
# Support des Travaux pratiques Git

**ahmedhosni.contact@gmail.com**

# Démarrage rapide


- À propos de la gestion de version
  - Les systèmes de gestion de version locaux
  - Les systèmes de gestion de version centralisés
  - Les systèmes de gestion de version distribués
- Une rapide histoire de Git
- Rudiments de Git
  - Des instantanés, pas des différences
  - Presque toutes les opérations sont locales
  - Git gère l'intégrité
  - Généralement, Git ne fait qu'ajouter des données
  - Les trois états
- La ligne de commande
- Installation de Git
  - Installation sur Linux
  - Installation sur Mac
  - Installation sur Windows
  - Installation depuis les sources
- Paramétrage à la première utilisation de Git
  - Votre identité
  - Votre éditeur de texte
  - Vérifier vos paramètres
- Obtenir de l'aide
- Résumé


Ce chapitre traite du démarrage rapide avec Git. Nous commencerons par expliquer les bases de la gestion de version, puis nous parlerons de l'installation de Git sur votre système et finalement du paramétrage pour commencer à l'utiliser. À la fin de ce chapitre vous devriez en savoir assez pour comprendre pourquoi on parle beaucoup de Git, pourquoi vous devriez l'utiliser et vous devriez en avoir une installation prête à l'emploi.

# À propos de la gestion de version

Qu'est-ce que la gestion de version et pourquoi devriez-vous vous en soucier ? Un gestionnaire de version est un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment. Dans les exemples de ce livre, nous utiliserons des fichiers sources de logiciel comme fichiers sous gestion de version, bien qu'en réalité on puisse l'utiliser avec pratiquement tous les types de fichiers d'un ordinateur.

Si vous êtes un dessinateur ou un développeur web, et que vous voulez conserver toutes les versions d'une image ou d'une mise en page (ce que vous souhaiteriez assurément), un système de gestion de version (VCS en anglais pour *Version Control System*) est un outil qu'il est très sage d'utiliser. Il vous permet de ramener un fichier à un état précédent, de ramener le projet complet à un état précédent, de visualiser les changements au cours du temps, de voir qui a modifié quelque chose qui pourrait causer un problème, qui a introduit un problème et quand, et plus encore. Utiliser un VCS signifie aussi généralement que si vous vous trompez ou que vous perdez des fichiers, vous pouvez facilement revenir à un état stable. De plus, vous obtenez tous ces avantages avec peu de travail additionnel.

## Les systèmes de gestion de version locaux

La méthode courante pour la gestion de version est généralement de recopier les fichiers dans un autre répertoire (peut-être avec un nom incluant la date dans le meilleur des cas). Cette méthode est la plus courante parce que c'est la plus simple, mais c'est aussi la moins fiable. Il est facile d'oublier le répertoire dans lequel vous êtes et d'écrire accidentellement dans le mauvais fichier ou d'écraser des fichiers que vous vouliez conserver.

Pour traiter ce problème, les programmeurs ont développé il y a longtemps des VCS locaux qui utilisaient une base de données simple pour conserver les modifications d'un fichier.

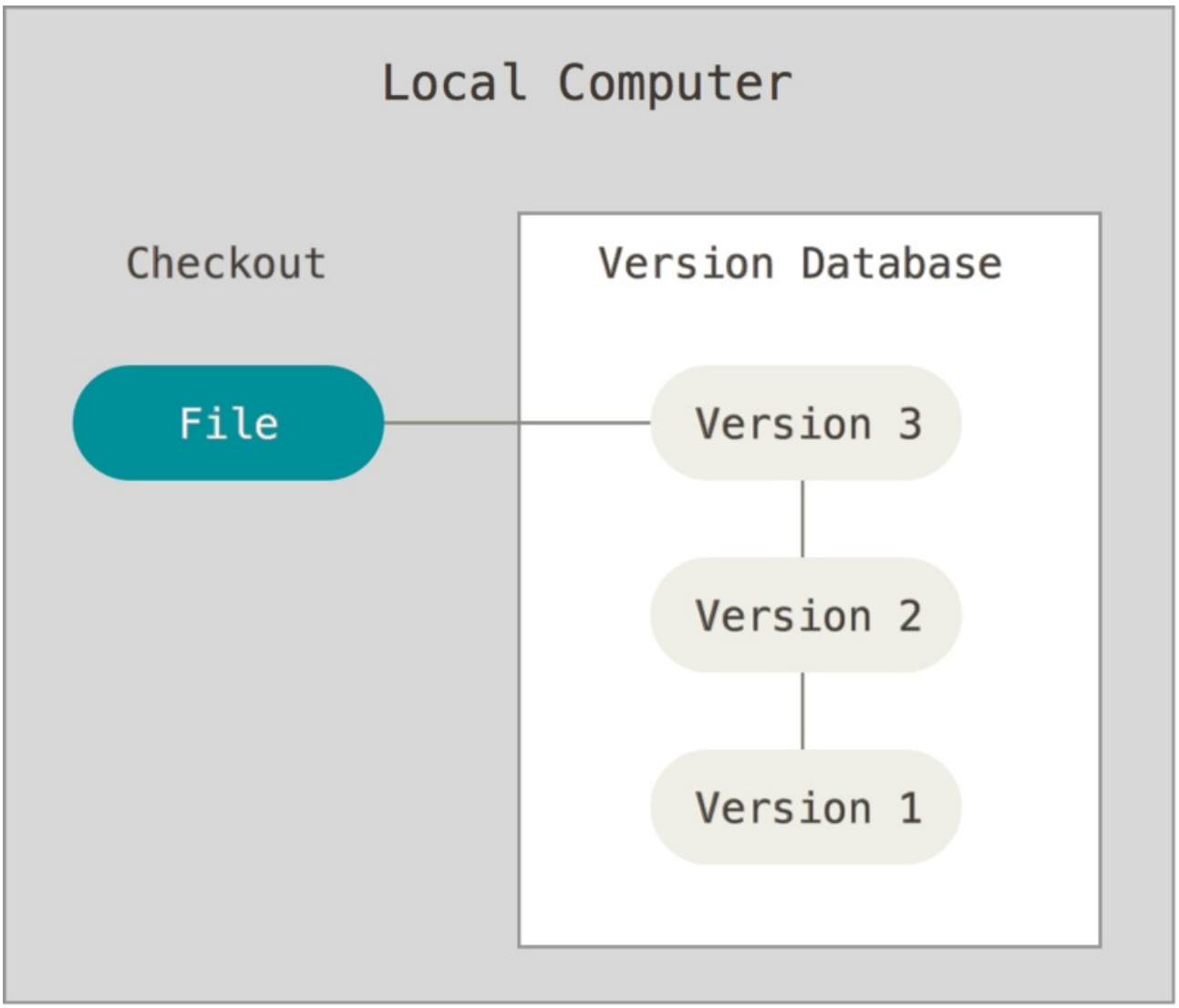


*Figure 2 : Diagramme de gestion de version locale*

Figure 1. Gestion de version locale.

Un des systèmes les plus populaires était RCS, qui est encore distribué avec de nombreux systèmes d'exploitation aujourd'hui. Même le système d'exploitation populaire Mac OS X inclut le programme `rcs` lorsqu'on installe les outils de développement logiciel. Cet outil fonctionne en conservant des ensembles de patchs (c'est-à-dire la différence entre les fichiers) d'une version à l'autre dans un format spécial sur disque ; il peut alors restituer l'état de n'importe quel fichier à n'importe quel instant en ajoutant toutes les différences.

## Les systèmes de gestion de version centralisés

Le problème majeur que les gens rencontrent est qu'ils ont besoin de collaborer avec des développeurs sur d'autres ordinateurs. Pour traiter ce problème, les systèmes de gestion de version centralisés (CVCS en anglais pour *Centralized Version Control Systems*) furent développés. Ces systèmes tels que CVS, Subversion, et Perforce, mettent en place un serveur central qui contient tous les fichiers sous gestion de version, et des clients qui peuvent extraire les fichiers de ce dépôt central. Pendant de nombreuses années, cela a été le standard pour la gestion de version.

