Git Magique

Ben Lynn

Août 2007

Préface

Git est un couteau suisse de la gestion de versions. Un outil de gestion de révisions multi-usage, pratique et fiable, dont la flexibilité en rend l'apprentissage pas si simple, sans parler de le maîtriser !

Comme Arthur C. Clarke le fait observer, toute technologie suffisamment avancée se confond avec la magie. C'est une approche intéressante pour Git : les débutants peuvent ignorer ses mécanismes internes et l'utiliser comme une baguette magique afin d'époustoufler les amis et rendre furieux les ennemis par ses fabuleuses capacités.

Plutôt que de rentrer dans le détails, nous donnons des instructions pour obtenir tel ou tel effet. À force d'utilisation, petit à petit, vous comprendrez comment fonctionne chaque truc et comment composer vos propres recettes pour répondre à vos besoins.

- Chinois (Simplifié) : par JunJie, Meng et JiangWei.
- Française : par Alexandre Garel, Paul Gaborit et Nicolas Deram. Hébergé aussi chez itaapy.
- Allemande : par Benjamin Bellee et Armin Stebich. Hébergé aussi sur le site web d'Armin.
- Italien : par Mattia Rigotti.
- Portugaise : par Leonardo Siqueira Rodrigues [version ODT].
- Russe: par Tikhon Tarnavsky, Mikhail Dymskov et d'autres.
- Espagnole : par Rodrigo Toledo et Ariset Llerena Tapia.
- Vietnamienne: par Trần Ngọc Quân. Hébergé aussi sur son site Web.
- sur une seule page web : HTML nu, sans CSS ;
- fichier PDF : version imprimable.

- package Debian, package Ubuntu : obtenez rapidement une copie locale de ce site. Pratique lorsque ce serveur est arrêté.
- un vrai livre [Amazon.com] : 64 pages, 15.24cm x 22.86cm, noir et blanc, en anglais. Pratique en cas de panne courant.

Merci !

Je reste modeste devant le travail fourni par tant de monde pour traduire ces pages. J'apprécie beaucoup d'élargir mon audience grâce aux efforts des personnes déjà citées.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin et Tyler Breisacher ont contribué aux corrections et aux améliorations.

François Marier maintient le paquet Debian, créé à l'origine par Daniel Baumarr.

Ma gratitude va également à beaucoup d'autres pour leurs encouragements et compliments. Je suis tenté de vous citer ici, toutefois ceci risquerait de porter vos attentes à des sommets ridicules.

Si par erreur je vous ai oublié, merci de me le signaler ou, plus simplement, envoyez-moi un patch !

- http://repo.or.cz/ héberge des projets libres. C'est le premier hébergement Git, fondé et maintenu par les premiers développeurs Git.
- http://gitorious.org/ est un autre site d'hébergement Git fait pour les projets Open-Source.
- http://github.com/ héberge des projets Open-Source gratuitement et des projets privés contre paiement.

Un grand merci à ces sites pour l'hébergement de ce guide.

Licence


```
$ git clone git://repo.or.cz/gitmagic.git  # Pour créer le dossier gitmagic
```

ou à partir d'un des miroirs :

```
$ git clone git://github.com/blynn/gitmagic.git
$ git clone git://gitorious.org/gitmagic/mainline.git
```

Introduction

Je vais me servir d'une analogie pour présenter la gestion de versions. Référez-vous à la page de wikipedia sur la gestion de versions pour une explication plus censée.

Le travail comme un jeu

J'ai joué à des jeux vidéos presque toute ma vie. Par contre, je n'ai commencé à utiliser des systèmes de gestion de versions qu'à l'âge adulte. Je pense ne pas être le seul dans ce cas et la comparaison entre les deux peut rendre les concepts plus simples à expliquer et à comprendre.

Pensez à l'édition de votre code, ou de votre document, comme s'il s'agissait de jouer à un jeu. Quand vous avez bien progressé, vous aimeriez faire une sauvegarde. Pour cela vous cliquez sur le bouton *enregistrer* de votre fidèle éditeur.

Mais ceci va écraser l'ancienne version. C'est comme ces anciens jeux qui n'avaient qu'un emplacement : oui vous pouviez faire une sauvegarde mais vous ne pouviez pas revenir dans un état précédent. Quel dommage, vu que votre sauvegarde précédente pouvait éventuellement être située à un passage du jeu particulièrement amusant sur lequel vous seriez bien revenu un de ces jours. Ou, encore pire, votre seule sauvegarde était dans un état qui ne permettait pas de gagner et vous deviez tout recommencer à zéro.

Gestion de versions

Lorsque vous modifiez un document, dans le but de conserver les anciennes versions, vous pouvez l'"Enregistrer Sous…" un nom de fichier différent ou le recopier ailleurs avant de l'enregistrer. Vous pouvez même compresser ces copies pour gagner de l'espace. C'est une forme primitive et laborieuse de gestion de versions. Les jeux vidéo se sont améliorés sur ce point depuis longtemps puisque la plupart proposent différents emplacements de sauvegarde automatiquement horodatés.

Rendons le problème légèrement plus coriace. Imaginez que vous ayez un ensemble de fichiers qui vont ensemble comme le code source d'un projet ou les fichiers d'un site web. Dans ce cas si vous voulez conserver une ancienne version, vous devez archiver le dossier en entier. Conserver un grand nombre de versions à la main n'est pas pratique et devient rapidement fastidieux.

Dans le cas de certains jeux vidéo, l'enregistrement d'une partie est réellement constitué d'un dossier rempli de fichiers. Ces jeux cachent ce détail au joueur et présentent une interface adaptée pour gérer différentes versions de ce dossier.

Les systèmes de gestion de versions ne font pas autre chose. Ils offrent tous une belle interface pour gérer un dossier rempli de plein de choses. Vous pouvez enregistrer l'état du dossier aussi souvent que vous voulez et, plus tard, vous pouvez recharger l'un des états enregistrés. À la différence de la plupart des jeux vidéos, ils sont généralement habiles pour économiser l'espace nécessaire. Typiquement, seuls quelques fichiers changent d'une version à une autre, et pas de beaucoup. Stocker ces différences au lieu des nouvelles copies complètes économise de l'espace.

Gestion distribuée

Imaginez maintenant un jeu vidéo très difficile. Si difficile à terminer que plein de joueurs expérimentés de toute la planète décident de faire équipe et de partager leurs parties enregistrées pour essayer d'en venir à bout. Les Speedruns en sont un exemple concret : des joueurs qui se spécialisent dans différents niveaux du même jeu collaborent pour produire des résultats surprenants.

Quel système mettriez-vous en place pour qu'ils puissent accéder facilement aux sauvegardes des uns et des autres ? Et pour qu'ils puissent en téléverser de nouvelles ?

Dans le passé, tous les projets utilisaient une gestion de versions centralisée. Quelque part un serveur contenait l'ensemble des sauvegardes du jeu et personne d'autre. Chaque joueur conservait au plus quelques sauvegardes de parties sur leur machine. Quand un joueur voulait progresser, il téléchargeait les dernières sauvegardes du serveur, jouait un moment, puis sauvegardait et téléversait ses nouvelles sauvegardes vers le serveur pour les mettre à disposition de tous les autres.

Qu'en était-il si pour une raison quelconque, des joueurs voulaient obtenir une partie enregistrée antérieurement ? Peut-être la sauvegarde actuelle de la partie était-elle dans un état sans possibilité de victoire parce que quelqu'un avait oublié de prendre un objet au niveau trois, et voulaient-ils retrouver la dernière partie enregistrée au moment où la partie pouvait encore être gagnée. Ou peut-être souhaitaient-ils comparer deux parties enregistrées précédemment pour voir le travail réalisé par un joueur précis.

Il peut y avoir de nombreuses raisons de vouloir récupérer une ancienne version, mais le résultat est le même : ils devaient demander au serveur central cette partie précédemment sauvegardée. Et plus ils voulaient de parties sauvegardées, plus ils devaient communiquer.

La nouvelle génération des systèmes de gestion de versions, dont Git fait partie, sont dits systèmes distribués et peuvent être vus comme une généralisation des systèmes centralisés. Quand les joueurs téléchargent du serveur principal ils obtiennent toutes les parties sauvegardées, pas uniquement la dernière. C'est comme s'ils faisaient un miroir du serveur central.

Cette opération initiale de clonage peut être coûteuse, surtout s'il y a un long historique, mais ça paie sur le long terme. Un avantage immédiat est que lorsqu'on désire une partie enregistrée, quelle qu'en soit la raison, aucune communication avec le serveur central n'est nécessaire.

Une superstition idiote

Un croyance populaire veut que les systèmes distribués ne soient pas adaptés aux projets qui ont besoin d'un dépôt central officiel. Rien n'est moins vrai. Photographier quelqu'un n'a jamais eu pour effet de voler son âme. De même, cloner le dépôt principal ne diminue pas son importance.

Une première approximation assez bonne est que tout ce qu'un système centralisé de gestion de versions peut faire, un système distribué bien conçu peut le faire en mieux. Les ressources réseau sont simplement plus coûteuses que les ressources locales. Bien que nous verrons plus loin qu'il peut y avoir quelques inconvénients à l'approche distribuée, il y a moins de risques de faire des comparaisons erronées en utilisant cette approximation.

Un petit projet peut ne nécessiter qu'une fraction des fonctionnalités offertes par un tel système, mais utiliser un système qui n'autorise pas les changements d'échelle pour les petits projets c'est comme utiliser les chiffres romains pour les calculs sur des petits nombres.

De plus, votre projet peut grossir au-delà de vos prévisions initiales. Utiliser Git même pour les cas simples est comparable au fait d'avoir sur soi un couteau Suisse que vous utilisez surtout pour déboucher des bouteilles. Le jour où vous avez besoin d'un tournevis vous êtes content d'avoir plus qu'un simple tire-bouchon.

Conflits fusionnels

Pour aborder ce sujet, notre analogie avec le jeu vidéo serait trop tirée par les cheveux. En revanche, revenons au cas de l'édition d'un document.

Imaginons que Alice insère une ligne au début d'un fichier, et que Bob en ajoute une à la fin de sa propre copie. Ils envoient tout les deux leurs modifications. La plupart des systèmes vont automatiquement déduire un traitement raisonnable des actions : accepter et fusionner leur modifications, ainsi les modifications de Alice et de Bob sont appliquées.

Maintenant imaginez que Alice et Bob ont fait des modifications différentes sur la même ligne. Il devient alors impossible de procéder sans intervention humaine. Celui qui envoie ses modifications en second est informé d'un *conflit de fusion* (*merge conflict*), et doit choisir l'une des deux versions de la ligne, ou revoir complètement cette ligne.

Des situations plus complexes peuvent se présenter. Les systèmes de gestion de versions s'occupent eux même des cas les plus simples, et laissent les cas difficiles aux humains. Généralement leur comportement est configurable.

Astuces de base

Plutôt que de plonger dans l'océan des commandes Git, utilisez ces commandes élémentaires pour commencer en vous trempant les pieds. Malgré leur simplicité, chacune d'elles est utile. En effet, lors de mon premier mois d'utilisation de Git, je ne me suis jamais aventuré au-delà de ce qui est exposé dans ce chapitre.

Enregistrer l'état courant

Vous êtes sur le point d'effectuer une opération drastique ? Avant de le faire, réalisez une capture de tous les fichiers du dossier courant :

```
$ git init
$ git add .
$ git commit -m "Ma première sauvegarde"
```

Si jamais votre opération tourne mal, vous pouvez retrouver votre version initiale, immaculée :

```
$ git reset --hard
```

Pour enregistrer un nouvel état :

```
$ git commit -a -m "Une autre sauvegarde"
```

Ajouter, supprimer, renommer

Les commandes ci-dessus ne font que garder traces des fichiers qui étaient présents lorsque vous avez executé **git add** pour la première fois. Si vous ajoutez de nouveaux fichiers ou sous-dossiers, il faut le signaler à Git :

```
$ git add readme.txt Documentation
```

De même, si vous voulez que Git oublie certains fichiers :

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

Git supprime les fichiers pour vous si vous ne l'avez pas encore fait.

