git commit --fixup=B` (ou `--squash=B`) crée un commit marqué comme étant à fusionner dans *B* au cours du prochain `git rebase --autosquash` qui inclura ce commit.

REMARQUE La différence entre `squash` et `fixup` est que le premier ouvre un éditeur pour modifier le message de commit résultant de la fusion, alors que le second conserve le message du commit cible.

Aide syntaxique : treeish et intervalles

Quand nous mentionnons *A* (ou *B*) dans une description de commande, il est possible d'utiliser une multitude de syntaxes pour désigner un commit. Si on se contente souvent de donner le nom d'une branche pointant implicitement sur le dernier commit qu'elle contient, il y a bien des situations où l'on voudra désigner d'autres commits. La documentation (anglaise) de Git désigne ces paramètres (*A* ou *B*) comme étant des *treeish*, ce que l'on pourrait traduire par « qui ressemble à une arborescence ». Cette formulation signifie que Git accepte toute expression lui permettant d'identifier un objet de type arborescence (« *tree* »). Voici les syntaxes les plus utiles :

chaîne-hex

Toute chaîne de caractère hexadécimal (0-9, a-f) sera interprétée comme étant un identifiant de commit. Si la chaîne est plus courte que les 40 caractères requis par une somme de contrôle SHA1, elle est interprétée comme étant le début d'un tel identifiant (en pratique, l'identification ne nécessite souvent que 5 à 7 caractères).
EXEMPLE `dae86e1950b1277e545cee180551750029cf735`, `dae86e`

refs/heads/foo
refs/tags/bar
**refs/remotes/
depot/baz**

Nom complet d'une référence symbolique désignant respectivement la branche locale *foo*, le tag *bar*, la branche distante *baz* dans le dépôt distant *depot*.

foo

Nom court d'une référence symbolique, que Git va résoudre en prenant le premier objet valide dans la liste suivante :

`$GIT_DIR/`**foo** (gère le cas des références HEAD et *_HEAD)
refs/**foo**
refs/**tags/****foo**
refs/**heads/****foo**
refs/**remotes/****foo**
refs/**remotes/****foo/****HEAD**

EXEMPLE `master`, `heads/master`, `refs/heads/master`, `origin/master`, `remotes/origin/master`, `refs/remotes/origin/master`

A^

Le premier commit parent du commit pointé par *A*.

A^n

Le *n*ième parent du commit pointé par *A*. Pour des valeurs *n* > 1, cette syntaxe n'est valable que pour des commits fusionnant plusieurs branches.

A~n

Le *n*ième commit en arrière dans l'historique. *A~1* désigne le même commit que *A^*, de même pour *A~2* et *A^2*, etc.

foo@{n}

Le commit sur lequel pointait *foo* en revenant *n* « changements » de *foo* en arrière (par « changement », on entend toute opération qui a fait pointer *foo* sur un commit différent). Si *foo* est omis, on utilise le nom de la branche courante. Ex : `master@{2}, @{3}`.
REMARQUE Cette syntaxe utilise le *reflog* de git, une sauvegarde automatique (sur une durée configurable) des anciennes versions des branches présentes dans le dépôt.

foo@{date}

Le commit sur lequel pointait *foo* à la *date* (et heure) indiquée. La date peut s'exprimer de nombreuses manières différentes comme *yesterday*, *3 days 1 hour ago* ou simplement *2012-09-15 15:32*. Ex : `master@{1 week ago}, HEAD@{14:30}`

REMARQUE Cette syntaxe utilise également le *reflog* de git, ce qui veut dire que cela renvoie l'état de la branche *locale* à la date indiquée, et non pas l'état de l'éventuelle branche distante associée.

At fichier

Le *fichier* de l'arborescence associée au commit pointé par *A*. Contrairement aux autres syntaxes présentées, cette syntaxe ne renvoie pas un objet de type *commit* mais un objet de type *blob*. Elle utilise notamment avec `git show` qui sait afficher le contenu de n'importe quel objet, ou encore avec `git diff` qui sait comparer deux fichiers arbitraires.

REMARQUE Retrouvez toutes ces syntaxes dans la page de manuel `gitrevisions(7)`.