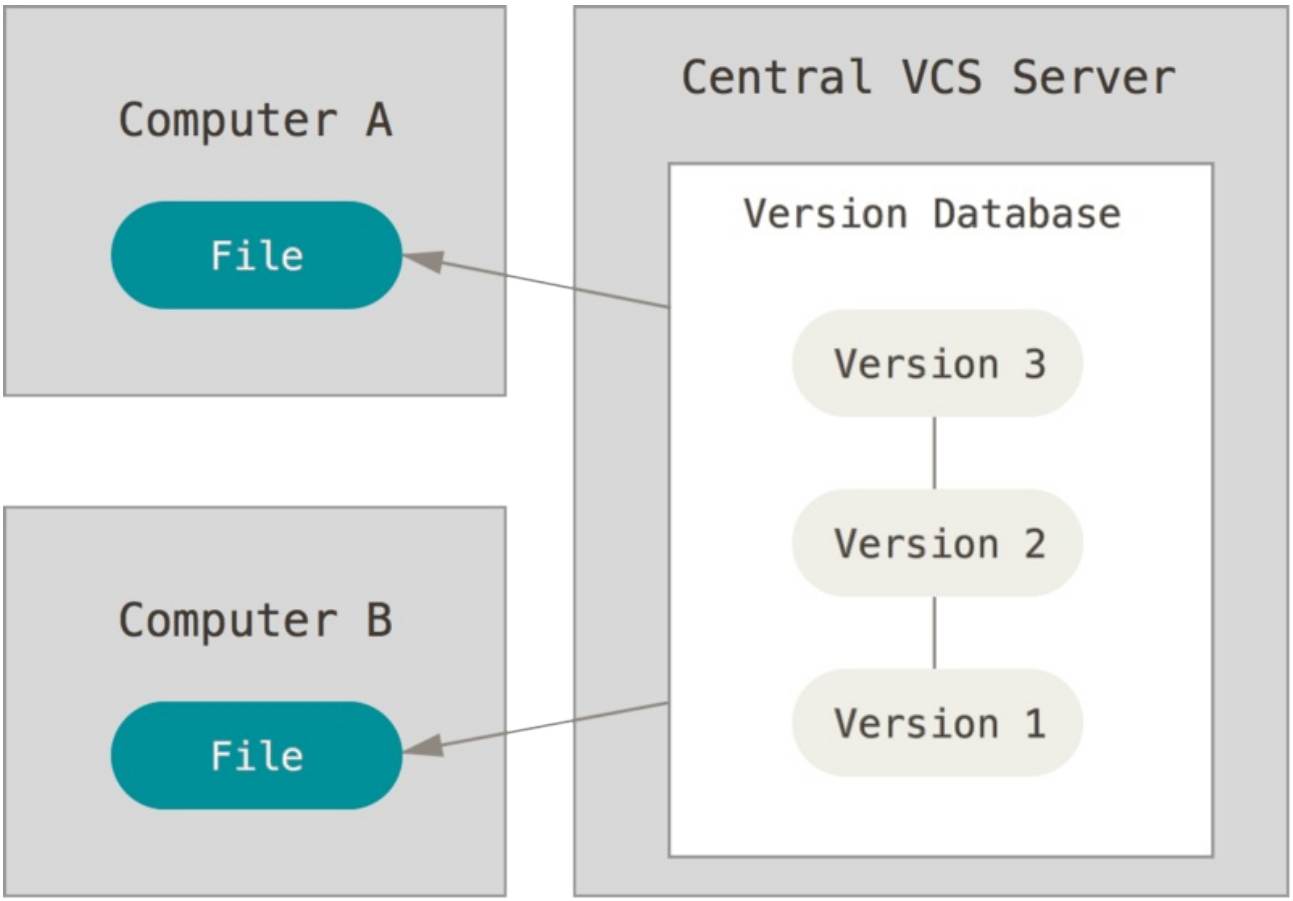


*Figure 3 : Diagramme de gestion de version centralisée.*

Figure 2. Gestion de version centralisée.

Ce schéma offre de nombreux avantages par rapport à la gestion de version locale. Par exemple, chacun sait jusqu'à un certain point ce que tous les autres sont en train de faire sur le projet. Les administrateurs ont un contrôle fin des permissions et il est beaucoup plus facile d'administrer un CVCS que de gérer des bases de données locales.

Cependant ce système a aussi de nombreux défauts. Le plus visible est le point unique de panne que le serveur centralisé représente. Si ce serveur est en panne pendant une heure, alors durant cette heure, aucun client ne peut collaborer ou enregistrer les modifications issues de son travail. Si le disque dur du serveur central se corrompt, et s'il n'y a pas eu de sauvegarde, vous perdez absolument tout de l'historique d'un projet en dehors des sauvegardes locales que les gens auraient pu réaliser sur leurs machines locales. Les systèmes de gestion de version locaux souffrent du même problème — dès qu'on a tout l'historique d'un projet sauvegardé à un endroit unique, on prend le risque de tout perdre.

## Les systèmes de gestion de version distribués

C'est à ce moment que les systèmes de gestion de version distribués entrent en jeu (DVCS en anglais pour *Distributed Version Control Systems*). Dans un DVCS (tel que Git, Mercurial, Bazaar ou Darcs), les clients n'extraient plus seulement la dernière version d'un fichier, mais ils dupliquent complètement le dépôt. Ainsi, si le serveur disparaît et si les systèmes collaboraient via ce serveur, n'importe quel dépôt d'un des clients peut être copié sur le serveur pour le restaurer. Chaque extraction devient une sauvegarde complète de toutes les données.

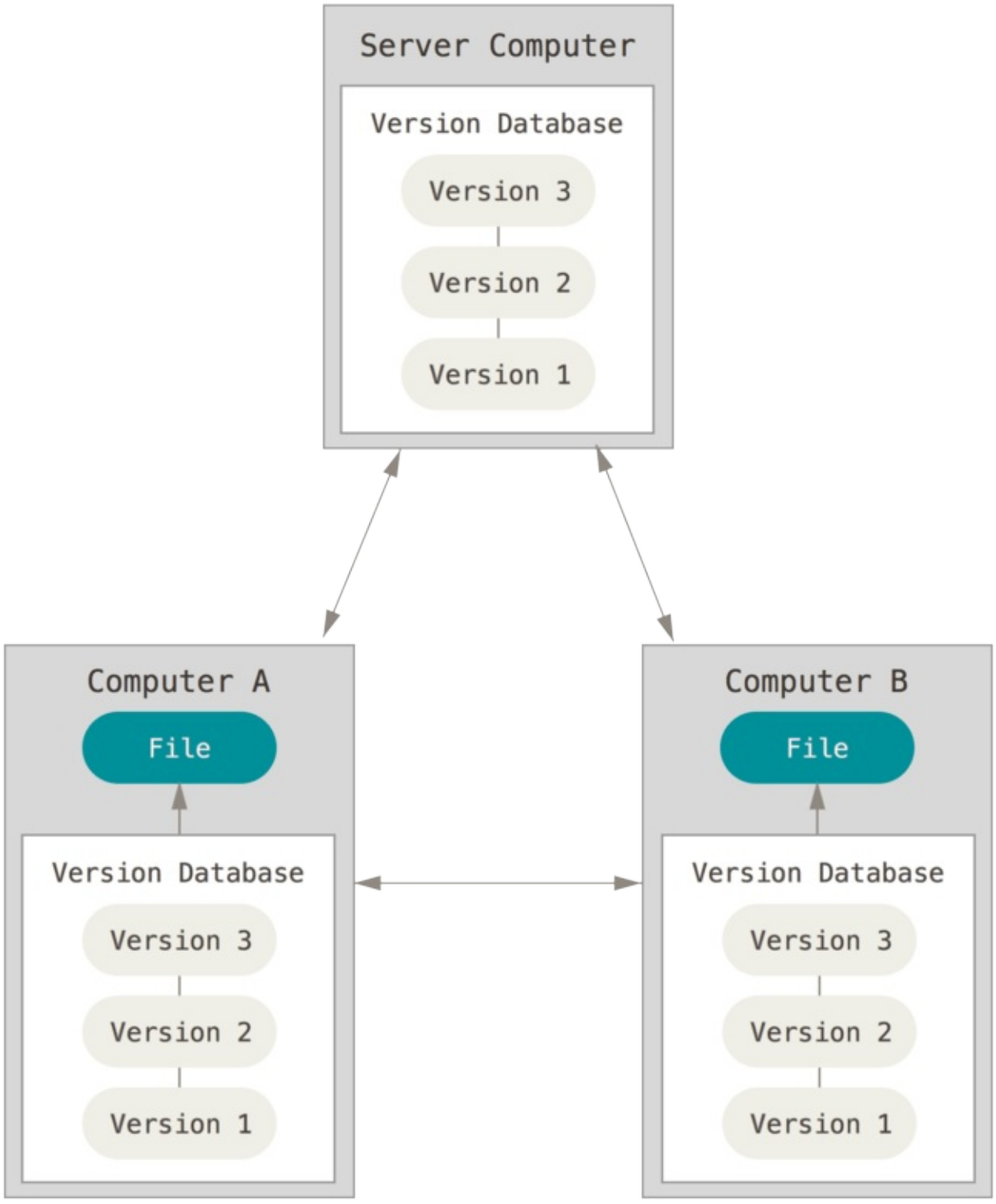


*Figure 4 : Diagramme de gestion de version distribuée*

Figure 3. Gestion de version distribuée.

De plus, un grand nombre de ces systèmes gère particulièrement bien le fait d'avoir plusieurs dépôts avec lesquels travailler, vous permettant de collaborer avec différents groupes de personnes de manières différentes simultanément dans le même projet. Cela permet la mise en place de différentes chaînes de traitement qui ne sont pas réalisables avec les systèmes centralisés, tels que les modèles hiérarchiques.

# Une rapide histoire de Git

Comme de nombreuses choses extraordinaires de la vie, Git est né avec une dose de destruction créative et de controverse houleuse. Le noyau Linux est un projet libre de grande envergure. Pour la plus grande partie de sa vie (1991–2002), les modifications étaient transmises sous forme de patchs et d'archives de fichiers. En 2002, le projet du noyau Linux commença à utiliser un DVCS propriétaire appelé BitKeeper.

En 2005, les relations entre la communauté développant le noyau Linux et la société en charge du développement de BitKeeper furent rompues, et le statut de gratuité de l'outil fut révoqué. Cela poussa la communauté du développement de Linux (et plus particulièrement Linus Torvalds, le créateur de Linux) à développer son propre outil en se basant sur les leçons apprises lors de l'utilisation de BitKeeper. Certains des objectifs du nouveau système étaient les suivants :

- vitesse ;
- conception simple ;
- support pour les développements non linéaires (milliers de branches parallèles) ;
- complètement distribué ;
- capacité à gérer efficacement des projets d'envergure tels que le noyau Linux (vitesse et compacité des données).

Depuis sa naissance en 2005, Git a évolué et mûri pour être facile à utiliser tout en conservant ses qualités initiales. Il est incroyablement rapide, il est très efficace pour de grands projets et il a un incroyable système de branches pour des développements non linéaires (voir Les branches avec Git).

# Rudiments de Git

Donc, qu'est-ce que Git en quelques mots ? Il est important de bien comprendre cette section, parce que si on comprend la nature de Git et les principes sur lesquels il repose, alors utiliser efficacement Git devient simple. Au cours de l'apprentissage de Git, essayez de libérer votre esprit de ce que vous pourriez connaître d'autres VCS, tels que Subversion et Perforce ; ce faisant, vous vous éviterez de petites confusions à l'utilisation de cet outil. Git enregistre et gère l'information très différemment des autres systèmes, même si l'interface utilisateur paraît similaire ; comprendre ces différences vous évitera des surprises.