Renommer un fichier revient à supprimer l'ancien nom et ajouter le nouveau. Il y a également le raccourci **git mv** qui a la même syntaxe que la commande mv. Par exemple :

```
$ git mv bug.c feature.c
```

Annuler/Reprendre avancé

Parfois vous voulez seulement revenir en arrière et oublier les modifications effectuées depuis un certain temps parce qu'elles sont toutes fausses. Dans ce cas :

```
$ git log
```

vous montre une liste des commits récents, accompagnés de leur empreinte SHA1 :

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date: Tue Mar 14 01:59:26 2000 -0800

    Remplacement de prinf() par write()

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Commit initial
```

Les premiers caractères de l'empreinte sont suffisants pour spécifier un commit ; ou alors, copiez et collez l'empreinte en entier. Saisissez :

```
$ git reset --hard 766f
```

pour restaurer l'état correspondant au commit donné et supprimer de manière permanente tous les commits plus récents de l'enregistrement.

Parfois vous ne voulez faire qu'un bref saut dans un état précédent. Dans ce cas, saisissez :

```
$ git checkout 82f5
```

Ceci vous ramène en arrière dans le temps, tout en conservant les commits récents. Toutefois, comme pour le voyage temporel de la science-fiction, si vous faites des modifications suivies d'un commit, vous entrerez dans une réalité parallèle puisque vos actions sont différentes de ce qu'elles étaient la première fois.

Cette réalité parallèle est appelée une *branche* (*branch*), et nous en dirons plus après. Pour le moment rappelez-vous simplement que :

```
$ git checkout master
```

vous ramènera dans le présent. De plus, pour éviter que Git se plaigne, réalisez toujours un commit ou un reset de vos modifications avant de faire un checkout.

Pour reprendre l'analogie du jeu vidéo :

- `git reset --hard` : recharge une ancienne sauvegarde et supprime toutes les sauvegardes plus récentes.
- `git checkout` : recharge une ancienne partie, mais si vous jouez avec, l'état de la partie va différer des enregistrement suivants que vous aviez réalisés la première fois. Chaque nouvelle sauvegarde sera sur une branche séparée représentant la réalité parallèle dans laquelle vous êtes entré. On s'en occupe plus loin.

Vous pouvez choisir de ne restaurer que certains fichiers et sous-dossiers en les nommant à la suite de la commande :

```
$ git checkout 82f5 un.fichier un-autre.fichier
```

Faites attention car cette forme de **checkout** peut écraser vos fichiers sans avertissement. Pour éviter les accidents, faites un commit avant toute commande checkout, surtout quand vous débutez avec Git. En général, quand vous n'êtes pas sûr des conséquences d'une opération, et pas seulement des commandes Git, faites d'abord un **git commit -a**.

Vous n'aimez pas copier et coller les empreintes ? Alors utilisez :

```
$ git checkout :/"Ma première s"
```

pour arriver sur le commit qui commence avec ce message. Vous pouvez aussi demander la cinquième sauvegarde en arrière :

```
$ git checkout master~5
```

Reprise (revert)

Dans une cour de justice, certains évènements peuvent être effacés du procès verbal. De même vous pouvez sélectionner des commits spécifiques à défaire :

```
$ git commit -a
$ git revert 1b6d
```

défera le dernier commit ayant cette empreinte. La reprise est enregistrée comme un nouveau commit, ce que vous pourrez constater en lançant un **git log**.

Génération du journal des modifications (changelog)

Certains projets demandent un changelog. Créez-le en tapant :

```
$ git log > ChangeLog
```

Télécharger des fichiers

Faites une copie d'un projet géré par Git en saisissant :

```
$ git clone git://serveur/chemin/vers/les/fichiers
```

Par exemple, pour récupérer les fichiers utilisés pour créer ce site :

```
$ git clone git://git.or.cz/gitmagic.git
```

Nous aurons beaucoup à dire sur la commande **clone** d'ici peu.

Le dernier cri

Si vous avez déjà téléchargé une copie d'un projet en utilisant **git clone**, vous pouvez la mettre à jour vers la dernière version avec :

```
$ git pull
```

Publication instantanée

Imaginez que vous avez écrit un script que vous voudriez partager avec d'autres. Vous pourriez leur dire de le télécharger de votre ordinateur, mais s'ils le font au moment où vous êtes en train d'améliorer le script ou d'y effectuer des modifications expérimentales, ils peuvent se retrouver dans la panade. Bien sûr, c'est pour cela qu'on a créé la publication de versions successives. Les développeurs peuvent travailler sur un projet fréquemment, mais ils ne rendent le code disponible quand lorsqu'ils le trouvent présentable.

Pour faire ça avec Git, dans le dossier qui contient votre script :

```
$ git init
$ git add .
$ git commit -m "Première publication"
```

Ensuite vous pouvez dire à vos utilisateurs de lancer :

```
$ git clone votre.ordinateur:/chemin/vers/le/script
```

pour télécharger votre script. En considérant qu'ils ont accès à votre ordinateur via ssh. Sinon, lancez **git daemon** et dites plutôt à vos utilisateurs de lancer :

```
$ git clone git://votre.ordinateur/chemin/vers/le/script
```

À partir de maintenant, chaque fois que votre script est prêt à être publié, exécutez :

```
$ git commit -a -m "Nouvelle version"
```

et vos utilisateurs peuvent mettre à jour leur version en allant dans leur dossier contenant votre script et en saisissant :

```
$ git pull
```

Vos utilisateurs ne se retrouveront jamais avec une version de votre script que vous ne vouliez pas leur montrer.

Qu'ai-je fait ?

Retrouvez les modifications faites depuis le dernier commit avec :

```
$ git diff
```

Ou depuis hier :

```
$ git diff "@{yesterday}"
```

Ou entre une version spécifique et la version deux commits en arrière :

```
$ git diff 1b6d "master~2"
```

Dans chacun de ces cas, la sortie est un **patch** (rustine) qui peut être appliqué en utilisant **git apply**. Vous pouvez aussi essayer :

```
$ git whatchanged --since="2 weeks ago"
```

Souvent je parcours plutôt l'historique avec qgit, pour sa pimpante interface photogénique, ou tig, une interface en mode texte qui fonctionne même sur les connexions lentes. Autrement, installez un serveur web, lancez **git instaweb** et dégainez n'importe quel navigateur internet.

Exercice

Soit A, B, C, D quatre commits successifs où B est identique à A à l'exception de quelques fichiers qui ont été supprimés. Nous voudrions remettre les fichiers en D. Comment faire ?

Il y a au moins trois solutions. En considérant que nous sommes à D :

1. La différence entre A et B sont les fichiers supprimés. Nous pouvons créer un patch représentant cette différence et l'appliquer :

   ```
   $ git diff B A | git apply
   ```

2. Vu que nous avions enregistré les fichiers en A, nous pouvons les reprendre :

   ```
   $ git checkout A foo.c bar.h
   ```

3. Nous pouvons aussi voir le chemin de A à B comme une modification à défaire :

   ```
   $ git revert B
   ```

Quel est le meilleur choix ? Celui que vous préférez. C'est facile d'obtenir ce que vous voulez avec Git, et souvent il y a plein de manières de le faire.

Clonons gaiement

Avec les anciens systèmes de gestion de versions, l'opération standard pour obtenir des fichiers est le checkout. Vous obtenez un ensemble de fichiers correspondant à un état particulier précédemment enregistré.

Avec Git et d'autres systèmes distribués de gestion de versions, le clonage est l'opération standard. Pour obtenir des fichiers, on crée un *clone* du dépôt entier. En d'autres termes, il s'agit de faire un miroir du serveur central. Tout ce qui peut se faire sur le dépôt central peut être fait sur le vôtre.

Synchronisation entre machines

Je peux imaginer faire des archives **tar** ou utiliser **rsync** pour des sauvegardes ou une synchronisation simple. Mais parfois j'édite sur mon portable, d'autres fois sur mon fixe, et les deux peuvent ne pas avoir communiqué entre temps.

Initialisez un dépôt Git et faites un **commit** de vos fichiers depuis une machine. Ensuite sur l'autre :

```
$ git clone autre.ordinateur:/chemin/vers/les/fichiers
```

pour créer une deuxième copie de ces fichiers et du dépôt Git.

À partir de ce moment,

```
$ git commit -a
$ git pull autre.ordinateur:/chemin/vers/les/fichiers HEAD
```

ira chercher l'état des fichiers sur l'autre ordinateur pour mettre à jour celui sur lequel vous travaillez. Si récemment vous avez fait des modifications d'un même fichier en conflit entre elles, Git vous le signalera et vous devrez répéter à nouveau le commit après avoir résolu ces conflits.

Gestion classique des sources

Initialisez le dépôt Git de vos fichiers :

```
$ git init
$ git add .
$ git commit -m "Commit initial"
```

Sur le serveur central, initialisez un *dépôt nu* (**bare** dans la terminologie Git) dans un dossier quelconque :

```
$ mkdir proj.git
$ cd proj.git
$ git init --bare
$  # variante en une ligne : GIT_DIR=proj.git git init
```

Si besoin, démarrez le démon (service) :

```
$ git daemon --detach  # peut être tourne-t-il déjà
```

Pour les services d'hébergement en ligne, suivez les instructions fournies pour mettre en place le dépôt Git initialement vide. En général il s'agit de remplir un formulaire sur une page web.

Poussez votre projet vers le serveur central en utilisant :

```
$ git push git://serveur.central/chemin/du/proj.git HEAD
```

Pour obtenir les sources, un développeur saisit :

```
$ git clone git://serveur.central/chemin/du/proj.git
```

Après avoir fait des modifications, le développeur les enregistre en local :

```
$ git commit -a
```

Pour se mettre à jour par rapport à la dernière version :

```
$ git pull
```

Tout conflit lors de la fusion doit être résolu puis validé :

```
$ git commit -a
```

Pour envoyer les modifications locales vers le dépôt central :

```
$ git push
```

Si le serveur principal a de nouvelles modifications dues à d'autres développeurs, l'envoi échoue et le développeur doit se mettre à jour de la dernière version, résoudre les éventuels conflits de fusion, puis essayer à nouveau.

Dépôts nus

Un dépôt nu (**bare repository**) est nommé ainsi car il n'a pas de dossier de travail : il ne contient que des fichiers qui sont normalement cachés dans le sous dossier `.git`. En d'autres termes, il ne conserve que l'historique d'un projet et ne contient jamais le rendu d'une version donnée.

Un dépôt nu joue un rôle similaire à celui du serveur principal dans un système de gestion de versions centralisé : le réceptacle de vos projets. Les développeurs clonent vos projets à partir de celui-ci et y poussent les dernières modifications officielles. En général il est placé sur un serveur qui ne fait quasiment que ce

travail de distribution de l'information. Le développement s'opère sur les clones de sorte que le dépôt principal peut se passer d'un dossier de travail.

Beaucoup des commandes de Git échouent sur un dépôt nu tant que la variable d'environnement `GIT_DIR` n'est pas renseignée avec le chemin vers le dépôt ou que l'option `--bare` n'est pas utilisée.

Envoi vs rapatriement (push vs pull)

Pourquoi a-t-on introduit la commande `push` (*pousser* ou *envoyer*) au lieu de se contenter de la commande `pull` (*tirer* ou *rapatrier*) plus familière ? Premièrement, la commande `pull` échoue sur un dépôt nu : il faut y utiliser la commande `fetch` dont nous parlerons plus tard. Mais même si nous conservions un dépôt standard sur le serveur central, y rapatrier les modifications serait peu pratique. Nous devrions d'abord nous connecter au serveur et donner en argument à la commande `pull` l'adresse de la machine depuis laquelle nous souhaitons rapatrier des modifications. Des pare-feux peuvent éventuellement nous embêter, et comment faire si nous n'avons pas d'accès `shell` au serveur ?

Quoi qu'il en soit, ce cas mis à part, nous décourageons l'envoi (en comparaison du rapatriement) parce que cela peut entraîner des confusions lorsque la destination possède un dossier de travail.

En résumé, pendant votre phase d'apprentissage de Git, n'utilisez l'envoi (push) que lorsque la destination est un dépôt nu ; sinon rapatriez (pull).

Forker un projet

Vous en avez marre de la manière dont est géré un projet ? Vous pensez pouvoir faire mieux ? Dans ce cas, sur votre serveur :

```
$ git clone git://serveur.principal/chemin/vers/les/fichiers
```

Ensuite, informez tout le monde du fork de ce projet sur votre serveur.

Par la suite, vous pouvez fusionner les modifications venant du projet originel grâce à :

```
$ git pull
```

Système ultime de sauvegarde

Vous voulez des archives redondantes et géographiquement distribuées, permettant de faire face à un désastre ? Si votre projet a beaucoup de développeurs, ne faites rien ! Chaque clone de votre code est de fait une sauvegarde. Non seulement de l'état actuel, mais de l'historique complet. Grâce aux empreintes

cryptographiques, si le clone de quelqu'un est corrompu, il sera repéré dès qu'il tentera de communiquer avec d'autres.

Si votre projet n'est pas si populaire, trouvez autant de serveurs que possible afin d'héberger vos clones.

Le vrai paranoïaque devrait toujours noter la dernière empreinte SHA1 de 20 octets du HEAD dans un endroit sûr. Ce doit être sûr, pas privé. Par exemple, la publier dans un quotidien marcherait bien, parce qu'il est difficile de réaliser une attaque modifiant l'ensemble des exemplaires d'un journal.

Le multi-tâche à la vitesse de la lumière

Imaginons que vous souhaitiez travailler sur plusieurs fonctionnalités en parallèle. Dans ce cas validez (`commit`) votre projet et lancez :

```
$ git clone . /un/nouveau/dossier
```

Grâce aux liens matériels, les clones locaux sont créés plus rapidement et occupent moins de place que de simples copies.

Vous pouvez maintenant travailler simultanément sur deux fonctionnalités indépendantes. Par exemple vous pouvez modifier l'un des clones pendant que l'autre est en cours de compilation. À tout moment vous pouvez valider (`commit`) vos modifications puis rapatrier (`pull`) les modifications depuis l'autre clone.