Aide syntaxique : treeish et intervalles

Syntaxe pour décrire des ensembles de commits sous forme d'intervalles (ranges)

A *A* et tous ses ancêtres.

REMARQUE Lorsque Git attend un intervalle, désigner un commit inclus automatiquement tous les commits anciêtres.

A .. B *A* et tous ses ancêtres, sauf *B* et ses ancêtres.

B..A

A...B *A* et *B* et tous leurs ancêtres sauf les ancêtres communs aux deux.

A! Le commit *A* tout seul.

L'arme ultime pour isoler le commit fautif : git bisect

Un bogue qu'on peut reproduire de manière fiable est un bogue à moitié corrigé, dit-on souvent. C'est d'autant plus vrai si ce bogue est une régression, car Git fournit une commande formidable pour identifier le commit fautif, `git bisect`, qui implémente une recherche par dichotomie. Il faut donc spécifier un ensemble de commits à tester en indiquant une version connue comme étant affectée par le problème (`bad`) et une version non affectée (`good`). Git extrait le commit médian de cet ensemble et attend votre verdict : cette version est-elle affectée ou non ? Partant de là, Git réduit de moitié l'espace des commits à tester. Après quelques itérations, Git indique le premier commit affecté par le problème.

git bisect start
**git bisect start [A
[B...]]**

Démare une recherche par dichotomie. Si *A* est précisé, il s'agit d'un « mauvais » commit (c'est-à-dire affecté par le problème). Si *B* est précisé, il s'agit d'un « bon » commit (non affecté par le problème).

git bisect bad [A]
git bisect good [A]

Pour restreindre l'ensemble des commits à ceux qui modifient des fichiers, il est possible d'en indiquer les chemins (fichiers ou répertoires) après deux-tirets (`..`).

git bisect skip [A]
git bisect reset

Enregistre *A* comme étant affecté par le problème, il s'agit donc d'une *borne haute* à l'ensemble de commits à tester. Si *A* est omis, c'est HEAD qui est utilisé.
Enregistre *A* comme étant non affecté par le problème, il s'agit donc d'une *borne basse* à l'ensemble de commits à tester. Si *A* est omis, c'est HEAD qui est utilisé.

git bisect run commande

Enregistre *A* comme étant non testable, si *A* est omis, c'est HEAD qui est utilisé.
ATTENTION Si le commit fautif est adjacent à un commit non testable, cela peut empêcher Git de l'identifier.

git bisect visualize

Termine ou interrompt la recherche par dichotomie. Le répertoire de travail et le dépôt sont remis dans leur état initial.
Automatisse la recherche à l'aide d'une commande externe. Pour chaque test, Git exécute *commande* et utilise le code de retour pour déterminer le statut du commit : 0, le commit est « bon » ; 1-127 (sauf 128), le commit est « mauvais » ; 128+, le commit est non testable ; 128-255, une erreur s'est produite et le processus est interrompu.
Affiche l'ensemble des commits restant à tester dans `gitk`.

Bonnes pratiques

Cohérence et taille des commits : enregistrez des changements cohérents et minimaux

Un commit doit contenir des **changements cohérents**, qui vont ensemble. Ainsi les corrections de deux bogues différents doivent en règle générale se retrouver dans deux commits différents. De plus, les petits commits sont bien plus **faciles à lire** pour les autres développeurs. En cas de problème, cela permet aussi de retrouver plus facilement le changement fautif (grâce à `git bisect` notamment).

Si des changements hétérogènes sont mélangés dans votre répertoire de travail, enregistrez-les par petits bouts :

git add -p, git checkout -p, git rebase -i, git reset -p.

Rythme des commits : faites des commits fréquents et prévenez les conflits

Cela permet d'enregistrer des changements cohérents et de petite taille, et de partager votre code plus souvent avec les autres développeurs.

En intégrant vos développements en cours plus régulièrement, vous prévenez les conflits importants qui arrivent souvent en cas d'intégration tardive d'une branche. Avoir peu de commits contenant beaucoup de modifications, et ne les partager que rarement, génère des conflits plus gros et plus difficiles à résoudre.