## Des instantanés, pas des différences

La différence majeure entre Git et les autres VCS (Subversion et autres) réside dans la manière dont Git considère les données. Au niveau conceptuel, la plupart des autres systèmes gèrent l'information comme une liste de modifications de fichiers. Ces systèmes (CVS, Subversion, Perforce, Bazaar et autres) considèrent l'information qu'ils gèrent comme une liste de fichiers et les modifications effectuées sur chaque fichier dans le temps.

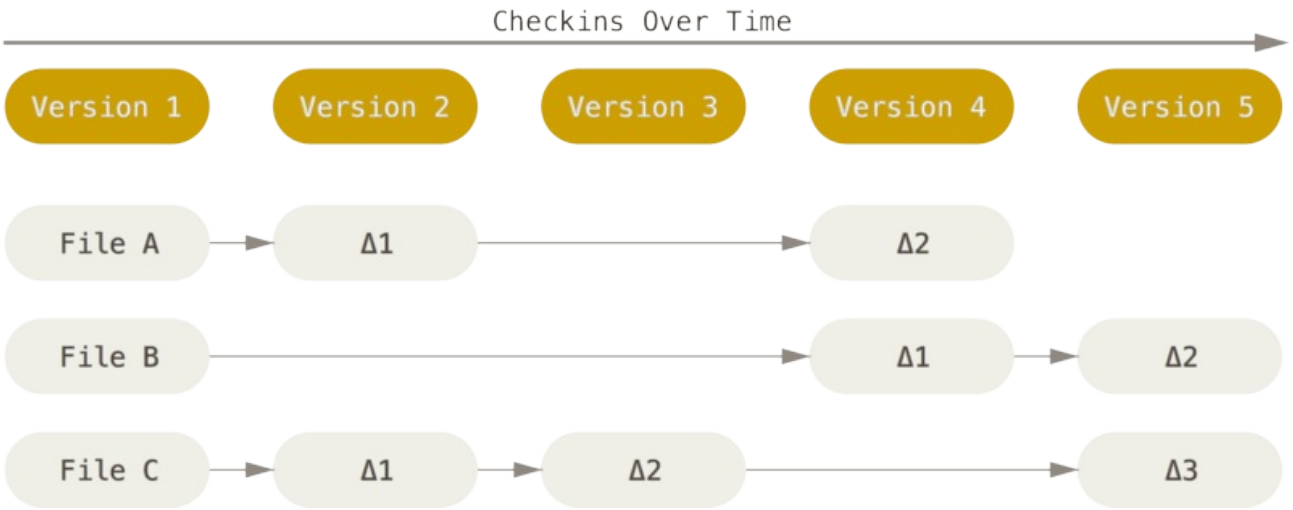


*Figure 5 : D'autres systèmes sauvent l'information comme des modifications sur des fichiers.*

Figure 4. D'autres systèmes sauvent l'information comme des modifications sur des fichiers.

Git ne gère pas et ne stocke pas les informations de cette manière. À la place, Git pense ses données plus comme un instantané d'un mini système de fichiers. À chaque fois que vous validez ou enregistrez l'état du projet dans Git, il prend effectivement un instantané du contenu de votre espace de travail à ce moment et enregistre une référence à cet instantané. Pour être efficace, si les fichiers n'ont pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original qu'il a déjà enregistré. Git pense ses données plus à la manière d'un **flux d'instantanés**.

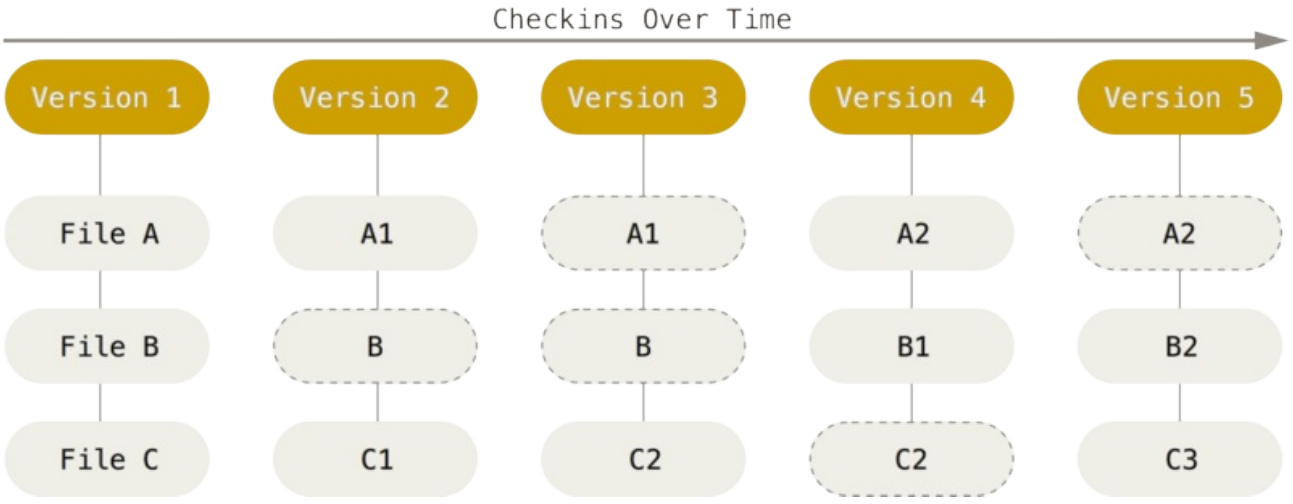


*Figure 6 : Git stocke les données comme des instantanés du projet au cours du temps.*

Figure 5. Git stocke les données comme des instantanés du projet au cours du temps.

C'est une distinction importante entre Git et quasiment tous les autres VCS. Git a reconsidéré quasiment tous les aspects de la gestion de version que la plupart des autres systèmes ont copiés des générations précédentes. Git ressemble beaucoup plus à un mini système de fichiers avec des outils incroyablement puissants construits dessus, plutôt qu'à un simple VCS. Nous explorerons les bénéfices qu'il y a à penser les données de cette manière quand nous aborderons la gestion de branches dans Les branches avec Git.

## Presque toutes les opérations sont locales

La plupart des opérations de Git ne nécessitent que des fichiers et ressources locaux — généralement aucune information venant d'un autre ordinateur du réseau n'est nécessaire. Si vous êtes habitué à un CVCS où toutes les opérations sont ralenties par la latence des échanges réseau, cet aspect de Git vous fera penser que les dieux de la vitesse ont octroyé leurs pouvoirs à Git. Comme vous disposez de l'historique complet du projet localement sur votre disque dur, la plupart des opérations semblent instantanées.

Par exemple, pour parcourir l'historique d'un projet, Git n'a pas besoin d'aller le chercher sur un serveur pour vous l'afficher ; il n'a qu'à simplement le lire directement dans votre base de données locale. Cela signifie que vous avez quasi-instantanément accès à l'historique du projet. Si vous souhaitez connaître les modifications introduites entre la version actuelle d'un fichier et son état un mois auparavant, Git peut rechercher l'état du fichier un mois auparavant et réaliser le calcul de différence, au lieu d'avoir à demander cette différence à un serveur ou de devoir récupérer l'ancienne version sur le serveur pour calculer la différence localement.

Cela signifie aussi qu'il y a très peu de choses que vous ne puissiez réaliser si vous n'êtes pas connecté ou hors VPN. Si vous voyagez en train ou en avion et voulez avancer votre travail, vous pouvez continuer à gérer vos versions sans soucis en attendant de pouvoir de nouveau vous connecter pour partager votre travail. Si vous êtes chez vous et ne pouvez avoir une liaison VPN avec votre entreprise, vous pouvez tout de même travailler. Pour de nombreux autres systèmes, faire de même est impossible ou au mieux très contraignant. Avec Perforce par exemple, vous ne pouvez pas faire grand-chose tant que vous n'êtes pas connecté au serveur. Avec Subversion ou CVS, vous pouvez éditer les fichiers, mais vous ne pourrez pas soumettre des modifications à votre base de données (car celle-ci est sur le serveur non accessible). Cela peut sembler peu important a priori, mais vous seriez étonné de découvrir quelle grande différence cela peut constituer à l'usage.

## Git gère l'intégrité

Dans Git, tout est vérifié par une somme de contrôle avant d'être stocké et par la suite cette somme de contrôle, signature unique, sert de référence. Cela signifie qu'il est impossible de modifier le contenu d'un fichier ou d'un répertoire sans que Git ne s'en aperçoive. Cette fonctionnalité est ancrée dans les fondations de Git et fait partie intégrante de sa philosophie. Vous ne pouvez pas perdre des données en cours de transfert ou corrompre un fichier sans que Git ne puisse le détecter.

Le mécanisme que Git utilise pour réaliser les sommes de contrôle est appelé une empreinte SHA-1. C'est une chaîne de caractères composée de 40 caractères hexadécimaux (de *0* à *9* et de *a* à *f*) calculée en fonction du contenu du fichier ou de la structure du répertoire considéré. Une empreinte SHA-1 ressemble à ceci :

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Vous trouverez ces valeurs à peu près partout dans Git car il les utilise pour tout. En fait, Git stocke tout non pas avec des noms de fichiers, mais dans la base de données Git indexée par ces valeurs.

## Généralement, Git ne fait qu'ajouter des données

Quand vous réalisez des actions dans Git, la quasi-totalité d'entre elles ne font qu'ajouter des données dans la base de données de Git. Il est très difficile de faire réaliser au système des actions qui ne soient pas réversibles ou de lui faire effacer des données d'une quelconque manière. Par contre, comme dans la plupart des systèmes de gestion de version, vous pouvez perdre ou corrompre des modifications qui n'ont pas encore été entrées en base ; mais dès que vous avez validé un instantané dans Git, il est très difficile de le perdre, spécialement si en plus vous synchronisez votre base de données locale avec un dépôt distant.