```
$ git pull /mon/autre/clone HEAD
```

Guérilla de la gestion de versions

Alors que vous travaillez sur un projet qui utilise un autre système de gestion de versions, Git vous manque ? Dans ce cas, initialisez un dépôt Git dans votre dossier de travail.

```
$ git init
$ git add .
$ git commit -m "Commit initial"
```

puis clonez-le :

```
$ git clone . /un/nouveau/dossier
```

Allez ensuite dans le nouveau dossier et travaillez plutôt là, utilisant Git comme vous le voulez. De temps à autre, quand vous voulez vous synchroniser avec les autres, rendez-vous dans le dossier de départ, synchronisez-le en utilisant l'autre système de gestion de version, puis saisissez :

```
$ git add .
$ git commit -m "Synchro avec les autres"
```

Ensuite allez dans le nouveau dossier et lancez :

```
$ git commit -a -m "Description de mes modifications"
$ git pull
```

La procédure pour partager vos modifications avec les autres dépend de l'autre système de gestion de versions. Le nouveau dossier contient les fichiers avec vos modifications. Lancez toute commande de l'autre système de gestion de versions nécessaire pour les envoyer au dépôt central.

Subversion, qui est peut être le meilleur système de gestion de versions centralisé est utilisé par d'innombrables projets. La commande **git svn** automatise la procédure ci-dessus pour les dépôts Subversion, et peut aussi être utilisée pour exporter un projet Git vers un dépôt Subversion.

Mercurial

Mercurial est un système de gestion de versions similaire qui peut travailler quasiment sans heurt avec Git. Avec le plugin `hg-git` un utilisateur de Mercurial peut, sans rien perdre, envoyer (push) vers ou rapatrier (pull) depuis un dossier Git.

Téléchargez le plugin `hg-git` avec Git :

```
$ git clone git://github.com/schacon/hg-git.git
```

ou Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

Malheureusement, il ne semble pas y avoir de plugin analogue pour Git. Pour cette raison, il semble préférable d'utiliser Git plutôt que Mercurial pour le dépôt principal. Avec un dépôt Mercurial, il faut généralement un volontaire qui maintienne un dépôt Git en parallèle alors que, grâce au plugin `hg-git`, un dépôt Git fait l'affaire même pour les utilisateurs de Mercurial.

Bien que ce plugin puisse convertir un dépôt Mercurial en un dépôt Git en le poussant dans un dépôt vide, cette tâche est plus simple avec le script `hg-fast-export-git`, disponible via :

```
$ git clone git://repo.or.cz/fast-export.git
```

Pour faire une conversion, dans un nouveau dossier :

```
$ git init
$ hg-fast-export.sh -r /depot/hg
```

ceci après avoir ajouté le script à votre `$PATH`.

Bazaar

Nous allons rapidement évoquer Bazaar parce que c'est le système de gestion de versions distribué libre le plus populaire après Git et Mercurial.

Bazaar à l'avantage d'avoir plus de recul, étant donné qu'il est relativement jeune ; ses concepteurs ont pu tirer les leçons du passé et éviter des petits écueils historiques. De plus ses développeurs ont le souci de la portabilité et de l'interopérabilité avec les autres systèmes de gestion de versions.

Un plugin `bzr-git` permet aux utilisateurs de Bazaar de travailler avec les dépôts Git dans une certaine mesure, et permet de le faire de manière incrémentale, tandis que `bzr-fast-export` est fait pour les conversions uniques.

Pourquoi j'utilise Git

Au départ j'ai choisi Git parce que j'ai entendu qu'il gérait l'inimaginablement ingérable source du noyaux Linux. Je n'ai jamais ressenti le besoin d'en changer. Git m'a rendu de fiers services et je ne me suis toujours pas heurté à ses limites. Comme j'utilise surtout Linux, les éventuels problèmes sur d'autres plateformes n'entrent pas en ligne de compte.

De plus je préfère les programmes C et les scripts bash aux exécutables comme les scripts Pythons : il y a moins de dépendances et je suis accro aux temps d'exécution rapides.

J'ai réfléchi à la manière d'améliorer Git, allant jusqu'à écrire mes propres outils de type Git, mais uniquement à des fins de recherche. Même si j'avais terminé ce projet j'aurais tout de même continué avec Git vu que les avantages sont trop peu significatifs pour justifier l'utilisation d'un système farfelu.

Bien sur, vos besoins et envies diffèrent sûrement et vous pouvez très bien vous trouver mieux avec un autre système. Néanmoins vous ne pouvez pas faire une grosse erreur en choisissant Git.

La sorcellerie des branches

Des branchements et des fusions quasi-instantanés sont les fonctionnalités les plus puissantes qui font de Git un vrai tueur.

Problème : des facteurs externes amènent nécessairement à des changements de contexte. Un gros bug se manifeste sans avertissement dans la version déployée. La date limite pour une fonctionnalité particulière est avancée. Un développeur qui vous aidait pour une partie clé du projet n'est plus disponible. Bref, en tous cas, vous devez brusquement arrêter la tâche en cours pour vous focaliser sur une tâche tout autre.

Interrompre votre réflexion peut être nuisible à votre productivité et le changement de contexte amène encore plus de perte. Avec un système de gestion de versions centralisé, il faudrait télécharger une nouvelle copie de travail depuis le serveur central. Un système de gestion de versions décentralisé est bien meilleur puisqu'il peut cloner localement la version voulue.

Mais un clone implique encore la copie de tout le dossier de travail ainsi que de l'historique complet jusqu'au point voulu. Même si Git réduit ce coût grâce aux fichiers partagés et au liens matériels, les fichiers du projet doivent tout de même être entièrement recréés dans le nouveau dossier de travail.

Solution : dans ce genre de situations, Git offre un outil bien meilleur puisque plus rapide et moins consommateur d'espace disque : les branches.

Grâce à un mot magique, les fichiers de votre dossier se transforment d'une version à une autre. Cette transformation peut être bien plus qu'un simple voyage dans l'historique. Vos fichiers peuvent se transformer de la dernière version stable vers une version expérimentale, vers la version courante de développement, vers la version d'un collègue, etc.

La touche du chef

N'avez-vous jamais joué à l'un de ces jeux qui, à l'appui d'une touche particulière (la «touche du chef»), affiche instantanément une feuille de calcul ? Ceci vous permet de cacher votre écran de jeu dès que le chef arrive.

Dans un dossier vide :

```
$ echo "Je suis plus intelligent que mon chef." > myfile.txt
$ git init
$ git add .
$ git commit -m "Commit initial"
```

Vous venez de créer un dépôt Git qui gère un fichier contenant un message. Maintenant tapez :

```
$ git checkout -b chef  # rien ne semble avoir changé
$ echo "Mon chef est plus intelligent que moi." > myfile.txt
$ git commit -a -m "Un autre commit"
```

Tout se présente comme si vous aviez réécrit votre fichier et intégrer (commit) ce changement. Mais ce n'est qu'une illusion. Tapez :

```
$ git checkout master  # bascule vers la version originale du fichier
```

et ça y est ! Le fichier texte est restauré. Et si le chef repasse pour regarder votre dossier, tapez :

```
$ git checkout chef  # bascule vers la version visible par le chef
```

Vous pouvez basculer entre ces deux versions autant de fois que voulu, et intégrer (commit) vos changements à chacune d'elles indépendamment.

Travail temporaire

Supposons que vous travailliez sur une fonctionnalité et que, pour une raison quelconque, vous ayez besoin de revenir trois versions en arrière afin d'ajouter temporairement quelques instructions d'affichage pour voir comment quelque chose fonctionne. Faites :

```
$ git commit -a
$ git checkout HEAD~3
```

Maintenant vous pouvez ajouter votre code temporaire là où vous le souhaitez. Vous pouvez même intégrer (commit) vos changements. Lorsque vous avez terminé, tapez :

```
$ git checkout master
```

pour retourner à votre travail d'origine. Notez que tous les changement non intégrés sont définitivement perdus (NdT : les changements intégrés via commit sont conservés quelques jours et sont accessibles en connaissant leur empreinte SHA1).

Que faire si vous voulez nommer ces changements temporaires ? Rien de plus simple :

```
$ git checkout -b temporaire
```

et faites un commit avant de rebasculer vers la branche master. Lorsque vous souhaitez revenir à vos changements temporaires, tapez simplement :

```
$ git checkout temporaire
```

Nous aborderons la commande *checkout* plus en détail lorsque nous parlerons du chargement d'anciens états. Mais nous pouvons tout de même en dire quelques mots : les fichiers sont bien amenés dans l'état demandé mais en quittant la branche master. À ce moment, tout commit poussera nos fichiers sur une route différente, qui pourra être nommée plus tard.

En d'autres termes, après un checkout vers un état ancien, Git nous place automatiquement dans une nouvelle branche anonyme qui pourra être nommée et enregistrée grâce à **git checkout -b**.

Corrections rapides

Vous travaillez sur une tâche particulière et on vous demande de tout laisser tomber pour corriger un nouveau bug découvert dans la version '1b6d…' :

```
$ git commit -a
$ git checkout -b correction 1b6d
```

Puis quand vous avez corrigé le bug, saisissez :

```
$ git commit -a -m "Bug corrigé"
$ git checkout master
```

pour vous ramener à votre tâche originale. Vous pouvez même fusionner (*merge*) avec la correction de bug toute fraîche :

```
$ git merge correction
```

Fusionner

Dans certains systèmes de gestion de versions, la création de branches est facile mais les fusionner est difficile. Avec Git, la fusion est si simple que vous n'y prêterez plus attention.

En fait, nous avons déjà rencontré la fusion. La commande **pull** ramène (*fetch*) une série de versions puis les fusionne (*merge*) dans votre branche courante. Si vous n'avez effectué aucun changement local alors la fusion est un simple bon en avant (un *fast forward*), un cas dégénéré qui s'apparente au rapatriement de la dernière version dans un système de gestion de versions centralisé. Si vous avez effectué des changements locaux, Git les fusionnera automatiquement et préviendra s'il y a des conflits.

Habituellement, une version à une seule *version parente*, qu'on appelle la version précédente. Une fusion de branches entre elles produit une version avec plusieurs parents. Ce qui pose la question suivante : à quelle version se réfère 'HEAD~10' ? Puisqu'une version peut avoir plusieurs parents, par quel parent remonterons-nous ?

Il s'avère que cette notation choisit toujours le premier parent. C'est souhaitable puisque la branche courante devient le premier parent lors d'une fusion. Nous nous intéressons plus fréquemment aux changements que nous avons faits dans la branche courante qu'à ceux fusionnés depuis d'autres branches.

Vous pouvez choisir un parent spécifique grâce à l'accent circonflexe. Voici, par exemple, comment voir le log depuis le deuxième parent :

```
$ git log HEAD^2
```

Vous pouvez omettre le numéro pour le premier parent. Voici, par exemple, comment voir les différences avec le premier parent ;

```
$ git diff HEAD^
```

Vous pouvez combiner cette notation avec les autres. Par exemple :

```
$ git checkout 1b6d^^2~10 -b ancien
```

démarre la nouvelle branche «ancien» dans l'état correspondant à 10 versions en arrière du deuxième parent du premier parent de la version 1b6d.

Workflow sans interruption

La plupart du temps dans un projet de réalisation matérielle, la seconde étape du plan ne peut commencer que lorsque la première étape est terminée. Une voiture en réparation reste bloquée au garage jusqu'à la livraison d'une pièce. Le montage d'un prototype est suspendu en attendant la fabrication d'une puce.

Les projets logiciels peuvent être similaires. La deuxième partie d'une nouvelle fonctionnalité doit attendre que la première partie soit sortie et testée. Certains projets exigent une validation de votre code avant son acceptation, vous êtes donc obligé d'attendre que la première partie soit validée avant de commencer la seconde.

Grâce aux branches et aux fusions faciles, vous pouvez contourner les règles et travailler sur la partie 2 avant que la partie 1 soit officiellement prête. Supposons que vous ayez terminé la version correspondant à la partie 1 et que vous l'ayez envoyée pour validation. Supposons aussi que vous soyez dans la branche `master`. Alors, branchez-vous :

```
$ git checkout -b part2
```

Ensuite, travaillez sur la partie 2 et intégrez (via `commit`) vos changements autant que nécessaire. L'erreur étant humaine, vous voudrez parfois revenir en arrière pour effectuer des corrections dans la partie 1. Évidemment, si vous êtes chanceux ou très bon, vous pouvez sauter ce passage.