Enfin, avant de partager votre code, utilisez si nécessaire `git rebase -i` pour réordonner, reformuler et réécrire des **commits** afin de présenter un historique lisible aux autres programmeurs, dans lequel chaque **commit** représente un ensemble minimal et cohérent de modifications clairement documentées.

N'enregistrez pas du travail à moitié fini : utilisez git stash nom et git stash apply

Un **commit** doit contenir un travail « complet ». Ceci n'est pas en opposition avec le fait qu'il faut faire des commits de petite taille : il ne s'agit pas de faire en un seul commit une fonctionnalité compliquée, mais de découper la fonctionnalité en plusieurs petites tâches, et d'enregistrer un changement par petite tâche.

Ne faites pas de commit le soir avant de quitter le bureau juste pour avoir un répertoire de travail propre. Dans ce genre de cas, recourez à `git stash`, car les commits temporaires ont la fâcheuse tendance à se retrouver dans les dépôts centraux à cause de `git push` trop hâtifs par exemple...

Documentation : écrivez de bons messages de commit

La gestion de versions perd son sens si les messages de commit sont de mauvaise qualité. Les messages s'adressent tant aux autres qu'à vous-même, ils seront lus par toutes les personnes qui vont ausculter l'historique et se demander invariablement « Pourquoi ce changement ? Qu'est-ce qui a changé ? » plutôt que « Comment fonctionne le code ? ».

Un bon message de commit commence par un résumé d'une cinquantaine de caractères qui décrit le changement. Suit une ligne vide qui le sépare du message détaillé, qui sera aussi long que nécessaire. Un bon message précise donc : 1) ce qui a motivé le changement (bogue, nouvelle fonctionnalité, nettoyage, optimisation...) ; 2) ce qui a changé par rapport à la version précédente.

Ce n'est pas le lieu pour décrire l'implémentation en détail, elle doit l'être dans le code sous forme de commentaires.

N'enregistrez jamais de changements non fonctionnels : testez avant de committer !

Avant de publier un commit, il est critique de vérifier que le code en est correct (tests, règles de programmation, etc.), d'autant plus s'il s'agit de séries de changements remises en forme en utilisant `git rebase -i` car une mauvaise manipulation au cours de cette opération peut introduire des régressions (même une régression uniquement présente sur des commits intermédiaires peut nuire en rendant l'usage de `git bisect` plus difficile).

En d'autres termes, testez vos changements, l'un après l'autre. Ne publiez pas des changements mal testés, vous risqueriez de faire perdre un temps précieux aux autres programmeurs et de rendre des commandes comme `git bisect` totalement inutiles.

GÉRÉNEMENT Git en ligne : github, gitorious...

Des services tels [GitHub.com](#) et [Gitorious.com](#) proposent d'héberger vos dépôts Git en ligne. Ils sont généralement gratuits pour les projets de logiciel libre. Chaque dépôt est avant tout associé à un utilisateur, et il n'est pas rare d'avoir plusieurs centaines de copies d'un dépôt pour un logiciel populaire. C'est la conséquence naturelle du modèle de développement associé : toute contribution externe est réalisée dans une copie du dépôt (`fork`) sur une branche (publique) que l'on peut ensuite soumettre pour intégration (*via une pull request*).

La création de dépôts s'effectue par l'interface web mais ils sont ensuite gérés directement *via git* et un dépôt distant accessible par SSH (l'utilisateur doit donc enregistrer sa clé publique SSH auprès de ces services).

Bonnes pratiques

N'hésitez pas à user et abuser des branches

Git a dès l'abord été conçu pour faciliter la création de branches et surtout leur fusion. Des changements peuvent ainsi rester totalement indépendants ; cela ouvre de nombreuses perspectives de travail :

- branches de correctifs intégrables dans plusieurs versions du logiciel ;
- expérimentations sans risque (il est facile de « jeter » une branche, bien moins de se débarrasser de commits entrelacés avec le développement « normal ») ;
- branches par fonctionnalités, pour que les autres développeurs puissent facilement tester un sous-ensemble de vos changements et vous faire des retours dessus ;
- branches temporaires de travail collaboratif sur une fonctionnalité avec un autre développeur avant la soumission finale, etc.