Cela fait de l'usage de Git un vrai plaisir, car on peut expérimenter sans danger de casser définitivement son projet. Pour une information plus approfondie sur la manière dont Git stocke ses données et comment récupérer des données qui pourraient sembler perdues, référez-vous à Annuler des actions.

## Les trois états

Un peu de concentration maintenant. Il est primordial de se souvenir de ce qui suit si vous souhaitez que le reste de votre apprentissage s'effectue sans difficulté. Git gère trois états dans lesquels les fichiers peuvent résider : validé, modifié et indexé. Validé signifie que les données sont stockées en sécurité dans votre base de données locale. Modifié signifie que vous avez modifié le fichier mais qu'il n'a pas encore été validé en base. Indexé signifie que vous avez marqué un fichier modifié dans sa version actuelle pour qu'il fasse partie du prochain instantané du projet.

Ceci nous mène aux trois sections principales d'un projet Git : le répertoire Git, le répertoire de travail et la zone d'index.

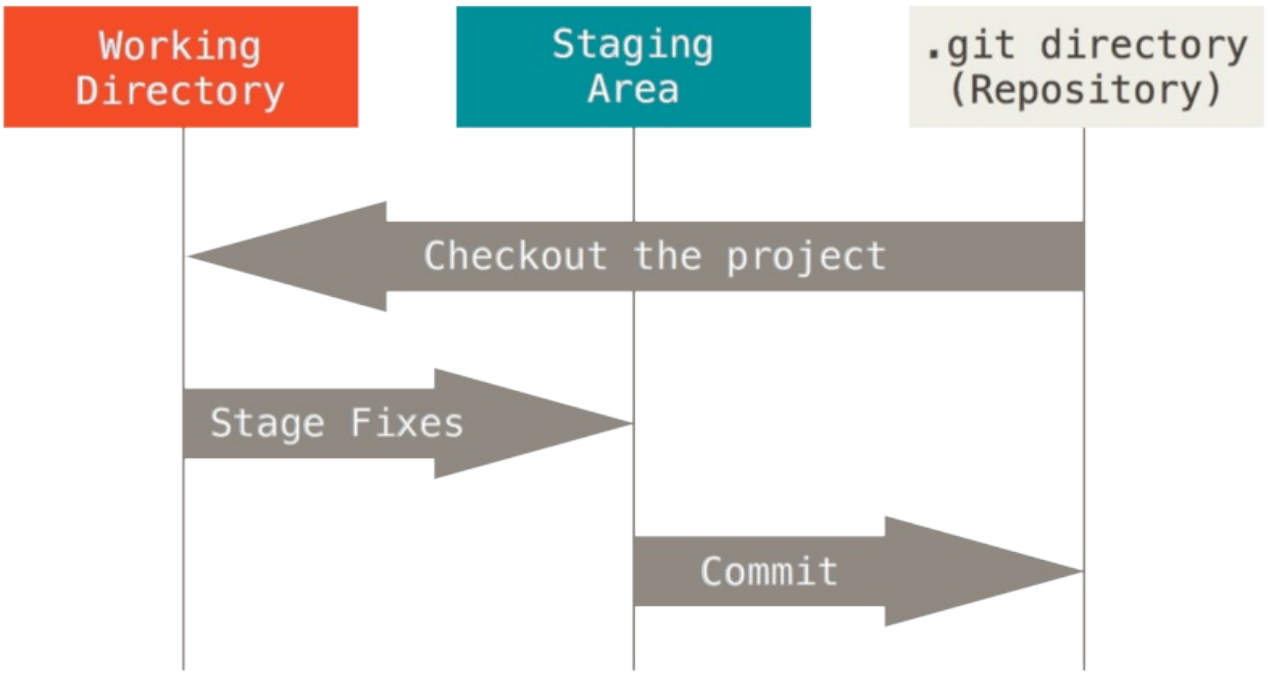


*Figure 7 : Répertoire de travail, zone d'index et répertoire Git.*

Figure 6. Répertoire de travail, zone d'index et répertoire Git.

Le répertoire Git est l'endroit où Git stocke les méta-données et la base de données des objets de votre projet. C'est la partie la plus importante de Git, et c'est ce qui est copié lorsque vous clonez un dépôt depuis un autre ordinateur.

Le répertoire de travail est une extraction unique d'une version du projet. Ces fichiers sont extraits depuis la base de données compressée dans le répertoire Git et placés sur le disque pour pouvoir être utilisés ou modifiés.

La zone d'index est un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané. On l'appelle aussi des fois la zone de préparation.

L'utilisation standard de Git se passe comme suit :

1. vous modifiez des fichiers dans votre répertoire de travail ;
2. vous indexez les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index ;
3. vous validez, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans la base de données du répertoire Git.

Si une version particulière d'un fichier est dans le répertoire Git, il est considéré comme validé. S'il est modifié mais a été ajouté dans la zone d'index, il est indexé. S'il a été modifié depuis le dernier instantané mais n'a pas été indexé, il est modifié. Dans Les bases de Git, vous en apprendrez plus sur ces états et comment vous pouvez en tirer parti ou complètement occulter la phase d'indexation.

# La ligne de commande

Il existe de nombreuses manières différentes d'utiliser Git. Il y a les outils originaux en ligne de commande et il y a de nombreuses interfaces graphiques avec des capacités variables. Dans ce livre, nous utiliserons Git en ligne de commande. Tout d'abord, la ligne de commande est la seule interface qui permet de lancer **toutes** les commandes Git - la plupart des interfaces graphiques simplifient l'utilisation en ne couvrant qu'un sous-ensemble des fonctionnalités de Git. Si vous savez comment utiliser la version en ligne de commande, vous serez à même de comprendre comment fonctionne la version graphique, tandis que l'inverse n'est pas nécessairement vrai. De plus, le choix d'un outil graphique est sujet à des goûts personnels, mais *tous* les utilisateurs auront les commandes en lignes installées et utilisables.

Nous considérons que vous savez ouvrir un Terminal sous Mac ou une invite de commandes ou Powershell sous Windows. Si ce n'est pas le cas, il va falloir tout d'abord vous renseigner sur ces applications pour pouvoir comprendre la suite des exemples et descriptions du livre.

# Installation de Git

Avant de commencer à utiliser Git, il faut qu'il soit disponible sur votre ordinateur. Même s'il est déjà installé, c'est probablement une bonne idée d'utiliser la dernière version disponible. Vous pouvez l'installer soit comme paquet ou avec un installateur, soit en téléchargeant le code et en le compilant par vous-même.

| | |
|---|---|
| | Ce livre a été écrit en utilisant Git version **2.0.0**. Bien que la plupart des commandes utilisées fonctionnent vraisemblablement encore avec d'anciennes version de Git, certaines peuvent agir différemment. Comme Git est particulièrement excellent pour préserver les compatibilités amont, toute version supérieure à 2.0 devrait fonctionner sans différence. |

## Installation sur Linux

Si vous voulez installer les outils basiques de Git sur Linux via un installateur binaire, vous pouvez généralement le faire au moyen de l'outil de gestion de paquet fourni avec votre distribution. Sur Fedora, par exemple, vous pouvez utiliser dnf :

```
$ dnf install git-all
```

Sur une distribution basée sur Debian, telle que Ubuntu, essayer apt-get :

```
$ apt-get install git-all
```

Pour plus d'options, des instructions d'installation sur différentes versions Unix sont disponibles sur le site web de Git, à http://git-scm.com/download/linux.

## Installation sur Mac

Il existe plusieurs méthodes d'installation de Git sur un Mac. La plus facile est probablement d'installer les *Xcode Command Line Tools*. Sur Mavericks (10.9) ou postérieur, vous pouvez simplement essayer de lancer *git* dans le terminal la première fois. S'il n'est pas déjà installé, il vous demandera de le faire.

Si vous souhaitez une version plus à jour, vous pouvez aussi l'installer à partir de l'installateur binaire. Un installateur de Git pour OS X est maintenu et disponible au téléchargement sur le site web de Git à http://git-scm.com/download/mac.


*Figure 8 : Installateur de Git pour OS X.*

Figure 7. Installateur de Git pour OS X.

Vous pouvez aussi l'installer comme sous-partie de l'installation de GitHub pour Mac. Leur outil Git graphique a une option pour installer les outils en ligne de commande. Vous pouvez télécharger cet outil depuis le site web de GitHub pour Mac, à http://mac.github.com.

## Installation sur Windows

Il existe aussi plusieurs manières d'installer Git sur Windows. L'application officielle est disponible au téléchargement sur le site web de Git. Rendez-vous sur http://git-scm.com/download/win et le téléchargement démarrera automatiquement. Notez que c'est un projet nommé *Git for Windows* (appelé aussi msysGit), qui est séparé de Git lui-même ; pour plus d'information, rendez-vous à http://msysgit.github.io/.

Une autre méthode facile pour installer Git est d'installer *Github for Windows*. L'installateur inclut une version en ligne de commande avec l'interface graphique. Elle fonctionne aussi avec Powershell et paramètre correctement les caches d'authentification et les réglages CRLF. Nous en apprendrons plus sur ces sujets plus tard, mais il suffit de savoir que ces options sont très utiles. Vous pouvez télécharger ceci depuis le site de *Github for Windows*, à l'adresse http://windows.github.com.

## Installation depuis les sources

Certains peuvent plutôt trouver utile d'installer Git depuis les sources car on obtient la version la plus récente. Les installateurs de version binaire tendent à être un peu en retard, même si Git a gagné en maturité ces dernières années, ce qui limite les évolutions.