```
$ git checkout master  # Retour à la partie 1
$ correction_des_bugs
$ git commit -a        # Intégration de la correction
$ git checkout part2   # Retour à la partie 2
$ git merge master     # Fusion de la correction.
```

Finalement, la partie 1 est validée.

```
$ git checkout master    # Retour à la partie 1
$ diffusion des fichiers # Diffusion au reste du monde !
$ git merge part2        # Fusion de la partie 2
$ git branch -d part2    # Suppression de la branche 'part2'.
```

À cet instant vous êtes à nouveau dans la branche `master` avec la partie 2 dans votre dossier de travail.

Il est facile d'étendre cette astuce à de nombreuses branches. Il est aussi facile de créer une branche rétroactivement : imaginons qu'après 7 commits, vous vous rendiez compte que vous auriez dû créer une branche. Tapez alors :

```
$ git branch -m master part2  # Renommer la branche "master" en "part2".
$ git branch master HEAD~7   # Recréer une branche "master" 7 commits en arrière.
```

La branche `master` contient alors uniquement la partie 1 et la branche `part2` contient le reste ; nous avons créé `master` sans basculer vers elle car nous souhaitons continuer à travailler sur `part2`. Ce n'est pas très courant. Jusqu'à présent nous avions toujours basculé vers une branche dès sa création, comme dans :

```
$ git checkout HEAD~7 -b master  # Créer une branche et basculer vers elle.
```

Réorganiser le foutoir

Peut-être aimez-vous travailler sur tous les aspects d'un projet dans la même branche. Vous souhaitez que votre travail en cours ne soit accessible qu'à vous-même et donc que les autres ne puissent voir vos versions que lorsqu'elles sont proprement organisées. Commencez par créer deux branches :

```
$ git branch propre      # Créer une branche pour les versions propres
$ git checkout -b foutoir # Créer et basculer vers une branche pour le foutoir
```

Ensuite, faites tout ce que vous voulez : corriger des bugs, ajouter des fonctionnalités, ajouter du code temporaire et faites-en des versions autant que voulu. Puis :

```
$ git checkout propre
$ git cherry-pick foutoir^^
```

applique les modifications de la version grand-mère de la version courante du «foutoir» à la branche «propre». Avec les cherry-picks appropriés vous pouvez construire une branche qui ne contient que le code permanent et où toutes les modifications qui marchent ensemble sont regroupées.

Gestion des branches

Pour lister toutes les branches, tapez :

```
$ git branch
```

Par défaut, vous commencez sur la branche nommée «master». Certains préconisent de laisser la branche «master» telle quelle et de créer de nouvelles branches pour vos propres modifications.

Les options **-d** et **-m** vous permettent de supprimer et renommer les branches. Voir **git help branch**.

La branche «master» est une convention utile. Les autres supposent que votre dépôt possède une telle branche et qu'elle contient la version officielle de votre projet. Bien qu'il soit possible de renommer ou d'effacer cette branche «master», il peut-être utile de respecter les traditions.

Les branches temporaires

Après un certain temps d'utilisation, vous vous apercevrez que vous créez fréquemment des branches éphémères toujours pour les mêmes raisons : elles vous servent juste à sauvegarder l'état courant, vous permettant ainsi de revenir momentanément à état précédent pour corriger un bug.

C'est exactement comme si vous zappiez entre deux chaînes de télévision. Mais au lieu de presser deux boutons, il vous faut créer, basculer, fusionner et supprimer des branches temporaires. Par chance, Git propose un raccourci qui est aussi pratique que la télécommande de votre télévision :

```
$ git stash
```

Cela mémorise l'état courant dans un emplacement temporaire (un *stash*) et restaure l'état précédent. Votre dossier courant apparaît alors exactement comme il était avant que vous ne commenciez à faire des modifications et vous pouvez corriger des bugs, aller rechercher (pull) une modification de dépôt central ou toute autre chose. Lorsque vous souhaitez revenir à l'état mémorisé dans votre *stash*, tapez :

```
$ git stash apply  # Peut-être faudra-t-il résoudre quelques conflits.
```

Vous pouvez avoir plusieurs *stash* et les manipuler de différents manières. Voir **git help stash**. Comme vous l'aurez deviné, pour faire ces tours de magie, dans les coulisses Git gère des branches.

Travailler comme il vous chante

Vous vous demandez sans doute si l'usage des branches en vaut la peine. Après tout, des clones sont tout aussi rapides et vous pouvez basculer de l'un à l'autre par un simple **cd** au lieu de commandes Git ésotériques.

Considérez les navigateurs Web. Pourquoi proposer plusieurs onglets ainsi que plusieurs fenêtres ? Parce proposer les deux permet de s'adapter à une large gamme d'utilisations. Certains préfèrent n'avoir qu'une seule fenêtre avec plein d'onglets. D'autres font tout le contraire : plein de fenêtres avec un seul onglet. D'autres encore mélangent un peu des deux.

Les branches ressemblent à des onglets de votre dossier de travail et les clones ressemblent aux différents fenêtres de votre navigateur. Ces opérations sont toutes rapides et locales. Alors expérimentez pour trouver la combinaison qui vous convient. Git vous laisse travailler exactement comme vous le souhaitez.

Les leçons de l'histoire

L'une des conséquences de la nature distribuée de Git est qu'il est facile de modifier l'historique. Mais si vous réécrivez le passé, faites attention : ne modifiez que la partie de l'historique que vous êtes le seul à posséder. Sinon, comme des nations qui se battent éternellement pour savoir qui a commis telle ou telle atrocité, si quelqu'un d'autre possède un clone dont l'historique diffère du vôtre, vous aurez des difficultés à vous réconcilier lorsque vous interagirez.

Certains développeurs insistent très fortement pour que l'historique soit considéré comme immuable. D'autres pensent au contraire que les historiques doivent être rendus présentables avant d'être présentés publiquement. Git s'accommode des deux points de vue. Comme les clones, les branches et les fusions, la réécriture de l'historique est juste un pouvoir supplémentaire que vous donne Git. C'est à vous de l'utiliser à bon escient.

Je me corrige...

Que faire si vous avez fait un commit mais que vous souhaitez y attacher un message différent ? Pour modifier le dernier message, tapez :

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

Vous apercevez-vous que vous avez oublié un fichier ? Faites **git add** pour l'ajouter puis exécutez la commande ci-dessus.

Voulez-vous ajouter quelques modifications supplémentaires au dernier commit ? Faites ces modifications puis exécutez :

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

... et bien plus

Supposons que le problème précédent est dix fois pire. Après une longue séance, vous avez effectué une série de commits. Mais vous n'êtes pas satisfait de la manière dont ils sont organisés et certains des messages associés doivent être revus. Tapez alors :

```
$ git rebase -i HEAD~10
```

et les dix derniers commits apparaissent dans votre $EDITOR favori. Voici un petit extrait :

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Ensuite :

- Supprimez un commit en supprimant sa ligne.
- Réordonnez des commits en réordonnant leurs lignes.
- Remplacez `pick` par :
  - `edit` pour marquer ce commit pour amendement.
  - `reword` pour modifier le message associé.
  - `squash` pour fusionner ce commit avec le précédent.
  - `fixup` pour fusionner ce commit avec le précédent en supprimant le message associé.

Sauvegardez et quittez. Si vous avez marqué un commit pour amendement alors tapez :

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

Sinon, tapez :

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

Donc faites des commits très tôt et faites-en souvent : vous pourrez tout ranger plus tard grâce à *rebase*.

Les changements locaux en dernier

Vous travaillez sur un projet actif. Vous faites quelques commits locaux puis vous vous resynchronisez avec le dépôt officiel grâce à une fusion (merge). Ce cycle se répète jusqu'au moment où vous êtes prêt à pousser vos contributions vers le dépôt central.

Mais à cet instant l'historique de votre clone Git local est un fouillis infâme mélangeant les modifications officielles et les vôtres. Vous préféreriez que toutes vos modifications soient contiguës et se situent après toutes les modifications officielles.

C'est un boulot pour **git rebase** comme décrit ci-dessus. Dans la plupart des cas, vous pouvez utilisez l'option **--onto** et éviter les interactions.

Lisez **git help rebase** pour des exemples détaillés sur cette merveilleuse commande. Vous pouvez scinder des commits. Vous pouvez même réarranger des branches de l'arbre.

Réécriture de l'histoire

De temps en temps, vous avez besoin de faire des modifications équivalentes à la suppression d'une personne d'une photo officielle, la gommant ainsi de l'histoire d'une manière quasi Stalinienne. Supposons que vous ayez publié un projet mais

en y intégrant un fichier que vous auriez dû conserver secret. Par exemple, vous avez accidentellement ajouté un fichier texte contenant votre numéro de carte de crédit. Supprimer ce fichier n'est pas suffisant puisqu'il pourra encore être retrouvé via d'anciennes versions du projet. Vous devez supprimer ce fichier dans toutes les versions :

```
$ git filter-branch --tree-filter 'rm top/secret/fichier' HEAD
```

La documentation **git help filter-branch** explique cette exemple et donne une méthode plus rapide. De manière générale, **filter-branch** vous permet de modifier des pans entiers de votre historique grâce à une seule commande.

Après cela, le dossier `.git/refs/original` contiendra l'état de votre dépôt avant l'opération. Vérifiez que la commande filter-branch a bien fait ce que vous souhaitiez puis effacer ce dossier si vous voulez appliquer d'autres commandes filter-branch.

Finalement, remplacez tous les clones de votre projet par votre version révisée si vous voulez pouvoir interagir avec eux plus tard.

Faire l'histoire

Voulez-vous faire migrer un projet vers Git ? S'il est géré par l'un des systèmes bien connus alors il y a de grandes chances que quelqu'un ait déjà écrit un script afin d'importer l'ensemble de l'historique dans Git.

Sinon, regarder du côté de **git fast-import** qui lit un fichier texte dans un format spécifique pour créer un historique Git à partir de rien. Typiquement un script utilisant cette commande est un script jetable qui ne servira qu'une seule fois pour migrer le projet d'un seul coup.

À titre d'exemple, collez le texte suivant dans un fichier temporaire (`/tmp/historique`) :

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Commit initial
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
```

```
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Remplacement de printf() par write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

Puis créez un dépôt Git à partir de ce fichier temporaire en tapant :

```
$ mkdir projet; cd projet; git init
$ git fast-import --date-format=rfc2822 < /tmp/historique
```

Vous pouvez extraire la dernière version de ce projet avec :