Choisissez des règles de travail et conventions communes : adoptez le même workflow

La souplesse de Git (par opposition à des systèmes de gestion de versions centralisés comme CVS, Subversion, perforce) est l'une de ses forces... mais elle peut devenir un cauchemar si tous les développeurs ne travaillent pas de la même manière. Établissez donc quelques règles dans la façon de travailler :

- utilisation d'un dépôt central ;
- nommage des différentes branches « officielles » ;
- règles pour soumettre les *patches* (directement, par soumission sur une liste de diffusion, via des outils de relecture...), etc.

Choisissez des outils qui s'intègrent bien avec Git

Le système de gestion de versions est la clef de voûte d'une bonne gestion de projet ; préférez ceux des autres outils classiques (gestion de tickets, relecture de code, intégration continue, etc.) qui s'intègrent bien avec Git, pour automatiser des tâches répétitives, sources d'erreurs multiples. Choisissez par exemple un système de gestion de tickets qui « comprend » Git pour « fermer » les tickets depuis les messages de commit afin d'éviter des oubli humains. Utilisez un outil de relecture de code sachant manipuler directement un dépôt Git, pour éviter des erreurs lors de l'intégration d'une branche.

POUR ALLER PLUS LOIN Outils de l'écosystème Git

■ Interfaces graphiques

Pour Windows, [Git Extensions](#) s'intègre dans l'explorateur : <http://sourceforge.net/projects/gitextensions/>
Pour Mac OS X, [GHX](#) est une application indépendante : <http://gitx.lauzon.com/>

■ Gestion de patches

[S4Git](#) gère les patches à la quitt dans Git : <http://www.procode.org/stgit/>
[TopGit](#) gère les *topic* branches liées entre elles : <http://repo.or.cz/w/topgit.git>

■ Hébergement et contrôle d'accès à des dépôts Git

[Gitorious](#) : <http://gitorious.org/gitorious/mainline>
[Gitolite](#) : <http://github.com/starnac/gitolite>

■ Revue de code

[Gerrit](#) : <http://code.google.com/p/gerrit/>

Chez le même éditeur

[Debian Squeeze](#) (Cahier de l'Admin), R. HERTZOG

[BSD](#) (Cahier de l'Admin), E. DREYFUS

[Mémento Unix/Linux](#), 2^e éd., I. HUBAIN & E. DREYFUS

[Asterisk](#) (Cahier de l'admin), P. SULTAN

[HTML5](#). Une référence pour le développeur web, R. RIMEL

[CSS avancées](#), 2^e éd., [Mémento Unix/Linux](#), 2^e éd.,

I. HUBAIN & E. DREYFUS R. GOETTER

[Apprendre à programmer avec Python](#), 3^e éd., G. SWINNEN

Code éditeur : G13433



ISBN : 978-2-212-33438-4

9 782212 134384

Conception : Nord Concept

Raphaël Hertzog
Pierre Habouzit

EYROLLES

L'arme ultime pour isoler le commit fautif : git bisect

git bisect log

Affiche un historique de la recherche par dichotomie, qui peut être sauvegardé pour être rejoué (voir `git bisect replay`). En cas d'erreur sur l'une des réponses, on peut modifier l'historique avant de le rejouer.

git bisect replay fichier

Extrait la liste des bons et mauvais commits depuis le *fichier* et continue la recherche du commit fautif à partir de là.

EXEMPLE git bisect pour identifier le commit fautif

```
git bisect start -- src/archives.c
git bisect bad 1.16.0
git bisect good 1.15.7
Bisecting: 35 revisions left to test after this (roughly
5 steps)
[5d74139fdale4850c436765131b949232b54cc80] Use fdio instead of ad-hoc code to handle interrupted I/O
git bisect good
Bisecting: 17 revisions left to test after this (roughly
4 steps)
[ecb9d03ffd24e0e7319872bf190ce99dbb37b18e] libdpkg: Rename Tarinfo to tar_entry
git bisect bad
d25407536dbed4cad2943187b36fb6c92a6b5ab is the first bad
commit
git bisect reset
```

Utiliser Git avec un dépôt Subversion : git svn

`git svn` crée un dépôt Git local s'interfaisant avec un dépôt distant Subversion. Git est utilisé normalement pour travailler et enregistrer des changements, quelques commandes spécifiques permettant de récupérer et d'envoyer les changements depuis/vers le serveur Subversion.