Pour installer Git, vous avez besoin des bibliothèques suivantes : autotools, curl, zlib, openssl, expat, libiconv. Par exemple, si vous avez un système d'exploitation qui utilise dnf (tel que Fedora) ou apt-get (tel qu'un système basé sur Debian), vous pouvez utiliser l'une des commandes suivantes pour installer les dépendances :

```
$ dnf install curl-devel expat-devel gettext-devel \
    openssl-devel zlib-devel
```

```
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
    libz-dev libssl-dev
```

Quand vous avez toutes les dépendances nécessaires, vous pouvez poursuivre et télécharger la dernière version de Git depuis plusieurs sites. Vous pouvez l'obtenir via Kernel.org, à https://www.kernel.org/pub/software/scm/git, ou sur le miroir sur le site web GitHub à https://github.com/git/git/releases.

Puis, compiler et installer :

```
$ tar -zxf git-1.9.1.tar.gz
$ cd git-1.9.1
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Après ceci, vous pouvez obtenir Git par Git lui-même pour les mises à jour :

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

# Paramétrage à la première utilisation de Git

Maintenant que vous avez installé Git sur votre système, vous voudrez personnaliser votre environnement Git. Vous ne devriez avoir à réaliser ces réglages qu'une seule fois ; ils persisteront lors des mises à jour. Vous pouvez aussi les changer à tout instant en relançant les mêmes commandes.

Git contient un outil appelé `git config` pour vous permettre de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git. Ces variables peuvent être stockées dans trois endroits différents :

- Fichier `/etc/gitconfig` : Contient les valeurs pour tous les utilisateurs et tous les dépôts du système. Si vous passez l'option `--system` à `git config`, il lit et écrit ce fichier spécifiquement.
- Fichier `~/.gitconfig` : Spécifique à votre utilisateur. Vous pouvez forcer Git à lire et écrire ce fichier en passant l'option `--global`.
- Fichier `config` dans le répertoire Git (c'est-à-dire `.git/config`) du dépôt en cours d'utilisation : spécifique au seul dépôt en cours.

Chaque niveau surcharge le niveau précédent, donc les valeurs dans `.git/config` surchargent celles de `/etc/gitconfig`.

Sur les systèmes Windows, Git recherche le fichier `.gitconfig` dans le répertoire `$HOME` (`%USERPROFILE%` dans l'environnement natif de Windows) qui est `C:\Documents and Settings\$USER` ou `C:\Users\$USER` la plupart du temps, selon la version (`$USER` devient `%USERNAME%` dans l'environnement de Windows). Il recherche tout de même `/etc/gitconfig`, bien qu'il soit relatif à la racine MSys, qui se trouve où vous aurez décidé d'installer Git sur votre système Windows. Si vous utilisez une version 2.x ou supérieure de Git pour Windows, il y a aussi un fichier de configuration système à `C:\Documents and Settings\All Users\Application Data\Git\config` sur Windows XP, et dans `C:\ProgramData\Git\config` sur Windows Vista et supérieur. Ce fichier de configuration ne peut être modifié qu'avec la commande `git config -f <fichier>` en tant qu'administrateur.

## Votre identité

La première chose à faire après l'installation de Git est de renseigner votre nom et votre adresse de courriel. C'est une information importante car toutes les validations dans Git utilisent cette information et elle est indélébile dans toutes les validations que vous pourrez réaliser :

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

Encore une fois, cette étape n'est nécessaire qu'une fois si vous passez l'option `--global`, parce que Git utilisera toujours cette information pour tout ce que votre utilisateur fera sur ce système. Si vous souhaitez surcharger ces valeurs avec un nom ou une adresse de courriel différents pour un projet spécifique, vous pouvez lancer ces commandes sans option `--global` lorsque vous êtes dans ce projet.

De nombreux outils graphiques vous aideront à le faire la première fois que vous les lancerez.

## Votre éditeur de texte

À présent que votre identité est renseignée, vous pouvez configurer l'éditeur de texte qui sera utilisé quand Git vous demande de saisir un message. Par défaut, Git utilise l'éditeur configuré au niveau système, qui est généralement Vi ou Vim. Si vous souhaitez utiliser un éditeur de texte différent, comme Emacs, vous pouvez entrer ce qui suit :

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

| | |
|---|---|
| | Vim et Emacs sont des éditeurs de texte populaires chez les développeurs sur les systèmes à base Unix tels que Linux et Mac. Si vous n'êtes habitué à aucun de ces deux éditeurs ou utilisez un système Windows, il se peut que vous deviez chercher les instructions pour renseigner votre éditeur favori. Si vous ne renseignez pas un éditeur et ne connaissez pas Vim ou Emacs, vous risquez fort d'avoir des surprises lorsqu'ils démarreront. |

## Vérifier vos paramètres

Si vous souhaitez vérifier vos réglages, vous pouvez utiliser la commande `git config --list` pour lister tous les réglages que Git a pu trouver jusqu'ici :

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
…
```

Vous pourrez voir certains paramètres apparaître plusieurs fois car Git lit les mêmes paramètres depuis plusieurs fichiers ( `/etc/gitconfig` et `~/.gitconfig`, par exemple). Git utilise la dernière valeur pour chaque paramètre.

Vous pouvez aussi vérifier la valeur effective d'un paramètre particulier en tapant `git config <paramètre>` :

```
$ git config user.name
John Doe
```

# Obtenir de l’aide

Si vous avez besoin d’aide pour utiliser Git, il y a trois moyens d’obtenir les pages de manuel pour toutes les commandes de Git :

```
$ git help <commande>
$ git <commande> --help
$ man git-<commande>
```

Par exemple, vous pouvez obtenir la page de manuel pour la commande config en lançant :

```
$ git help config
```

Ces commandes sont vraiment sympathiques car vous pouvez y accéder depuis partout, y compris hors connexion. Si les pages de manuel et ce livre ne sont pas suffisants, vous pouvez essayer les canaux `#git` ou `#github` sur le serveur IRC Freenode (irc.freenode.net). Ces canaux sont régulièrement fréquentés par des centaines de personnes qui ont une bonne connaissance de Git et sont souvent prêtes à aider.

# Résumé

Vous devriez avoir à présent une compréhension initiale de ce que Git est et en quoi il est différent des CVCS que vous pourriez déjà avoir utilisés. Vous devriez aussi avoir une version de Git en état de fonctionnement sur votre système, paramétrée avec votre identité. Il est temps d’apprendre les bases d’utilisation de Git.

# Les bases de Git


- Démarrer un dépôt Git
  - Initialisation d'un dépôt Git dans un répertoire existant
  - Cloner un dépôt existant
- Enregistrer des modifications dans le dépôt
  - Vérifier l'état des fichiers
  - Placer de nouveaux fichiers sous suivi de version
  - Indexer des fichiers modifiés
  - Statut court
  - Ignorer des fichiers
  - Inspecter les modifications indexées et non indexées
  - Valider vos modifications
  - Passer l'étape de mise en index
  - Effacer des fichiers
  - Déplacer des fichiers
- Visualiser l'historique des validations
  - Limiter la longueur de l'historique
- Annuler des actions
  - Désindexer un fichier déjà indexé
  - Réinitialiser un fichier modifié
- Travailler avec des dépôts distants
  - Afficher les dépôts distants
  - Ajouter des dépôts distants
  - Récupérer et tirer depuis des dépôts distants
  - Pousser son travail sur un dépôt distant
  - Inspecter un dépôt distant
  - Retirer et renommer des dépôts distants
- Étiquetage
  - Lister vos étiquettes
  - Créer des étiquettes
  - Les étiquettes annotées
  - Les étiquettes légères
  - Étiqueter après coup
  - Partager les étiquettes
  - Extraire une étiquette
- Les alias Git
- Résumé


Si vous ne deviez lire qu'un chapitre avant de commencer à utiliser Git, c'est celui-ci. Ce chapitre couvre les commandes de base nécessaires pour réaliser la vaste majorité des activités avec Git. À la fin de ce chapitre, vous devriez être capable de configurer et initialiser un dépôt, commencer et arrêter le suivi de version de fichiers, d'indexer et valider des modifications. Nous vous montrerons aussi comment paramétrer Git pour qu'il ignore certains fichiers ou patrons de fichiers, comment revenir sur les erreurs rapidement et facilement, comment parcourir l'historique de votre projet et voir les modifications entre deux validations, et comment pousser et tirer les modifications avec des dépôts distants.

# Démarrer un dépôt Git

Vous pouvez principalement démarrer un dépôt Git de deux manières. La première consiste à prendre un projet ou un répertoire existant et à l'importer dans Git. La seconde consiste à cloner un dépôt Git existant sur un autre serveur.

### Initialisation d'un dépôt Git dans un répertoire existant

Si vous commencez à suivre un projet existant dans Git, vous n'avez qu'à vous positionner dans le répertoire du projet et saisir :

```
$ git init
```

Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt — un squelette de dépôt Git. Pour l'instant, aucun fichier n'est encore versionné. (Cf. Les tripes de Git pour plus d'information sur les fichiers contenus dans le répertoire `.git` que vous venez de créer.)

Si vous souhaitez démarrer le contrôle de version sur des fichiers existants (par opposition à un répertoire vide), vous devrez probablement suivre ces fichiers et faire un commit initial. Vous pouvez le réaliser avec quelques commandes `add` qui spécifient les fichiers que vous souhaitez suivre, suivies par un `git commit` :

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

### Cloner un dépôt existant

Si vous souhaitez obtenir une copie d'un dépôt Git existant — par exemple, un projet auquel vous aimeriez contribuer — la commande dont vous avez besoin s'appelle `git clone`. Si vous êtes familier avec d'autres systèmes de gestion de version tels que Subversion, vous noterez que la commande est `clone` et non `checkout`. C'est une distinction importante — Git reçoit une copie de quasiment toutes les données dont le serveur dispose. Toutes les versions de tous les fichiers pour l'historique du projet sont téléchargées quand vous lancez `git clone`. En fait, si le disque du serveur se corrompt, vous pouvez utiliser n'importe quel clone pour remettre le serveur dans l'état où il était au moment du clonage (vous pourriez perdre quelques paramètres du serveur, mais toutes les données sous gestion de version seraient récupérées — cf. Installation de Git sur un serveur pour de plus amples détails).