```
$ git checkout master .
```

La commande **git fast-export** peut convertir n'importe quel dépôt Git en un fichier au format **git fast-import** ce qui vous permet de l'étudier pour écrire des scripts d'exportation mais vous permet aussi de transporter un dépôt dans un format lisible. Ces commandes permettent aussi d'envoyer un dépôt via un canal qui n'accepte que du texte pur.

Qu'est-ce qui a tout cassé ?

Vous venez tout juste de découvrir un bug dans une fonctionnalité de votre programme et pourtant vous êtes sûr qu'elle fonctionnait encore parfaitement il y a quelques mois. Zut ! D'où provient ce bug ? Si seulement vous aviez testé cette fonctionnalité pendant vos développements.

Mais il est trop tard. En revanche, en supposant que vous avez fait des commits suffisamment souvent, Git peut cerner le problème.

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git extrait un état à mi-chemin entre ces deux versions (HEAD et 1b6d). Testez la fonctionnalité et si le bug se manifeste :

```
$ git bisect bad
```

Si elle ne se manifeste pas, remplacer "bad" (mauvais) par "good" (bon). Git vous transporte à nouveau dans un état à mi-chemin entre la bonne et la mauvaise version, en réduisant ainsi les possibilités. Après quelques itérations, cette recherche dichotomique vous amènera au commit où le bug est survenu. Une fois vos investigations terminées, retourner à votre état original en tapant :

```
$ git bisect reset
```

Au lieu de tester chaque état à la main, automatisez la recherche en tapant :

```
$ git bisect run mon_script
```

Git utilise la valeur de retour du script fourni pour décider si un état est bon ou mauvais : mon_script doit retourner 0 si l'état courant est ok, 125 si cet état doit être sauté et n'importe quelle valeur entre 1 et 127 si l'état est mauvais. Une valeur négative abandonne la commande bisect.

Vous pouvez faire bien plus : la page d'aide explique comment visualiser les bisects, comment examiner ou rejouer le log d'un bisect et comment éliminer des changements que vous savez sans conséquence afin d'accélérer la recherche.

Qui a tout cassé ?

Comme de nombreux systèmes de gestion de versions, Git a sa commande blame :

```
$ git blame bug.c
```

Cette commande annote chaque ligne du fichier afin de montrer par qui et quand elle a été modifiée la dernière fois. À l'inverse de la plupart des autres systèmes, cette commande marche hors-ligne et ne lit que le disque local.

Expérience personnelle

Avec un système de gestion de versions centralisé, la modification de l'historique est une opération difficile et faisable uniquement par les administrateurs. Créer un clone, créer une branche ou en fusionner plusieurs sont des opérations impossibles à réaliser sans communication réseau. Il en est de même pour certains opérations basiques telles que parcourir l'historique ou intégrer une modification. Avec certains systèmes, des communications réseaux sont même nécessaires juste pour voir ses propres modifications ou pour ouvrir un fichier avec le droit de modification.

Ces systèmes centralisés empêchent le travail hors-ligne et nécessitent une infrastructure réseau d'autant plus lourde que le nombre de développeurs augmentent. Plus important encore, certaines opérations deviennent si lentes que les utilisateurs les évitent à moins qu'elles soient absolument indispensables. Dans les cas extrêmes cela devient vrai même pour les commandes les plus basiques. Lorsque les utilisateurs doivent effectuer des opérations lentes, la productivité souffre des interruptions répétées.

J'ai moi-même vécu ce phénomène. Git a été le premier système de gestion de versions que j'ai utilisé. Je me suis vite accoutumé à lui, tenant la plupart de ses fonctionnalités pour acquises. Je pensais que les autres systèmes étaient similaires : le choix d'un système de gestion de versions ne devait pas être bien différent du choix d'un éditeur de texte ou d'un navigateur web.

J'ai été très surpris lorsque, plus tard, il m'a fallu utilisé un système centralisé. Une liaison internet épisodique importe peu avec Git mais rend le développement quasi impossible lorsque le système exige qu'elle soit aussi fiable que les accès au disque local. De plus, je me restreignais afin d'éviter certaines commandes trop longues, ce qui m'empêchait de suivre ma méthode de travail habituelle.

Lorsqu'il me fallait utiliser ces commandes lentes, cela interrompait mes réflexions et avait des effets pervers. En attendant la fin des communications avec le serveur, je me lançais dans autre chose pour passer le temps comme lire mes mails ou écrire de la documentation. Lorsque je revenais à mon travail initial, la commande s'était terminée depuis longtemps et je perdais du temps à retrouver le fil de mes pensées. Les être humains ne sont pas bons pour changer de contexte.

Il y a aussi un effet intéressant du type « tragédie des biens communs » : afin d'anticiper la congestion du réseau, certains vont consommer plus de bandes passantes que nécessaire pour effectuer des opérations visant à réduire leurs attentes futures. Ces efforts combinés vont encore augmenter la congestion, incitant ces personnes à consommer encore plus de bande passante pour éviter ces délais toujours plus longs.

Git multijoueur

Au départ, j'utilisais Git pour un projet privé où j'étais le seul développeur. Parmi toutes les commandes liées à la nature distribuée de Git, je n'avais besoin que de **pull** et **clone** afin de disposer de mon projet en différents lieux.

Plus tard, j'ai voulu publier mon code via Git et inclure des modifications de plusieurs contributeurs. J'ai dû apprendre à gérer des projets avec de nombreux développeurs à travers le monde. Heureusement c'est l'un des points forts de Git et peut-être même sa *raison d'être* (en français dans le texte).

Qui suis-je ?

À chaque commit sont associés le nom et le mail de l'auteur, ceux qui sont montrés par **git log**. Par défaut, Git utilise les valeurs fournies par le système pour remplir ces champs. Pour les configurer explicitement, tapez :

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Supprimer l'option `--global` pour que ces valeurs soient locales au dépôt courant.

Git via SSH et HTTP

Supposez que vous ayez un accès SSH à un serveur Web sur lequel Git n'est pas installé. Bien que ce soit moins efficace que le protocole natif, Git sait communiquer par dessus HTTP.

Télécharger, compiler et installer Git sur votre compte et créer un dépôt dans votre dossier web :

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Avec les vieilles versions de Git, la commande de copie échoue et vous devez faire :

```
$ chmod a+x hooks/post-update
```

Maintenant vous pouvez transmettre vos modifications via SSH depuis n'importe lequel de vos clones :

```
$ git push web.server:/path/to/proj.git master
```

et n'importe qui peut récupérer votre projet grâce à :

```
$ git clone http://web.server/proj.git
```

Git via n'importe quoi

Besoin de synchroniser des dépôts sans passer par un serveur ni même une connexion réseau ? Besoin d'improviser dans l'urgence ? Nous avons déjà vu que **git fast-export** et **git fast-import** savent convertir et recréer un dépôt via un simple fichier. Nous pourrions utiliser des fichiers de ce type pour assurer le transport entre des dépôts Git via n'importe quel canal. Mais un outil plus puissant existe : **git bundle**.

L'émetteur crée un 'bundle' :

```
$ git bundle create monbundle HEAD
```

puis il transmet ce bundle, `monbundle`, à l'autre partie par n'importe quel moyen : email, clé USB, impression puis reconnaissance de caractères, lecture des bits au téléphone, signaux de fumée, etc. Le récepteur retrouve les mises à jour du bundle en tapant :

```
$ git pull monbundle
```

Le récepteur peut même faire cela dans un dépôt entièrement vide. Malgré sa petite taille `monbundle` contient l'ensemble du dépôt Git d'origine.

Pour des projets plus gros, on peut réduire le gaspillage en incluant dans le bundle uniquement les changements manquants dans l'autre dépôt. En supposant par exemple que le commit «1b6d…» est le commit le plus récent partagé par les deux dépôts, on peut faire :

```
$ git bundle create monbundle HEAD ^1b6d
```

Si on fait cela souvent, il se peut qu'on ne sache plus quel est le dernier commit partagé. La page d'aide suggère d'utiliser des tags pour résoudre ce problème. En pratique, juste après l'envoi d'un bundle, tapez :

```
$ git tag -f dernierbundle HEAD
```

et pour créer un nouveau bundle faites :

```
$ git bundle create nouveaubundle HEAD ^dernierbundle
```

Les patches : la monnaie d'échange globale

Les patches sont des représentations textuelles de vos modifications qui peuvent être facilement compris par les ordinateurs comme par les humains. C'est ce qui leur donne leur charme. Vous pouvez envoyer un patch par mail à un développeur sans savoir quel système de gestion de versions il utilise. À partir du moment où on peut lire les mails que vous envoyez, on peut voir vos modifications. De votre côté, vous n'avez besoin que d'un compte mail : aucune nécessité de mettre en œuvre un dépôt Git en ligne.

Souvenez-vous du premier chapitre. La commande :

```
$ git diff 1b6d > mon.patch
```

produit un patch qui peut être collé dans un mail. Dans un dépôt Git, tapez :

```
$ git apply < mon.patch
```

pour appliquer le patch.

D'une manière plus formelle, lorsque le nom des auteurs et peut-être leur signature doit apparaître, générez tous les patches depuis un certain point en tapant :

```
$ git format-patch 1b6d
```

Les fichiers résultants peuvent être fournis à **git send-email** ou envoyés à la main. Vous pouvez aussi spécifier un intervalle entre deux commits :

```
$ git format-patch 1b6d..HEAD^^
```

Du côté du destinataire, enregistrez un mail dans un fichier puis tapez :

```
$ git am < mail.txt
```

Ça appliquera le patch reçu mais créera aussi un commit en y incluant toutes les informations telles que le nom des auteurs.

Si vous utilisez un client de messagerie dans un navigateur, il vous faudra sans doute appuyer sur un bouton afin de voir le mail dans son format brut avant de l'enregistrer dans un fichier.

Il y a de légères différences dans le cas des clients de messagerie se basant sur le format mbox, mais si vous utilisez l'un d'entre eux, vous êtes sans aucun doute capable de vous en débrouiller facilement sans lire des tutoriels !

(NdT : si votre dépôt contient des fichiers binaires, n'oubliez-pas d'ajouter l'option `--binary` aux commandes de création de patches ci-dessus.)

Le numéro de votre correspondant a changé

Après la création d'un clone d'un dépôt, l'utilisation de **git push** ou de **git pull** se référera automatiquement à l'URL du dépôt d'origine. Comment Git fait-il ? Le secret réside dans des options de configuration ajoutées dans le clone. Jetons-y un œil :

```
$ git config --list
```

L'option `remote.origin.url` détermine l'URL de la source ; «origin» est un alias donné au dépôt d'origine. Comme dans le cas de la branche principale qui se nomme «master» par convention, on peut changer ou supprimer cet alias mais il n'y a habituellement aucune raison de le faire.

Si le dépôt original change, vous pouvez modifier son URL via :

```
$ git config remote.origin.url git://nouvel.url/proj.git
```

L'option `branch.master.merge` spécifie le nom de la branche distante utilisée par défaut par la commande **git pull**. Lors du clonage initial, le nom choisi est celui de la branche courant du dépôt d'origine. Même si le HEAD du dépôt d'origine est déplacé vers une autre branche, la commande pull continuera à suivre fidèlement la branche initiale.

Cette option ne s'applique qu'au dépôt ayant servi au clonage initial, celui enregistré dans l'option `branch.master.remote`. Si nous effectuons un pull depuis un autre dépôt, nous devrons indiquer explicitement la branche voulue :

```
$ git pull git://example.com/other.git master
```

Les détails ci-dessus expliquent pourquoi nos appels à push et pull dans nos précédents exemples n'avaient pas besoin d'arguments.

Les branches distantes

Lorsque nous clonons un dépôt, nous clonons aussi toutes ses branches. Vous ne les avez sans doute pas remarquées car Git les cache : vous devez explicitement demander à les voir. Cela empêche les branches du dépôt distant d'interférer avec vos propres branches et cela rend aussi Git plus simple pour les débutants.

Listons les branches distantes :

```
$ git branch -r
```

Vous devriez voir quelque chose comme :

```
origin/HEAD
origin/master
origin/experimental
```

Ces noms sont ceux des branches et du HEAD du dépôt distant et ils peuvent être utilisés dans les commandes Git normales. Supposez par exemple que vous avez réalisé de nombreux commits et que vous vouliez voir la différence avec la dernière version ramenée par fetch. Vous pourriez rechercher dans le log pour retrouver l'empreinte SHA1 appropriée mais il est beaucoup plus simple de tapez :

```
$ git diff origin/HEAD
```

Vous pouvez aussi voir où en est rendu la branche "experimental" :