ATTENTION N'utilisez pas les fonctionnalités de fusion de Git car elles ne peuvent se transposer aisément à Subversion. `git svn` s'appuie à la place sur la fonctionnalité `rebase` pour maintenir une branche à jour.

`git svn clone svn-uri` Crée un dépôt Git miroir du répertoire pointé par l'URL `svn-uri`. Avec l'option `-s`, les branches et les tags SVN sont également importés. Dans ce cas, l'URL indiquée est implicitement suffisée de `trunk`, branches et tags. Lorsque cette structure n'est pas respectée, les options `-T`, `-b` et `-t` permettent d'indiquer l'URL pour chacun de ces 3 répertoires standard.

`git svn rebase` Met à jour la copie locale du dépôt Subversion puis rebâse la branche locale courante sur la branche SVN mise à jour. C'est l'équivalent de `svn update`.

`git svn fetch` Met à jour la copie locale du dépôt Subversion sans toucher à la branche locale courante.

`git svn dcommit` Envoie les changements locaux (de la branche courante) vers le dépôt Subversion. C'est l'équivalent de plusieurs `svn commit` successifs sur tous les changements non encore publiés.

`git svn branch foo` Crée une nouvelle branche Subversion nommée `foo` qui démarre du commit actuel.

`git svn tag foo` Crée un nouveau tag Subversion nommé `foo` qui pointe sur le commit actuel.

Fichiers de configuration

Les fichiers de configuration de Git ont un format proche du `.ini` classique. On les modifie avec `git config` ou dans votre éditeur favori.

/etc/gitconfig

Fichier de configuration système, normalement paramétré par le packager ou par l'administrateur système. Modifié par `git config --system option valeur`

~/.gitconfig

Fichier de configuration global de l'utilisateur. Modifié par `git config --global option valeur`

depot/.git/config

Fichier de configuration spécifique au dépôt local. Modifié par `git config option valeur` depuis n'importe quel répertoire du dépôt local.

En cas de définition multiple d'un paramètre de configuration : le fichier du dépôt local est prioritaire sur le fichier de l'utilisateur, lui-même prioritaire sur le fichier de configuration système. L'intégralité des paramètres de configuration de Git est documentée dans la page de manuel `git config(1)`.

Principales options de configuration de Git

alias.nom

Définit `nom` comme une nouvelle commande Git qui invoque en réalité la commande indiquée par la valeur de cette option (voir la section suivante pour des alias très utiles).

user.email

Adresse électronique utilisée pour les commits.

user.name

Nom complet utilisé pour les commits.

user.signingkey

Identifiant de la clé GPG à utiliser pour signer les tags. En général correctement déduit de `user.email`.

core.editor

Permet de choisir l'éditeur utilisé pour les commits. Attention, cet éditeur ne doit pas passer en arrière-plan une fois invoqué.

merge.conflictstyle

Peut être positionné sur `difft3`, ce qui permet d'avoir en plus des marques de conflit avec la version « de gauche » (marqueurs <<<<<), ou « de droite » (marqueurs >>>>>), la version de l'ancêtre commun (marqueurs |||||) et -----.

color.ui

Contrôle l'usage des couleurs dans la sortie des différentes commandes qui savent mettre en valeur ce qu'elles affichent. La valeur recommandée est `auto` qui active la couleur lorsque la commande est exécutée sur un terminal mais qui la désactive dans le cas où la sortie est redirigée vers un autre programme (dans un *pipe*). `never` prescrit de ne jamais utiliser la couleur, `always` de toujours le faire.

format.pretty

Format de présentation des commits employé par les commandes `git log`, `git show` et `git whatchanged`. La valeur par défaut est `medium`, les autres valeurs possibles étant `oneline`, `short`, `full`, `fuller`, `email`, `raw`.

merge.tool

Outil invoqué par `git mergetool` pour résoudre les conflits. `git mergetool --help` liste les outils pris en charge.

pretty.nom

Permet de créer des formats personnalisés pour `git log` sous le nom `nom`, qu'on utilise ensuite via `git log --pretty=nom`.

remote.dépôt.push

Définit explicitement la liste des références envoyées par défaut au dépôt distant lorsque l'on fait `git push dépôt`.