Vous clonez un dépôt avec `git clone [url]`. Par exemple, si vous voulez cloner la bibliothèque logicielle Git appelée libgit2, vous pouvez le faire de la manière suivante :

```
$ git clone https://github.com/libgit2/libgit2
```

Ceci crée un répertoire nommé "libgit2", initialise un répertoire `.git` à l'intérieur, récupère toutes les données de ce dépôt, et extrait une copie de travail de la dernière version. Si vous examinez le nouveau répertoire `libgit2`, vous y verrez les fichiers du projet, prêts à être modifiés ou utilisés. Si vous souhaitez cloner le dépôt dans un répertoire nommé différemment, vous pouvez spécifier le nom dans une option supplémentaire de la ligne de commande :

```
$ git clone https://github.com/libgit2/libgit2 monlibgit2
```

Cette commande réalise la même chose que la précédente, mais le répertoire cible s'appelle `monlibgit2`.

Git dispose de différents protocoles de transfert que vous pouvez utiliser. L'exemple précédent utilise le protocole `https://`, mais vous pouvez aussi voir `git://` ou `utilisateur@serveur:/chemin.git`, qui utilise le protocole de transfert SSH. Installation de Git sur un serveur introduit toutes les options disponibles pour mettre en place un serveur Git, ainsi que leurs avantages et inconvénients.

## Enregistrer des modifications dans le dépôt

Vous avez à présent un dépôt Git valide et une extraction ou copie de travail du projet. Vous devez faire quelques modifications et valider des instantanés de ces modifications dans votre dépôt chaque fois que votre projet atteint un état que vous souhaitez enregistrer.

Souvenez-vous que chaque fichier de votre copie de travail peut avoir deux états : sous suivi de version ou non suivi. Les fichiers suivis sont les fichiers qui appartenaient déjà au dernier instantané ; ils peuvent être inchangés, modifiés ou indexés. Tous les autres fichiers sont non suivis — tout fichier de votre copie de travail qui n'appartenait pas à votre dernier instantané et n'a pas été indexé. Quand vous clonez un dépôt pour la première fois, tous les fichiers seront sous suivi de version et inchangés car Git vient tout juste de les extraire et vous ne les avez pas encore édités.

Au fur et à mesure que vous éditez des fichiers, Git les considère comme modifiés, car vous les avez modifiés depuis le dernier instantané. Vous **indexez** ces fichiers modifiés et vous enregistrez toutes les modifications indexées, puis ce cycle se répète.

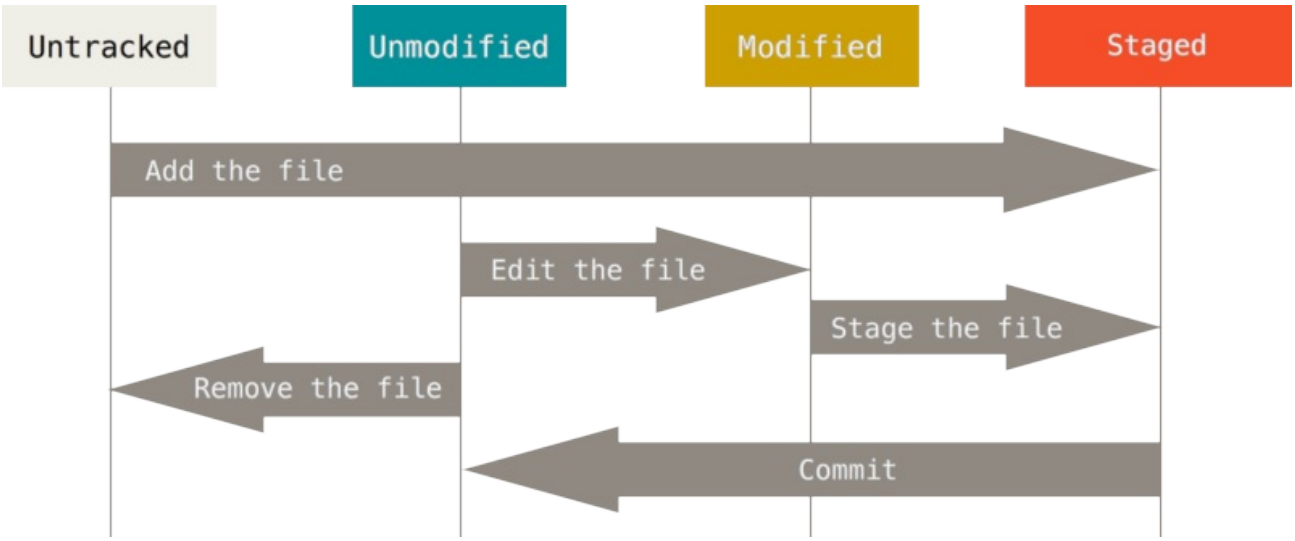


*Figure 9 : Le cycle de vie des états des fichiers.*

Figure 8. Le cycle de vie des états des fichiers.

## Vérifier l'état des fichiers

L'outil principal pour déterminer quels fichiers sont dans quel état est la commande `git status`. Si vous lancez cette commande juste après un clonage, vous devriez voir ce qui suit :

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
rien à valider, la copie de travail est propre
```

Ce message signifie que votre copie de travail est propre, en d'autres termes, aucun fichier suivi n'a été modifié. Git ne voit pas non plus de fichiers non-suivis, sinon ils seraient listés ici. Enfin, la commande vous indique sur quelle branche vous êtes. Pour l'instant, c'est toujours "master", qui correspond à la valeur par défaut ; nous ne nous en soucierons pas maintenant. Dans Les branches avec Git, nous parlerons plus en détail des branches et des références.

Supposons que vous souhaitez ajouter un nouveau fichier au projet, un simple fichier LISEZMOI. Si le fichier n'existait pas auparavant, et si vous lancez `git status`, vous voyez votre fichier non suivi comme suit :

```
$ echo 'Mon Projet' > LISEZMOI
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        LISEZMOI
```

```
aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez "git add" pour les suivre)
```

Vous pouvez constater que votre nouveau fichier `LISEZMOI` n'est pas en suivi de version, car il apparaît dans la section « Fichiers non suivis » de l'état de la copie de travail. « non suivi » signifie simplement que Git détecte un fichier qui n'était pas présent dans le dernier instantané ; Git ne le placera sous suivi de version que quand vous lui indiquerez de le faire. Ce comportement permet de ne pas placer accidentellement sous suivi de version des fichiers binaires générés ou d'autres fichiers que vous ne voulez pas inclure. Mais vous voulez inclure le fichier `LISEZMOI` dans l'instantané, alors commençons à suivre ce fichier.

## Placer de nouveaux fichiers sous suivi de version

Pour commencer à suivre un nouveau fichier, vous utilisez la commande `git add`. Pour commencer à suivre le fichier `LISEZMOI`, vous pouvez entrer ceci :

```
$ git add LISEZMOI
```

Si vous lancez à nouveau la commande `git status`, vous pouvez constater que votre fichier `LISEZMOI` est maintenant suivi et indexé :

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : LISEZMOI
```

Vous pouvez affirmer qu'il est indexé car il apparaît dans la section « Modifications qui seront validées ». Si vous validez à ce moment, la version du fichier à l'instant où vous lancez `git add` est celle qui sera dans l'historique des instantanés. Vous pouvez vous souvenir que lorsque vous avez précédemment lancé `git init`, vous avez ensuite lancé `git add (fichiers)` — c'était bien sûr pour commencer à placer sous suivi de version les fichiers de votre répertoire de travail. La commande `git add` accepte en paramètre un chemin qui correspond à un fichier ou un répertoire ; dans le cas d'un répertoire, la commande ajoute récursivement tous les fichiers de ce répertoire.

## Indexer des fichiers modifiés

Maintenant, modifions un fichier qui est déjà sous suivi de version. Si vous modifiez le fichier sous suivi de version appelé `CONTRIBUTING.md` et que vous lancez à nouveau votre commande `git status`, vous verrez ceci :

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : LISEZMOI

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md
```

Le fichier `CONTRIBUTING.md` apparaît sous la section nommée « Modifications qui ne seront pas validées » ce qui signifie que le fichier sous suivi de version a été modifié dans la copie de travail mais n'est pas encore indexé. Pour l'indexer, il faut lancer la commande `git add`. `git add` est une commande multi-usage — elle peut être utilisée pour placer un fichier sous suivi de

version, pour indexer un fichier ou pour d'autres actions telles que marquer comme résolus des conflits de fusion de fichiers. Sa signification s'approche plus de « ajouter ce contenu pour la prochaine validation » que de « ajouter ce contenu au projet ». Lançons maintenant `git add` pour indexer le fichier `CONTRIBUTING.md`, et relançons la commande `git status` :

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : LISEZMOI
        modifié :         CONTRIBUTING.md
```

À présent, les deux fichiers sont indexés et feront partie de la prochaine validation. Mais supposons que vous souhaitiez apporter encore une petite modification au fichier `CONTRIBUTING.md` avant de réellement valider la nouvelle version. Vous l'ouvrez à nouveau, réalisez la petite modification et vous voilà prêt à valider. Néanmoins, vous lancez `git status` une dernière fois :

```
$ vim CONTRIBUTING.md
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : LISEZMOI
        modifié :         CONTRIBUTING.md

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md
```

Que s'est-il donc passé ? À présent, `CONTRIBUTING.md` apparaît à la fois comme indexé et non indexé. En fait, Git indexe un fichier dans son état au moment où la commande `git add` est lancée. Si on valide les modifications maintenant, la version de `CONTRIBUTING.md` qui fera partie de l'instantané est celle correspondant au moment où la commande `git add CONTRIBUTING.md` a été lancée, et non la version actuellement présente dans la copie de travail au moment où la commande `git commit` est lancée. Si le fichier est modifié après un `git add`, il faut relancer `git add` pour prendre en compte l'état actuel de la copie de travail :

```
$ git add CONTRIBUTING.md
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : LISEZMOI
        modifié :         CONTRIBUTING.md
```

## Statut court

Bien que `git status` soit informatif, il est aussi plutôt verbeux. Git a aussi une option de status court qui permet de voir les modifications de façon plus compacte. Si vous lancez `git status -s` ou `git status --short`, vous obtenez une information bien plus simple.