```
$ git log origin/experimental
```

Dépôts distants multiples

Supposez que deux autres développeurs travaillent sur notre projet et que nous souhaitons garder un œil sur les deux. Nous pouvons suivre plus d'un dépôt à la fois grâce à :

```
$ git remote add un_autre git://example.com/un_depot.git
$ git pull un_autre une_branche
```

Maintenant nous avons fusionné avec une branche d'un second dépôt et nous avons accès facilement à toutes les branches de tous les dépôts :

```
$ git diff origin/experimental^ un_autre/une_branche~5
```

Mais comment faire si nous souhaitons juste comparer leurs modifications sans affecter notre travail ? En d'autres termes, nous voulons examiner leurs branches sans que leurs modifications envahissent notre dossier de travail. À la place d'un pull, faisons :

```
$ git fetch         # Rapatrier depuis le dépôt d'origin, par défaut
$ git fetch un_autre # Rapatrier depuis le dépôt d'un_autre
```

Cela ne rapatrie (fetch) que les historiques. Bien que notre dossier de travail reste inchangé, nous pouvons faire référence à n'importe quelle branche de n'importe quel dépôt dans nos commandes Git puisque nous en possédons maintenant une copie locale.

Rappelez-vous qu'en coulisse, un **pull** est simplement un **fetch** suivi d'un **merge**. Habituellement nous faisons appel à **pull** car nous voulons fusionner (merge) les dernières modifications distantes après les avoir rapatriées (fetch). La situation ci-dessus est une exception notable.

Voir **get help remote** pour savoir comment supprimer des dépôts distants, ignorer certaines branches et bien plus encore.

Mes préférences

Pour mes projets, j'aime que mes contributeurs se confectionnent un dépôt depuis lequel je peux effectuer des pull. Certains services d'hébergement Git vous permettent de créer votre propre dépôt clone d'un projet en cliquant simplement sur un bouton.

Après avoir rapatrié (fetch) un arbre de modifications, j'utilise les commandes Git pour parcourir et examiner ces modifications qui, idéalement, sont bien organisées et bien décrites. Je fusionne mes propres modifications et effectue parfois quelques modifications supplémentaires. Une fois satisfait, je les envoie (push) vers le dépôt principal.

Bien que recevant rarement des contributions, je pense que mon approche est parfaitement adaptable à grande échelle. Voir ce billet par Linus Torvalds.

Rester dans le monde Git est un peu plus pratique que de passer par des fichiers de patch puisque cela m'évite d'avoir à les convertir en commits Git. De plus, Git gère les détails tels qu'enregistrer le nom de l'auteur, son adresse mail ainsi que la date et l'heure et il demande à l'auteur de décrire ses propres modifications.

La maîtrise de Git

À ce stade, vous devez être capable de parcourir les pages de **git help** et comprendre presque tout (en supposant que vous lisez l'anglais). En revanche,

retrouver la commande exacte qui résoudra un problème précis peut être fastidieux. Je peux sans doute vous aider à gagner un peu de temps : vous trouverez ci-dessous quelques-unes des recettes dont j'ai déjà eu besoin.

Publication de sources

Dans mes projets, Git gère exactement tous les fichiers que je veux placer dans une archive afin de la publier. Pour créer une telle archive, j'utilise :

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

Gérer le changement

Indiquer à Git quels fichiers ont été ajoutés, supprimés ou renommés est parfois pénible pour certains projets. À la place, vous pouvez faire :

```
$ git add .
$ git add -u
```

Git cherchera les fichiers du dossier courant et gérera tous les détails tout seul. En remplacement de la deuxième commande *add*, vous pouvez utiliser `git commit -a` pour créer un nouveau commit directement. Lisez **git help ignore** pour savoir comment spécifier les fichiers qui doivent être ignorés.

Vous pouvez effectuer tout cela en une seule passe grâce à :

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Les options **-z** et **-0** empêchent les effets secondaires imprévus dûs au noms de fichiers contenant des caractères étranges. Comme cette commande ajoutent aussi les fichiers habituellement ignorés, vous voudrez sûrement utiliser les options `-x` ou `-X`.

Mon commit est trop gros !

Avez-vous négligé depuis longtemps de faire un commit ? Avez-vous codé furieusement et tout oublié de la gestion de versions jusqu'à présent ? Faites-vous plein de petits changements sans rapport entre eux parce que c'est votre manière de travailler ?

Pas de soucis. Faites :

```
$ git add -p
```

Pour chacune des modifications que vous avez faites, Git vous montrera le bout de code qui a changé et vous demandera si elle doit faire partie du prochain commit. Répondez par "y" (oui) ou par "n" (non). Vous avez aussi d'autres

options comme celle vous permettant de reporter votre décision ; tapez "?" pour en savoir plus.

Une fois satisfait, tapez :

```
$ git commit
```

pour faire un commit incluant exactement les modifications qui vous avez sélectionnées (les modifications indexées). Soyez certain de ne pas utiliser l'option **-a** sinon Git fera un commit incluant toutes vos modifications.

Que faire si vous avez modifié de nombreux fichiers en de nombreux endroits ? Vérifier chaque modification individuellement devient alors rapidement frustrant et abrutissant. Dans ce cas, utilisez la commande **git add -i** dont l'interface est moins facile mais beaucoup plus souple. En quelques touches vous pouvez ajouter ou retirer de votre index (voir ci-dessous) plusieurs fichiers d'un seul coup mais aussi valider ou non chacune des modifications individuellement pour certains fichiers. Vous pouvez aussi utiliser en remplacement la commande **git commit --interactive** qui effectuera un commit automatiquement quand vous aurez terminé.

L'index : l'aire d'assemblage

Jusqu'ici nous avons réussi à éviter de parler du fameux *index* de Git mais nous devons maintenant le présenter pour mieux comprendre ce qui précède. L'index est une aire d'assemblage temporaire. Git ne transfert que très rarement de données depuis votre dossier de travail directement vers votre historique. En fait, Git copie d'abord ces données dans l'index puis il copie toutes ces données depuis l'index vers leur destination finale.

Un **commit -a**, par exemple, est en fait un processus en deux temps. La première étape consiste à construire dans l'index un instantané de l'état actuel de tous les fichiers suivis par Git. La seconde étape enregistre cet instantané de manière permanente dans l'historique. Effectuer un commit sans l'option **-a** réalise uniquement cette deuxième étape et cela n'a de sens qu'après avoir effectué des commandes qui change l'index, telle que **git add**.

Habituellement nous pouvons ignorer l'index et faire comme si nous échangions directement avec l'historique. Dans certaines occasions, nous voulons un contrôle fin et nous gérons donc l'index. Nous plaçons dans l'index un instantané de certaines modifications (mais pas toutes) et enregistrons de manière permanente cet instantané soigneusement construit.

Ne perdez pas la tête

Le tag HEAD est comme un curseur qui pointe habituellement vers le tout dernier commit et qui avance à chaque commit. Certaines commandes Git vous

permettent de le déplacer. Par exemple :

```
$ git reset HEAD~3
```

déplacera HEAD trois commits en arrière. À partir de là, toutes les commandes Git agiront comme si vous n'aviez jamais fait ces trois commits, même si vos fichier restent dans leur état présent. Voir les pages d'aide pour quelques usages intéressants.

Mais comment faire pour revenir vers le futur ? Les commits passés ne savent rien du futur.

Si vous connaissez l'empreinte SHA1 du HEAD original, faites alors :

```
$ git reset 1b6d
```

Mais que faire si vous ne l'avez pas regardé ? Pas de panique : pour des commandes comme celle-ci, Git enregistre la valeur originale de HEAD dans un tag nommé ORIG_HEAD et vous pouvez revenir sain et sauf via :

```
$ git reset ORIG_HEAD
```

Chasseur de tête

Peut-être que ORIG_HEAD ne vous suffit pas. Peut-être venez-vous de vous apercevoir que vous avez fait une monumentale erreur et que vous devez revenir à une ancienne version d'une branche oubliée depuis longtemps.

Par défaut, Git conserve un commit au moins deux semaine même si vous avez demandé à Git de détruire la branche qui le contient. La difficulté consiste à retrouver l'empreinte appropriée. Vous pouvez toujours explorer les différentes valeurs d'empreinte trouvées dans `.git/objects` et retrouver celle que vous cherchez par essais et erreurs. Mais il existe un moyen plus simple.

Git enregistre l'empreinte de chaque commit qu'il traite dans `.git/logs`. Le sous-dossier `refs` contient l'historique de toute l'activité de chaque branche alors que le fichier `HEAD` montre chaque valeur d'empreinte que HEAD a pu prendre. Ce dernier peut donc servir à retrouver les commits d'une branche qui a été accidentellement élaguée.

La commande reflog propose une interface sympa vers ces fichiers de log. Essayez:

```
$ git reflog
```

Au lieu de copier/coller une empreinte listée par reflog, essayez :

```
$ git checkout "@{10 minutes ago}"
```

Ou basculez vers le cinquième commit précédemment visité via :

```
$ git checkout "@{5}"
```

Voir la section «Specifying Revisions» de **git help rev-parse** pour en savoir plus.

Vous pouvez configurer une plus longue période de rétention pour les commits condamnés. Par exemple :

```
$ git config gc.pruneexpire "30 days"
```

signifie qu'un commit effacé ne le sera véritablement qu'après 30 jours et lorsque $git gc* tournera.

Vous pouvez aussi désactiver le déclenchement automatique de **git gc** :

```
$ git config gc.auto 0
```

auquel cas les commits ne seront véritablement effacés que lorsque vous lancerez **git gc** manuellement.

Construire au-dessus de Git

À la manière UNIX, la conception de Git permet son utilisation comme un composant de bas niveau d'autres programmes tels que des interfaces graphiques ou web, des interfaces en ligne de commandes alternatives, des outils de gestion de patch, des outils d'importation et de conversion, etc. En fait, certaines commandes Git sont de simples scripts s'appuyant sur les commandes de base, comme des nains sur des épaules de géants. Avec un peu de bricolage, vous pouvez adapter Git à vos préférences.

Une astuce facile consiste à créer des alias Git pour raccourcir les commandes que vous utilisez le plus fréquemment :

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # affiche les alias connus
alias.co checkout
$ git co foo                          # identique à 'git checkout foo'
```

Une autre astuce consiste à intégrer le nom de la branche courante dans votre prompt ou dans le titre de la fenêtre. L'invocation de :

```
$ git symbolic-ref HEAD
```

montre le nom complet de la branche courante. En pratique, vous souhaiterez probablement enlever "refs/heads/" et ignorer les erreurs :

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

Le sous-dossier `contrib` de Git est une mine d'outils construits au-dessus de Git. Un jour, certains d'entre eux pourraient être promus au rang de commandes officielles. Dans Debian et Ubuntu, ce dossier est `/usr/share/doc/git-core/contrib`.

L'un des plus populaires de ces scripts est `workdir/git-new-workdir`. Grâce à des liens symboliques intelligents, ce script crée un nouveau dépôt dont l'historique est partagé avec le dépôt original.

```
$ git-new-workdir un/existant/depot nouveau/repertoire
```

Le nouveau dossier et ses fichiers peuvent être vus comme un clone, sauf que l'historique est partagé et que les deux arbres des versions restent automatiquement synchrones. Nul besoin de merge, push ou pull.

Audacieuses acrobaties

À ce jour, Git fait tout son possible pour que l'utilisateur ne puisse pas effacer accidentellement des données. Mais si vous savez ce que vous faites, vous pouvez passer outre les garde-fous des principales commandes.

Checkout : des modifications non intégrées (via commit) peuvent causer l'échec d'un checkout. Pour détruire vos modifications et réussir quoi qu'il arrive un checkout d'un commit donné, utilisez l'option d'obligation :

```
$ git checkout -f HEAD^
```

Inversement, si vous spécifiez des chemins particuliers pour un checkout alors il n'y a pas de garde-fous. Le contenu des chemins est silencieusement réécrit. Faites attention lorsque vous utilisez un checkout de cette manière.