Principales options de configuration de Git

push.default

Définit le mode de fonctionnement de `git push` [dépôt-distant] sans arguments (origin est utilisé comme dépôt-distant si ce paramètre est omis) lorsque `remote.dépôt-distant.push` n'est pas défini.

Les valeurs possibles de `push-default` sont :

- **nothing** : ne fait rien ;
- **matching** : pousse les branches qui correspondent entre le dépôt local et le dépôt distant (comportement par défaut) ;
- **upstream** : pousse la branche courante vers la branche du dépôt distant qu'elle suit. *C'est le comportement recommandé car le moins surprenant !* ;
- **current** : pousse la branche courante vers une branche de même nom dans le dépôt distant.

rebase.autosquash

Active par défaut l'option `--autosquash` de la commande `git rebase -i`.

status.showUntrackedFiles

Si cette option est à `no`, `git status` n'affiche plus les fichiers inconnus de Git (non gérés par celui-ci). Option intéressante pour les très gros dépôts où la commande `git status` est assez longue à s'exécuter.

url.base, pushInsteadOf

Par défaut Git utilise la même URL pour télécharger un dépôt que pour y envoyer des changements. Lorsque ce n'est pas possible, et pour éviter de devoir configurer une URL spécifique (`remote.name.pushurl`) pour chaque dépôt, cette option indique à Git de réécrire l'URL utilisée pour l'envoi en remplaçant le préfixe indiqué dans la valeur associée à `base`.

Alias fréquemment employés

Pour créer un alias, il faut exécuter :

`git config --global alias.nom commande`.

Alias	Commande	Explications
c1	commit	Version courte de cette commande fréquemment utilisée.
co	checkout	Idem.
st	status	Idem.
br	branch	Idem.
tip	log -n 1 --abbrev-commit --decorate	Affiche le dernier commit de la branche courante (ou de la référence indiquée si l'on exécute <code>git tip A</code>).
unstage	reset HEAD --	Supprime les changements en attente dans l'index. Si l'on indique une liste de fichiers, seuls ceux-ci sont affectés.
staged	diff --cached	Affiche les changements en attente dans l'index.
fix	commit --amend	Corrige le dernier commit.

Modifications (commandes avancées)

git am boîte-aux-lettres
git am fichiers-patch

Applique une série de patches reçus par courriel et crée un commit pour chaque patch. On peut lui passer une *boîte-aux-lettres* au format Mailbox ou Maildir, ou bien directement un ensemble de *patches* générés avec **git format-patch**. Les informations du commit sont directement extraites du message électronique (remarquez que la sortie de **git format-patch** génère effectivement un courriel).

REMARQUE Si le processus s'interrompt à cause d'un patch qui ne s'applique pas, vous pouvez appliquer ce dernier à la main et l'enregistrer dans l'index avant de relancer le processus avec **git am --continue**. Pour ignorer le patch problématique, on fera **git am -skip**. Pour abandonner l'opération entière : **git am --abort**.

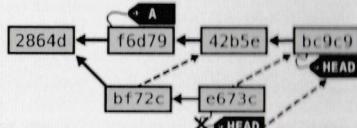
git cherry-pick A... Pour chaque commit indiqué (*A...*), crée un nouveau commit dans la branche courante et duplique le contenu du commit indiqué. Alors qu'une fusion intègre tous les changements d'une branche, cette commande permet de choisir exactement quels changements intégrer dans la branche courante.

REMARQUE L'opération peut générer des conflits qu'il convient de régler en enregistrant le résultat désiré dans l'index, puis en appelant **git cherry-pick --continue**. Si on veut annuler l'opération, effectuer **git cherry-pick --abort**.

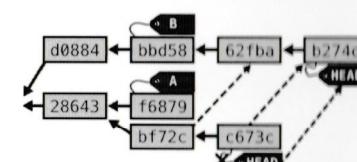
git revert A... Pour chaque commit indiqué (*A...*), crée un nouveau commit dans la branche courante qui annule les changements apportés par le commit indiqué.