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
```

```
?? LICENSE.txt
```

Les nouveaux fichiers qui ne sont pas suivis sont précédés de `??`, les fichiers nouveaux et indexés sont précédés de `A`, les fichiers modifiés de `M` et ainsi de suite. Il y a deux colonnes d'état - celle de gauche indique l'état de l'index et celle de droite l'état du dossier de travail. Donc l'exemple ci-dessus indique que le fichier `README` est modifié dans le répertoire de travail mais n'est pas encore indexé, tandis que le fichier `lib/simplegit.rb` est modifié et indexé. Le fichier `Rakefile` a été modifié, indexé puis modifié à nouveau, de sorte qu'il a des modifications à la fois indexées et non-indexées.

## Ignorer des fichiers

Il apparaît souvent qu'un type de fichiers présent dans la copie de travail ne doit pas être ajouté automatiquement ou même ne doit pas apparaître comme fichier potentiel pour le suivi de version. Ce sont par exemple des fichiers générés automatiquement tels que les fichiers de journaux ou de sauvegardes produits par l'outil que vous utilisez. Dans un tel cas, on peut énumérer les patrons de noms de fichiers à ignorer dans un fichier `.gitignore`. Voici ci-dessous un exemple de fichier `.gitignore` :

```
$ cat .gitignore
*.[oa]
*~
```

La première ligne ordonne à Git d'ignorer tout fichier se terminant en `.o` ou `.a` — des fichiers objet ou archive qui sont généralement produits par la compilation d'un programme. La seconde ligne indique à Git d'ignorer tous les fichiers se terminant par un tilde ( `~` ), ce qui est le cas des noms des fichiers temporaires pour de nombreux éditeurs de texte tels qu'Emacs. On peut aussi inclure un répertoire `log`, `tmp` ou `pid`, ou le répertoire de documentation générée automatiquement, ou tout autre fichier. Renseigner un fichier `.gitignore` avant de commencer à travailler est généralement une bonne idée qui évitera de valider par inadvertance des fichiers qui ne doivent pas apparaître dans le dépôt Git.

Les règles de construction des patrons à placer dans le fichier `.gitignore` sont les suivantes :

- les lignes vides ou commençant par `#` sont ignorées ;
- les patrons standards de fichiers sont utilisables ;
- si le patron se termine par une barre oblique ( `/` ), il indique un répertoire ;
- un patron commençant par un point d'exclamation ( `!` ) indique des fichiers à inclure malgré les autres règles.

Les patrons standards de fichiers sont des expressions régulières simplifiées utilisées par les shells. Un astérisque ( `*` ) correspond à un ou plusieurs caractères ; `[abc]` correspond à un des trois caractères listés dans les crochets, donc a ou b ou c ; un point d'interrogation ( `?` ) correspond à un unique caractère ; des crochets entourant des caractères séparés par un tiret ( `[0-9]` ) correspond à un caractère dans l'intervalle des deux caractères indiqués, donc ici de 0 à 9. Vous pouvez aussi utiliser deux astérisques pour indiquer une série de répertoires inclus ; `a/**/z` correspond donc à `a/z`, `a/b/z`, `a/b/c/z` et ainsi de suite.

Voici un autre exemple de fichier `.gitignore` :

```
# pas de fichier .a
*.a

# mais suivre lib.a malgré la règle précédente
!lib.a

# ignorer uniquement le fichier TODO à la racine du projet
/TODO

# ignorer tous les fichiers dans le répertoire build
build/

# ignorer doc/notes.txt, mais pas doc/server/arch.txt
doc/*.txt
```

```
# ignorer tous les fichiers .txt sous le répertoire doc/
doc/**/*.txt
```

| | |
|---|---|
| | GitHub maintient une liste assez complète d'exemples de fichiers `.gitignore` correspondant à de nombreux types de projets et langages. Voir https://github.com/github/gitignore pour obtenir un point de départ pour votre projet. |

## Inspecter les modifications indexées et non indexées

Si le résultat de la commande `git status` est encore trop vague — lorsqu'on désire savoir non seulement quels fichiers ont changé mais aussi ce qui a changé dans ces fichiers — on peut utiliser la commande `git diff`. Cette commande sera traitée en détail plus loin ; mais elle sera vraisemblablement utilisée le plus souvent pour répondre aux questions suivantes : qu'est-ce qui a été modifié mais pas encore indexé ? Quelle modification a été indexée et est prête pour la validation ? Là où `git status` répond de manière générale à ces questions, `git diff` montre les lignes exactes qui ont été ajoutées, modifiées ou effacées — le patch en somme.

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : LISEZMOI

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md
```

Pour visualiser ce qui a été modifié mais pas encore indexé, tapez `git diff` sans autre argument :

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Cette commande compare le contenu du répertoire de travail avec la zone d'index. Le résultat vous indique les modifications réalisées mais non indexées.

Si vous souhaitez visualiser les modifications indexées qui feront partie de la prochaine validation, vous pouvez utiliser `git diff --cached` (avec les versions 1.6.1 et supérieures de Git, vous pouvez aussi utiliser `git diff --staged`, qui est plus mnémotechnique). Cette commande compare les fichiers indexés et le dernier instantané :

```
$ git diff --staged
diff --git a/LISEZMOI b/LISEZMOI
new file mode 100644
index 0000000..1e17b0c
```

```
--- /dev/null
+++ b/LISEZMOI
@@ -0,0 +1 @@
+Mon Projet
```

Il est important de noter que `git diff` ne montre pas les modifications réalisées depuis la dernière validation — seulement les modifications qui sont non indexées. Cela peut introduire une confusion car si tous les fichiers modifiés ont été indexés, `git diff` n'indiquera aucun changement.

Par exemple, si vous indexez le fichier `CONTRIBUTING.md` et l'éditez ensuite, vous pouvez utiliser `git diff` pour visualiser les modifications indexées et non indexées de ce fichier. Si l'état est le suivant :

```
$ git add CONTRIBUTING.md
$ echo 'ligne de test' >> CONTRIBUTING.md
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        nouveau fichier : CONTRIBUTING.md

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md
```

À présent, vous pouvez utiliser `git diff` pour visualiser les modifications non indexées :

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+ligne de test
```

et `git diff --cached` pour visualiser ce qui a été indexé jusqu'à maintenant :

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

| | |
|---|---|
| | Git Diff dans un outil externe<br>Nous allons continuer à utiliser la commande `git diff` de différentes manières par la suite. Il existe une autre manière de visualiser les différences si vous préférez un outil graphique ou externe. Si vous lancez `git difftool` |

| | |
|---|---|
| | au lieu de `git diff`, vous pourrez visualiser les différences grâce à une application telle que Araxis, emerge, vimdiff ou autre. Lancez `git difftool --tool-help` pour connaître les applications disponibles sur votre système. |

## Valider vos modifications

Maintenant que votre zone d'index est dans l'état désiré, vous pouvez valider vos modifications. Souvenez-vous que tout ce qui est encore non indexé — tous les fichiers qui ont été créés ou modifiés mais n'ont pas subi de `git add` depuis que vous les avez modifiés — ne feront pas partie de la prochaine validation. Ils resteront en tant que fichiers modifiés sur votre disque.

Dans notre cas, la dernière fois que vous avez lancé `git status`, vous avez vérifié que tout était indexé, et vous êtes donc prêt à valider vos modifications. La manière la plus simple de valider est de taper `git commit` :

```
$ git commit
```

Cette action lance votre éditeur par défaut (qui est paramétré par la variable d'environnement `$EDITOR` de votre shell — habituellement vim ou Emacs, mais vous pouvez le paramétrer spécifiquement pour Git en utilisant la commande `git config --global core.editor` comme nous l'avons vu au Démarrage rapide).

L'éditeur affiche le texte suivant :

```
# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
# Sur la branche master
# Votre branche est à jour avec 'origin/master'.
#
# Modifications qui seront validées :
#       nouveau fichier : LISEZMOI
#       modifié :         CONTRIBUTING.md
#
```

Vous constatez que le message de validation par défaut contient une ligne vide suivie en commentaire par le résultat de la commande `git status`. Vous pouvez effacer ces lignes de commentaire et saisir votre propre message de validation, ou vous pouvez les laisser en place pour vous aider à vous rappeler ce que vous êtes en train de valider (pour un rappel plus explicite de ce que vous avez modifié, vous pouvez aussi passer l'option `-v` à la commande `git commit`). Cette option place le résultat du diff en commentaire dans l'éditeur pour vous permettre de visualiser exactement ce que vous avez modifié. Quand vous quittez l'éditeur (après avoir sauvegardé le message), Git crée votre *commit* avec ce message de validation (après avoir retiré les commentaires et le diff).

Autrement, vous pouvez spécifier votre message de validation en ligne avec la commande `git commit` en le saisissant après l'option `-m`, comme ceci :

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 LISEZMOI
```

À présent, vous avez créé votre premier *commit* ! Vous pouvez constater que le *commit* vous fournit quelques informations sur lui-même : sur quelle branche vous avez validé ( `master` ), quelle est sa somme de contrôle SHA-1 ( `463dc4f` ), combien de fichiers ont été modifiés, et quelques statistiques sur les lignes ajoutées et effacées dans ce *commit*.