Reset : un reset échoue aussi en présence de modifications non intégrées. Pour passer outre, faites :

```
$ git reset --hard 1b6d
```

Branch : la suppression de branches échoue si cela implique la perte de certains commits. Pour forcer la suppression, tapez :

```
$ git branch -D branche_morte  # à la place de -d
```

De manière similaire, une tentative visant à renommer une branche existante vers le nom d'une autre branche échoue si cela amène la perte de commits. Pour forcer le changement de nom, tapez :

```
$ git branch -M source target  # à la place de -m
```

Contrairement à checkout et reset, ces deux dernières commandes n'effectuent pas la suppression des informations immédiatement. Les commits destinés à disparaître sont encore disponibles dans le sous-dossier .git et peuvent encore être retrouvés grâce aux empreintes appropriées tel que retrouvées dans `.git/logs` (voir "Chasseur de tête" ci-dessus). Par défaut, ils sont conservés au moins deux semaines.

Clean : certaines commandes Git refusent de s'exécuter pour ne pas écraser des fichiers non suivis. Si vous êtes certain que tous ces fichiers et dossiers peuvent être sacrifiés alors effacez-les sans pitié via :

```
$ git clean -f -d
```

Ensuite, la commande trop prudente fonctionnera !

Se prémunir des commits erronés

Des erreurs stupides encombrent mes dépôts. Les plus effrayantes sont dues à des fichiers manquants car oubliés lors des **git add**. D'autres erreurs moins graves concernent les espaces blancs inutiles ou les conflits de fusion non résolus : bien qu'inoffensives, j'aimerais qu'elles n'apparaissent pas dans les versions publiques.

Si seulement je m'en étais prémuni en utilisant un *hook* (un crochet) pour m'alerter de ces problèmes :

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # Vieilles versions de Git : chmod +x pre-commit
```

Maintenant Git empêchera un commit s'il détecte des espace inutiles ou s'il reste des conflits de fusion non résolus.

Pour gérer ce guide, j'ai aussi ajouté les lignes ci-dessous au début de mon hook **pre-commit** pour me prémunir de mes inattentions :

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

Plusieurs opération de Git acceptent les hooks ; voir **git help hooks**. Nous avons déjà utilisé le hook **post-update** lorsque nous avons parlé de Git au-dessus de HTTP. Celui-ci se déclenche à chaque mouvement de HEAD. Le script d'exemple post-update met à jour les fichiers Git nécessaires à une communication au-dessus de transports agnostiques tels que HTTP.

Les secrets révélés

Nous allons jeter un œil sous le capot pour comprendre comment Git réalise ses miracles. Je passerai sous silence la plupart des détails. Pour des explications plus détaillées, référez-vous au manuel utilisateur.

L'invisibilité

Comment fait Git pour être si discret ? Mis à part lorsque vous faites des commits et des fusions, vous pouvez travailler comme si la gestion de versions n'existait pas. Et c'est lorsque vous en avez besoin que vous êtes content de voir que Git veillait sur vous tout le temps.

D'autres systèmes de gestion de versions vous mettent constamment aux prises avec de la paperasserie et de la bureaucratie. Les fichiers sont en lecture seule jusqu'à l'obtention depuis un serveur central du droit d'édition de tel ou tel fichier. Les commandes les plus basiques voient leurs performances s'écrouler au fur et à mesure que le nombre d'utilisateurs augmente. Le travail s'arrête dès lors que le réseau ou le serveur central est en panne.

À l'inverse, Git conserve tout l'historique de votre projet dans le sous-dossier `.git` de votre dossier de travail. C'est votre propre copie de l'historique et vous pouvez donc rester déconnecté tant que vous ne voulez pas communiquer avec les autres. Vous conservez un contrôle total sur le sort de vos fichiers puisque Git peut aisément les recréer à tout moment à partir de l'un des états enregistrés dans `.git`.

L'intégrité

La plupart des gens associent la cryptographie à la conservation du secret des informations mais l'un de ses buts tout aussi important est de conserver l'intégrité de ces informations. Un usage approprié des fonctions de hachage cryptographiques (celles qui calculent l'empreinte d'un document) permet d'empêcher la corruption accidentelle ou malicieuse des données.

Une empreinte SHA1 peut être vue comme un nombre de 160 bits identifiant de manière unique n'importe quelle suite d'octets que vous rencontrerez dans votre vie. On peut même aller plus loin : c'est vrai pour toutes les suites d'octets que les humains utiliseront sur plusieurs générations.

Comme une empreinte SHA1 est elle-même une suite d'octets, nous pouvons calculer l'empreinte d'une suite de caractères contenant d'autres empreintes. Cette simple observation est étonnamment utile (cherchez par exemple *hash chain*). Nous verrons plus tard comment Git utilise cela pour garantir efficacement l'intégrité des données.

En bref, Git conserve vos données dans le sous-dossier `.git/objects` mais à la place des noms de fichiers normaux, vous n'y trouverez que des ID. En utilisant ces ID comme noms de fichiers et grâce à quelques astucieux fichiers de verrouillage et d'horodatage, Git transforme un simple système de fichiers en une base de données efficace et robuste.

L'intelligence

Comment fait Git pour savoir que vous avez renommé un fichier même si vous ne lui avez pas dit explicitement ? Bien sûr, vous pouvez utiliser **git mv** mais c'est exactement la même chose que de faire **git rm** suivi par **git add**.

Git a des heuristiques pour débusquer les changements de noms et les copies entre les versions successives. En fait, il peut même détecter les bouts de code qui ont été déplacés ou copiés d'un fichier à un autre ! Bien que ne couvrant pas tous les cas, cela marche déjà très bien et cette fonctionnalité est encore en cours d'amélioration. Si cela échoue pour vous, essayez les options activant des méthodes de détection de copie plus coûteuses et envisager de faire une mise à jour.

L'indexation

Pour chaque fichier suivi, Git mémorise des informations, telles que sa taille et ses dates de création et de dernières modifications, dans un fichier appelé *index*. Pour déterminer si un fichier a changé, Git compare son état courant avec ce qu'il a mémorisé dans l'index. Si cela correspond alors Git n'a pas besoin de relire le fichier.

Puisque les appels à *stat* sont considérablement plus rapides que la lecture des fichiers, si vous n'avez modifié que quelques fichiers, Git peut déterminer son état en très peu de temps.

Nous avons dit plus tôt que l'index était une aire d'assemblage. Comment se peut-il qu'un simple fichier contant quelques informations sur les fichiers soit une aire d'assemblage ? Parce que la commande add ajoute les fichiers à la base de données de Git et met à jour l'index avec leurs informations alors que la commande commit, sans option, crée une nouvelle version basée uniquement sur cet index et les fichiers déjà inclus dans la base de données.

Les origines de Git

Ce message de la Mailing List du noyau Linux décrit l'enchaînement des évènements ayant mené à Git. L'ensemble de l'enfilade est un site archéologique fascinant pour les historiens de Git.

La base d'objets

Chacune des versions de vos données est conservée dans la base d'objets (*object database*) qui réside dans le sous-dossier `.git/objects` ; le reste du contenu du dossier `.git` représente moins de données : l'index, le nom des branches, les

tags, les options de configuration, les logs, l'emplacement actuel de HEAD, et ainsi de suite. La base d'objets est simple mais élégante et constitue la source de la puissance de Git.

Chaque fichier dans `.git/objects` est un objet. Il y a trois sortes d'objets qui nous concerne : les `blobs`, les arbres (`trees`) et les `commits`.

Les blobs

Tout d'abord, faisons un peu de magie. Choisissez un nom de fichier… n'importe quel nom de fichier ! Puis dans un dossier vide, faites (en remplaçant `VOTRE_NOM_DE_FICHIER` par le nom que vous avez choisi) :

```
$ echo joli > VOTRE_NOM_DE_FICHIER
$ git init
$ git add .
$ find .git/objects -type f
```

Vous verrez `.git/objects/06/80f15d4cb13a09f600a25b84eae36506167970`.

Comment puis-je le savoir sans connaître le nom de fichier que vous avez choisi ? Tout simplement parce que l'empreinte SHA1 de :

```
"blob" SP "5" NUL "joli" LF
```

est 0680f15d4cb13a09f600a25b84eae36506167970. Où SP est un espace, NUL est l'octet de valeur nulle et LF est un passage à la ligne. Vous pouvez vérifier cela en tapant :

```
$ printf "blob 5\000joli\n" | sha1sum
```

Git utilise un classement par contenu : les fichiers ne sont pas stockés selon leur nom mais selon l'empreinte des données qu'ils contiennent, dans un fichier que nous appelons un objet *blob*. Nous pouvons considérer l'empreinte comme un ID unique du contenu d'un fichier. Donc nous pouvons retrouver un fichier par son contenu. La chaîne initiale `blob 5` est simplement un entête indiquant le type de l'objet et sa longueur en octets ; cela simplifie le classement interne.

Je peux donc aisément prédire ce que vous voyez. Le nom du fichier ne compte pas : pour construire l'objet blob, seules comptent les données stockées dans le fichier.

Peut-être vous demandez-vous ce qui se produit pour des fichiers ayant le même contenu. Essayez en créant des copies de votre premier fichier, avec des noms quelconques. Le contenu de `.git/objects` reste le même quel que soit le nombre de copies que vous avez ajoutées. Git ne stocke le contenu qu'une seule fois.

À propos, les fichiers dans `.git/objects` sont compressés par zlib et, par conséquent, vous ne pouvez pas en consulter le contenu directement. Passez-les au travers du filtre zpipe -d ou tapez :

```
$ git cat-file -p 0680f15d4cb13a09f600a25b84eae36506167970
```

qui affiche proprement l'objet choisi.

Les arbres (`trees`)

Mais que deviennent les noms des fichiers ? Ils doivent bien être stockés quelque part à un moment. Git se préoccupe des noms de fichiers lors d'un commit :

```
$ git commit  # Tapez un message
$ find .git/objects -type f
```

Vous devriez voir maintenant trois objets. Mais là, je ne peux plus prédire le nom des deux nouveaux fichiers puisqu'ils dépendent en partie du nom de fichier que vous avez choisi. Nous continuerons en supposant que vous avez choisi «rose». Si ce n'est pas le cas, vous pouvez réécrire l'histoire pour que ce soit le cas :

```
$ git filter-branch --tree-filter 'mv VOTRE_NOM_DE_FICHIER rose'
$ find .git/objects -type f
```

Le fichier `.git/objects/9a/6a950c3b14eb1a3fb540a2749514a1cb81e206` devrait maintenant apparaître puisque c'est l'empreinte SHA1 du contenu suivant :

```
"tree" SP "32" NUL "100644 rose" NUL 0x9a6a950c3b14eb1a3fb540a2749514a1cb81e206
```

Vérifiez que ce contenu est le bon en tapant :

```
$ echo 9a6a950c3b14eb1a3fb540a2749514a1cb81e206 | git cat-file --batch
```

Avec zpipe, il est plus simple de vérifier l'empreinte :

```
$ zpipe -d < .git/objects/9a/6a950c3b14eb1a3fb540a2749514a1cb81e206 | sha1sum
```

La vérification de l'empreinte est plus difficile via cat-file puisque cette commande n'affiche pas que le contenu brut du fichier après décompression.

Cette fichier est un objet arbre (*tree*) : une liste de tuples constitués d'un type, d'un nom de fichier et d'une empreinte. Dans notre exemple, le type est 100644 qui indique que `rose` est un fichier normal et l'empreinte est celle de l'objet de type blob contenant le contenu de `rose`. Les autres types possibles pour un fichier sont exécutable, lien symbolique ou dossier. Dans ce dernier cas, l'empreinte représente un autre objet de type arbre.

Si vous faites appel à la commande filter-branch, vous verrez apparaître de vieux objets dont vous n'avez pas besoin. Même s'ils disparaîtront automatiquement une fois expirée la période de rétention, nous allons les effacer dès maintenant pour rendre notre petit exemple plus facile à suivre :

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
```

```
$ git prune
```

Sur de vrais projets, vous devriez éviter de telles commandes puisqu'elles détruisent les sauvegardes. Si vous voulez un dossier propre, il est conseillé de faire un tout nouveau clone. Faites aussi attention si vous manipulez directement le contenu de `.git` : que se passera-t-il si une commande Git s'effectue au même moment ou si le courant est soudainement coupé ?

De manière générale, les refs devraient toujours être effacées via **git update-ref -d** même si on considère comme sans risque la suppression manuelle de `refs/original`.