REMARQUE **git revert** est l'opération inverse de **git cherry-pick**. Il est implémenté avec la même fonction que **git cherry-pick**, mais c'est le patch inversé du commit indiqué qui est employé (un ajout devient une suppression et vice-versa).

git rebase A Réécrit la branche courante de telle manière que tous ses commits soient des descendants de *A*. Pour cela, tous les commits de la branche courante (depuis le premier descendant du premier ancêtre commun à *A* jusqu'à HEAD) sont recréés (comme avec **git cherry-pick**). Le plus ancien commit devient le descendant direct de *A*.



git rebase --onto B A Réécrit la branche courante (dérivée de *A*) de telle manière que tous ses commits soient des descendants de *B*. Pour cela, tous les commits de la branche courante (depuis le premier descendant du premier ancêtre commun à *A* jusqu'à HEAD) sont recréés (comme avec **git cherry-pick**). Le plus ancien commit devient le descendant direct de *B*.



git rebase -i A L'option **-i** permet de modifier de manière interactive l'historique des changements en réordonnant les *commits*, en fusionnant certains commits dans d'autres (**squash**, **fixup**), en modifiant ou découpant des commits (**edit**), voire en en supprimant certains. **git commit --fixup-B** (ou **--squash-B**) crée un commit marqué comme étant à fusionner dans *B* au cours du prochain **git rebase --autosquash -i** qui inclura ce commit.

REMARQUE La différence entre **squash** et **fixup** est que le premier ouvre un éditeur pour modifier le message de commit résultant de la fusion, alors que le second conserve le message du commit ciblé.

Aide syntaxique : treeish et intervalles

Quand nous mentionnons *A* (ou *B*) dans une description de commande, il est possible d'utiliser une multitude de syntaxes pour désigner un commit. Si on se contente souvent de donner le nom d'une branche pointant implicitement sur le dernier commit qu'elle contient, il y a bien des situations où l'on voudra désigner d'autres commits. La documentation (anglaise) de Git désigne ces paramètres (*A* ou *B*) comme étant des *treeish*, ce que l'on pourrait traduire par « qui ressemble à une arborescence ». Cette formulation signifie que Git accepte toute expression lui permettant d'identifier un objet de type arborescence (*« tree »*). Voici les syntaxes les plus utiles :

chaîne-hex Toute chaîne de caractère hexadécimal (0-9, a-f) sera interprétée comme étant un identifiant de commit. Si la chaîne est plus courte que les 40 caractères requis par une somme de contrôle SHA1, elle est interprétée comme étant le début d'un tel identifiant (en pratique, l'identification ne nécessite souvent que 5 à 7 caractères).
EXAMPLE dae86e1950b1277e45cee180551750029cfe735, dae86e

refs/heads/foo Nom complet d'une référence symbolique désignant respectivement la branche locale *foo*, le tag *bar*, la branche distante *baz* dans le dépôt distant *depot*.

foo Nom court d'une référence symbolique, que Git va résoudre en prenant le premier objet valide dans la liste suivante :

***GIT_DIR/foo** (gère le cas des références HEAD et *_HEAD)

refs/foo

refs/tags/foo

refs/heads/foo

refs/remotes/foo

refs/remotes/foo/HEAD

EXAMPLE master, heads/master, refs/heads/master, origin/master, remotes/origin/master, refs/remotes/origin/master

A^ Le premier commit parent du commit pointé par *A*.

A^n Le *n*ème parent du commit pointé par *A*. Pour des valeurs *n* > 1, cette syntaxe n'est valable que pour des commits fusionnant plusieurs branches.

A~n Le *n*ème commit en arrière dans l'historique. *A~1* désigne le même commit que *A^*, de même pour *A~2* et *A^2*, etc.

foo{[n]} Le commit sur lequel pointait *foo* en revenant *n* « changements » de *foo* en arrière (par « changement », on entend toute opération qui a fait pointer *foo* sur un commit différent). Si *foo* est omis, on utilise le nom de la branche courante. Ex : master@{2}, @{3}

REMARQUE Cette syntaxe utilise le *reflog* de git, une sauvegarde automatique (sur une durée configurable) des anciennes versions des branches présentes dans le dépôt.

foo{[date]} Le commit sur lequel pointait *foo* à la *date* (et heure) indiquée. La date peut s'exprimer de nombreuses manières différentes comme *yesterday, 3 days 1 hour ago* ou simplement *2012-02-09 15:32*. Ex : master@{1 week ago}, HEAD@{14:30}

REMARQUE Cette syntaxe utilise également le *reflog* de git, ce qui veut dire que cela renvoie l'état de la branche *locale* à la date indiquée, et non pas l'état de l'éventuelle branche distante associée.

A:fichier Le *fichier* de l'arborescence associée au commit pointé par *A*. Contrairement aux autres syntaxes présentées, cette syntaxe ne renvoie pas un objet de type *commit* mais un objet de type *blob*. S'utilise notamment avec **git show** qui sait afficher le contenu de n'importe quel objet, ou encore avec **git diff** qui sait comparer deux fichiers arbitraires.

REMARQUE Retrouvez toutes ces syntaxes dans la page de manuel **gitrevisions(7)**.

Aide syntaxique : treeish et intervalles

Syntaxe pour décrire des ensembles de commits sous forme d'intervalles (ranges)

A *A* et tous ses ancêtres.

REMARQUE Lorsque Git attend un intervalle, désigner un commit inclut automatiquement tous les commits anciêtres.

A ^B *A* et tous ses ancêtres, sauf *B* et ses ancêtres.

B..A

A...B *A* et *B* et tous leurs ancêtres sauf les ancêtres communs aux deux.

A! Le commit *A* tout seul.

L'arme ultime pour isoler le commit fautif : git bisect

Un bogue qu'on peut reproduire de manière fiable est un bogue à moitié corrigé, dit-on souvent. C'est d'autant plus vrai si ce bogue est une régression, car Git fournit une commande formidable pour identifier le commit fautif, **git bisect**, qui implémente une recherche par dichotomie. Il faut donc spécifier un ensemble de commits à tester en indiquant une version connue comme étant affectée par le problème (*bad*) et une version non affectée (*good*). Git extrait le commit médian de cet ensemble et attend votre verdict : cette version est-elle affectée ou non ? Partant de là, Git réduit de moitié l'espace des commits à tester. Après quelques itérations, Git indique le premier commit affecté par le problème.

git bisect start
git bisect start [A [B...]]

Démarre une recherche par dichotomie. Si *A* est précisé, il s'agit d'un « mauvais » commit (c'est-à-dire affecté par le problème). Si *B* est précisé, il s'agit d'un « bon » commit (non affecté par le problème).

Pour restreindre l'ensemble des commits à ceux qui modifient des fichiers, il est possible d'en indiquer les chemins (fichiers ou répertoires) après deux-tirets (-).

git bisect bad [A]

Enregistre *A* comme étant affecté par le problème, il s'agit donc d'une *bonne heure* à l'ensemble de commits à tester. Si *A* est omis, c'est HEAD qui est utilisé.

git bisect good [A]

Enregistre *A* comme étant non affecté par le problème, il s'agit donc d'une *bonne basse* à l'ensemble de commits à tester. Si *A* est omis, c'est HEAD qui est utilisé.

git bisect skip [A]

Enregistre *A* comme étant non testable. Si *A* est omis, c'est HEAD qui est utilisé.

ATTENTION Si le commit fautif est adjacent à un commit non testable, cela peut empêcher Git d'identifier.

git bisect reset

Termine ou interrompt la recherche par dichotomie. Le répertoire de travail et le dépôt sont remis dans leur état initial.

git bisect run commande

Automatisé la recherche à l'aide d'une commande externe. Pour chaque test, Git exécute *commande* et utilise le code de retour pour déterminer le statut du commit : 0, le commit est « bon » ; 1-127 (sauf 125), le commit est « mauvais » ; 125, le commit est non testable ; 128-255, une erreur s'est produite et le processus est interrompu.

git bisect visualize

Affiche l'ensemble des commits restant à tester dans **gitk**.