Les commits

Nous avons expliqué 2 des 3 types d'objets. Le troisième est l'objet *commit*. Son contenu dépend du message de commit ainsi que de la date et l'heure auxquelles il a été créé. Pour que vous obteniez la même chose qu'ici, nous devons bidouiller un peu :

```
$ git commit --amend -m Shakespeare  # Changement de message de commit
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"'  # Trucage de la date, l'heure et l'auteur.
$ find .git/objects -type f
```

Le fichier `.git/objects/ae/9d1241b2b6eea90529149a065f6bc444365c2a` devrait maintenant exister puisque c'est l'empreinte SHA1 du contenu suivant :

```
"commit 158" NUL
"tree 9a6a950c3b14eb1a3fb540a2749514a1cb81e206" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

Comme précédemment, vous pouvez utiliser zpipe ou cat-file pour vérifier par vous-même.

C'est le premier commit, ce qui explique pourquoi il n'y a pas de commit parent. Mais les commits suivants contiendront toujours au moins une ligne identifiant un commit parent.

Indiscernable de la magie

Les secrets de Git semblent trop simples. On imagine qu'il suffit de mélanger quelques scripts shell et d'y ajouter une pincée de code C pour mitonner un tel système en quelques heures : un assemblage d'opérations basiques sur les fichiers et de calcul d'empreintes SHA1 garni de quelques fichiers verrou et d'appels à fsync pour la robustesse. En fait, nous venons précisément de décrire les premières versions de Git. Malgré tout, mis à part quelques techniques astucieuses de compression pour gagner de la place et d'indexation pour gagner du temps, nous savons maintenant comment Git transforme adroitement un système de fichiers en une base de données parfaitement adaptée à de la gestion de versions.

Par exemple, si un fichier quelconque de la base d'objets vient à être corrompu par une erreur disque alors son empreinte ne correspond plus et nous sommes alertés du problème. En calculant l'empreinte des empreintes d'autres objets, nous maintenons l'intégrité à tous les niveaux. Les commits sont atomiques puisque ils ne peuvent jamais mémoriser des modifications partiellement stockées : nous ne pouvons calculer l'empreinte d'un commit et le stocker dans la base d'objets qu'après y avoir déjà stocké tous les arbres, blobs et parents relatifs à ce commit. La base d'objets est immunisée contre les interruptions inattendues telles que les coupures de courant.

Nous faisons même échouer les tentatives d'attaque les plus sournoises. Supposez que quelqu'un tente de modifier discrètement le contenu d'un fichier dans l'une des anciennes versions du projet. Pour rendre cohérent le contenu de la base d'objets, il lui faut changer l'empreinte de l'objet blob correspondant puisque elle doit maintenant représenter une chaîne d'octets différente. Cela signifie qu'il doit aussi changer l'empreinte de tous les arbres référençant ce blob et donc changer l'empreinte de tous les commits impliquant ces arbres ainsi que de tous les descendants de ces commits. Cela implique que l'empreinte du HEAD officiel diffère de celle du HEAD d'un dépôt corrompu. En remontant la suite d'empreintes erronées nous pouvons localiser avec précision le fichier corrompu ainsi que le premier commit où il l'a été.

En résumé, tant que nous sommes sûrs des 20 octets représentant le dernier commit, il est impossible d'altérer un dépôt Git.

Qu'en est-il des fameuses fonctionnalités de Git ? Des branchements ? Des fusions ? Des tags ? De simples détails. La tête courante est conservée dans le fichier `.git/HEAD` qui contient l'empreinte d'un objet commit. Cette empreinte sera tenue à jour durant un commit ainsi que durant de nombreuses autres commandes. Les branches fonctionnent de manière similaire : ce sont des fichiers dans `.git/refs/heads`. Et les tags aussi : ils sont dans `.git/refs/tags` mais ils sont mis à jour par un ensemble différent de commandes.

Appendix A: Les lacunes de Git

Git présente quelques problèmes que j'ai soigneusement cachés. Certains peuvent être résolus par des scripts et des hooks, d'autres nécessitent une réorganisation ou une redéfinition du projet et pour les quelques rares ennuis restants, il vous suffit d'attendre. Ou mieux encore, de donner un coup de main.

Les faiblesses de SHA1

Avec le temps, les spécialistes de cryptographie découvrent de plus en plus de faiblesses de SHA1. À ce jour, la découverte de collisions d'empreintes semble à la portée d'organisations bien dotées. Et d'ici quelques années, peut-être que même un simple PC aura assez de puissance de calcul pour corrompre de manière indétectable un dépôt Git.

Heureusement Git aura migré vers une fonction de calcul d'empreintes de meilleure qualité avant que de futures recherches détruisent SHA1.

Microsoft Windows

Git sur Microsoft Windows peut être jugé encombrant :

- Cygwin est un environnement de type Linux dans Windows proposant un portage de Git.
- Git pour Windows est un autre choix nécessitant beaucoup moins de place. Néanmoins quelques commandes doivent encore être améliorées.

Des fichiers sans relation

Si votre projet est très gros et contient de nombreux fichiers sans relation entre eux et changeant constamment, Git peut être plus défavorisé que d'autres systèmes puisque les fichiers pris séparément ne sont pas pistés. Git piste les changement de l'ensemble du projet, ce qui est habituellement bénéfique.

Une solution consiste à découper votre projet en plusieurs parties, chacune réunissant des fichiers en relation entre eux. Utilisez **git submodule** si vous souhaitez conserver tout cela dans un seul dossier.

Qui modifie quoi ?

Certains systèmes de gestion de versions vous oblige à marquer explicitement un fichier avant de pouvoir le modifier. Bien que particulièrement ennuyeux

puisque pouvant impliquer une communication avec un serveur central, cela présente deux avantages :

1. Les diffs sont plus rapides puisque seuls les fichiers marqués doivent être examinés.
2. Quelqu'un peut savoir qui travaille sur un fichier en demandant au serveur central qui l'a marqué pour modification.

Avec quelques scripts appropriés, vous pouvez obtenir la même chose avec Git. Cela nécessite la coopération du développeur qui doit exécuter un script particulier avant toute modification d'un fichier.

L'historique d'un fichier

Puisque Git enregistre les modifications de manière globale au projet, la reconstruction de l'historique d'un seul fichier demande plus de travail qu'avec un système de gestion de versions qui traque les fichiers individuellement.

Ce surplus est généralement négligeable et en vaut la peine puisque cela permet aux autres opérations d'être incroyablement efficaces. Par exemple, `git checkout` est plus rapide que `cp -a` et un delta de versions globale au projet se compresse mieux qu'une collection de delta fichier par fichier.

Le clone initial

La création d'un clone est plus coûteuse que l'extraction de code des autres systèmes quand il y a un historique conséquent.

Ce coût initial s'avère payant dans le temps puisque la plupart des opérations futures s'effectueront rapidement et hors-ligne. En revanche, dans certains situations, il est préférable de créer un clone superficiel grâce à l'option `--depth` (qui limite la profondeur de l'historique). C'est plus rapide mais le clone ainsi créé offre des fonctionnalités réduites.

Les projets versatiles

Git a été conçu pour être rapide au regard de la taille des changements. Les humains font de petits changement de version en version. Une correction de bug en une ligne ici, une nouvelle fonctionnalité là, un commentaire amendé ailleurs… Mais si vos fichiers changent radicalement à chaque révision alors, à chaque commit, votre historique grossit d'un poids équivalent à celui de votre projet.

Il n'y a rien qu'un système de gestion de versions puisse faire pour éviter cela, mais les utilisateurs de Git en souffrent plus puisque chaque clone contient habituellement l'historique complet.

Il faut rechercher les raisons de ces changements radicaux. Peut-être faut-il changer les formats des fichiers. Des modifications mineures ne devraient modifier que très peu de chose dans très peu de fichiers.

Peut-être qu'une base données ou une solution d'archivage est-elle plus adaptée comme solution qu'un système de gestion de versions. À titre d'exemple, un système de gestion de versions n'est certainement pas bien taillé pour gérer des photos prises périodiquement par une webcam.

Si les fichiers doivent absolument se transformer constamment et s'il faut absolument les gérer par version, une possibilité peut être une utilisation centralisée d'un dépôt Git. Chacun ne crée qu'un clone superficiel ne contenant qu'un historique récent voire inexistant du projet. Évidemment de nombreux outils Git ne seront plus utilisables et les corrections devront être fournies sous forme de patches. C'est sans doute acceptable sans en savoir plus sur les raisons réelles de la conservation de l'historique de nombreux fichiers instables.

Un autre exemple serait un projet dépendant d'un firmware qui prend la forme d'un énorme fichier binaire. L'historique de ce firmware n'intéresse pas les utilisateur et les mises à jour se compressent difficilement et donc les révisions de ce firmware vont faire grossir inutilement le dépôt.

Dans ce cas, le code source devrait être stocké dans le dépôt Git et les fichiers binaires conservés séparément. Pour rendre la vie meilleure, on peut distribuer un script qui utilisera un clone Git pour le code et rsync ou un clone Git superficiel pour le firmware.

Compteur global

Certains systèmes de gestion de versions centralisés gère un entier positif qui augmente à chaque commit accepté. Git fait référence à un changement par son empreinte ce qui est mieux pour de nombreuses raisons.

Mais certains aiment voir ce compteur. Par chance, il est très facile d'écrire un script qui se déclenchera à chaque mise à jour du dépôt Git central et incrémentera un compteur, peut-être dans un tag, qu'il associera à l'empreinte du dernier commit.

Chaque clone peut gérer un tel compteur mais c'est probablement sans intérêt puisque seul le compteur du dépôt central compte.

Les dossiers vides

Les sous-dossiers vides ne peuvent pas être suivis. Placez-y des fichiers sans intérêt pour remédier à ce problème.

Cette limitation n'est pas une fatalité due à la conception de Git mais un choix de l'implémentation actuelle. Avec un peu de chance, si de nombreux utilisateurs le demandent, cette fonctionnalité pourrait être ajoutée.

Le premier commit

Un informaticien typique compte à partir de 0 plutôt que de 1. Malheureusement, concernant les commits, Git n'adhère pas à cette convention. Plusieurs commandes ne fonctionnent pas avant le tout premier commit. De plus, certains cas limites doivent être gérés spécifiquement : par exemple, un rebasage vers une branche avec un commit initial différent.

Git bénéficierait à définir le commit zéro : dès la création d'un dépôt, HEAD serait défini comme la chaîne contenant 20 octets nuls. Ce commit spécial représenterait un arbre vide, sans parent, qui serait présent dans tous les dépôts Git.

Ainsi l'appel à git log, par exemple, pourrait indiquer à l'utilisateur qu'aucun commit n'a été fait au lieu de se terminer par une erreur fatale. Il en serait de même pour les autres outils.

Le commit initial serait un descendant implicite de ce commit zéro.

Mais ce n'est pas le cas et donc certains problèmes peuvent se poser. Si plusieurs branches avec des commits initiaux différents sont fusionnées alors le rebasage du résultat requiert de nombreuses interventions manuelles.

Bizarreries de l'interface

Étant donné deux commits A et B, le sens des expressions "A..B" et "A…B" diffère selon que la commande attend deux extrémités ou un intervalle. Voir **git help diff** et **git help rev-parse**.

Appendix B: Traduire ce guide

Faites un clone du source puis créer un répertoire correspondant au code IETF de la langue souhaitée : voir l'article du W3C concernant l'internationalisation. Par exemple pour l'anglais c'est "en", pour le japonais c'est "ja" et pour le chinois traditionnel c'est "zh-Hant". Dans ce nouveau répertoire, traduisez chacun des fichiers `txt` du répertoire "en" original.

Par exemple, pour créer ce guide en Klingon, vous devriez faire :

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" est le code IETF de la langue Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # Traduire le fichier.
```

et ainsi de suite pour tous les fichiers. Vous pouvez relire votre travail incrémentalement :

```
$ make LANG=tlh
$ firefox book.html
```

Faites souvent des commits pour vos modifications puis faites-le moi savoir dès que c'est prêt. GitHub.com propose une interface qui facilite les choses : faites un fork du projet "gitmagic", poussez-y vos modifications et demandez-moi de les fusionner.

J'aime avoir des traductions qui suivent le schéma ci-dessus car mes scripts peuvent alors produire les versions HTML et PDF. Et puis c'est pratique de conserver toutes les traductions dans le dépôt officiel. Mais que cela ne vous empêche pas de faire ce qui vous convient le mieux : par exemple, les traducteurs chinois préfèrent utiliser Google Docs. Je suis content tant que votre travail permet à d'autres de profiter de mon travail.