Souvenez-vous que la validation enregistre l'instantané que vous avez préparé dans la zone d'index. Tout ce que vous n'avez pas indexé est toujours en état modifié ; vous pouvez réaliser une nouvelle validation pour l'ajouter à l'historique. À chaque validation, vous enregistrez un instantané du projet en forme de jalon auquel vous pourrez revenir ou avec lequel comparer votre travail ultérieur.

## Passer l'étape de mise en index

Bien qu'il soit incroyablement utile de pouvoir organiser les *commits* exactement comme on l'entend, la gestion de la zone d'index est parfois plus complexe que nécessaire dans le cadre d'une utilisation normale. Si vous souhaitez éviter la phase de placement des fichiers dans la zone d'index, Git fournit un raccourci très simple. L'ajout de l'option `-a` à la commande `git commit` ordonne à Git de placer automatiquement tout fichier déjà en suivi de version dans la zone d'index avant de réaliser la validation, évitant ainsi d'avoir à taper les commandes `git add` :

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Notez bien que vous n'avez pas eu à lancer `git add` sur le fichier `CONTRIBUTING.md` avant de valider.

## Effacer des fichiers

Pour effacer un fichier de Git, vous devez l'éliminer des fichiers en suivi de version (plus précisément, l'effacer dans la zone d'index) puis valider. La commande `git rm` réalise cette action mais efface aussi ce fichier de votre copie de travail de telle sorte que vous ne le verrez pas réapparaître comme fichier non suivi en version à la prochaine validation.

Si vous effacez simplement le fichier dans votre copie de travail, il apparaît sous la section « Modifications qui ne seront pas validées » (c'est-à-dire, *non indexé*) dans le résultat de `git status` :

```
$ rm PROJECTS.md
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui ne seront pas validées :
  (utilisez "git add/rm <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        supprimé :        PROJECTS.md

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
```

Ensuite, si vous lancez `git rm`, l'effacement du fichier est indexé :

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        supprimé :        PROJECTS.md
```

Lors de la prochaine validation, le fichier sera absent et non-suivi en version. Si vous avez auparavant modifié et indexé le fichier, son élimination doit être forcée avec l'option `-f`. C'est une mesure de sécurité pour empêcher un effacement accidentel de données qui n'ont pas encore été enregistrées dans un instantané et qui seraient définitivement perdues.

Un autre scénario serait de vouloir abandonner le suivi de version d'un fichier tout en le conservant dans la copie de travail. Ceci est particulièrement utile lorsqu'on a oublié de spécifier un patron dans le fichier `.gitignore` et on a accidentellement indexé un fichier, tel qu'un gros fichier de journal ou une série d'archives de compilation `.a`. Pour réaliser ce scénario, utilisez l'option `--cached` :

```
$ git rm --cached LISEZMOI
```

Vous pouvez spécifier des noms de fichiers ou de répertoires, ou des patrons de fichiers à la commande `git rm`. Cela signifie que vous pouvez lancer des commandes telles que :

```
$ git rm log/\*.log
```

Notez bien la barre oblique inverse ( `\` ) devant `*`. Il est nécessaire d'échapper le caractère `*` car Git utilise sa propre expansion de nom de fichier en addition de l'expansion du shell. Ce caractère d'échappement doit être omis sous Windows si vous utilisez le terminal système. Cette commande efface tous les fichiers avec l'extension `.log` présents dans le répertoire `log/`. Vous pouvez aussi lancer une commande telle que :

```
$ git rm \*~
```

Cette commande élimine tous les fichiers se terminant par `~`.

## Déplacer des fichiers

À la différence des autres VCS, Git ne suit pas explicitement les mouvements des fichiers. Si vous renommez un fichier suivi par Git, aucune méta-donnée indiquant le renommage n'est stockée par Git. Néanmoins, Git est assez malin pour s'en apercevoir après coup — la détection de mouvement de fichier sera traitée plus loin.

De ce fait, que Git ait une commande `mv` peut paraître trompeur. Si vous souhaitez renommer un fichier dans Git, vous pouvez lancer quelque chose comme :

```
$ git mv nom_origine nom_cible
```

et cela fonctionne. En fait, si vous lancez quelque chose comme ceci et inspectez le résultat d'une commande `git status`, vous constaterez que Git gère le renommage de fichier :

```
$ git mv LISEZMOI.txt LISEZMOI
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        renommé :         LISEZMOI.txt -> LISEZMOI
```

Néanmoins, cela revient à lancer les commandes suivantes :

```
$ mv LISEZMOI.txt LISEZMOI
$ git rm LISEZMOI.txt
$ git add LISEZMOI
```

Git trouve implicitement que c'est un renommage, donc cela importe peu si vous renommez un fichier de cette manière ou avec la commande `mv`. La seule différence réelle est que `git mv` ne fait qu'une commande à taper au lieu de trois — c'est une commande de convenance. Le point principal est que vous pouvez utiliser n'importe quel outil pour renommer un fichier, et traiter les commandes `add` / `rm` plus tard, avant de valider la modification.

# Visualiser l'historique des validations

Après avoir créé plusieurs *commits* ou si vous avez cloné un dépôt ayant un historique de *commits*, vous souhaitez probablement revoir le fil des évènements. Pour ce faire, la commande `git log` est l'outil le plus basique et le plus puissant.

Les exemples qui suivent utilisent un projet très simple nommé `simplegit` utilisé pour les démonstrations. Pour récupérer le projet, lancez :

```
git clone https://github.com/schacon/simplegit-progit
```

Lorsque vous lancez `git log` dans le répertoire de ce projet, vous devriez obtenir un résultat qui ressemble à ceci :

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Par défaut, `git log` invoqué sans argument énumère en ordre chronologique inversé les *commits* réalisés. Cela signifie que les *commits* les plus récents apparaissent en premier. Comme vous le remarquez, cette commande indique chaque *commit* avec sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur, la date et le message du *commit*.

`git log` dispose d'un très grand nombre d'options permettant de paramétrer exactement ce que l'on cherche à voir. Nous allons détailler quelques-unes des plus utilisées.

Une des options les plus utiles est `-p`, qui montre les différences introduites entre chaque validation. Vous pouvez aussi utiliser `-2` qui limite la sortie de la commande aux deux entrées les plus récentes :

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

```
    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Cette option affiche la même information mais avec un diff suivant directement chaque entrée. C'est très utile pour des revues de code ou pour naviguer rapidement à travers l'historique des modifications qu'un collaborateur a apportées.

Vous pouvez aussi utiliser une liste d'options de résumé avec `git log`. Par exemple, si vous souhaitez visualiser des statistiques résumées pour chaque *commit*, vous pouvez utiliser l'option `--stat` :

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

Comme vous pouvez le voir, l'option `--stat` affiche sous chaque entrée de validation une liste des fichiers modifiés, combien de fichiers ont été changés et combien de lignes ont été ajoutées ou retirées dans ces fichiers. Elle ajoute un résumé des informations en fin de sortie. Une autre option utile est `--pretty`. Cette option modifie le journal vers un format différent. Quelques options incluses sont disponibles. L'option `oneline` affiche chaque *commit* sur une seule ligne, ce qui peut s'avérer utile lors de la revue d'un long journal. En complément, les options `short` (court), `full` (complet) et `fuller` (plus complet) montrent le résultat à peu de choses près dans le même format mais avec plus ou moins d'informations :

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

L'option la plus intéressante est `format` qui permet de décrire précisément le format de sortie. C'est spécialement utile pour générer des sorties dans un format facile à analyser par une machine — lorsqu'on spécifie intégralement et explicitement le format, on s'assure qu'il ne changera pas au gré des mises à jour de Git :

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Options utiles pour `git log --pretty=format` liste les options de formatage les plus utiles.

Table 1. Options utiles pour `git log --pretty=format`

| Option | Description du formatage |
|---|---|
| `%H` | Somme de contrôle du commit |
| `%h` | Somme de contrôle abrégée du commit |
| `%T` | Somme de contrôle de l'arborescence |
| `%t` | Somme de contrôle abrégée de l'arborescence |
| `%P` | Sommes de contrôle des parents |
| `%p` | Sommes de contrôle abrégées des parents |
| `%an` | Nom de l'auteur |
| `%ae` | E-mail de l'auteur |
| `%ad` | Date de l'auteur (au format de l'option -date=) |
| `%ar` | Date relative de l'auteur |
| `%cn` | Nom du validateur |
| `%ce` | E-mail du validateur |
| `%cd` | Date du validateur |
| `%cr` | Date relative du validateur |
| `%s` | Sujet |

Vous pourriez vous demander quelle est la différence entre *auteur* et *validateur*. L'*auteur* est la personne qui a réalisé initialement le travail, alors que le *validateur* est la personne qui a effectivement validé ce travail en gestion de version. Donc, si quelqu'un envoie un patch à un projet et un des membres du projet l'applique, les deux personnes reçoivent le crédit — l'écrivain en tant qu'auteur, et le membre du projet en tant que validateur. Nous traiterons plus avant de cette distinction dans le Git distribué.

Les options `oneline` et `format` sont encore plus utiles avec une autre option `log` appelée `--graph`. Cette option ajoute un joli graphe en caractères ASCII pour décrire l'historique des branches et fusions :

```
$ git log --pretty=format:"%h %s" --graph
```

```
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Ces options deviendront plus intéressantes quand nous aborderons les branches et les fusions dans le prochain chapitre.

Les options ci-dessus ne sont que des options simples de format de sortie de `git log` — il y en a de nombreuses autres. Options usuelles de `git log` donne une liste des options que nous avons traitées ainsi que d'autres options communément utilisées accompagnées de la manière dont elles modifient le résultat de la commande `log`.

Table 2. Options usuelles de `git log`

| Option | Description |
|---|---|
| `-p` | Affiche le patch appliqué par chaque commit |
| `--stat` | Affiche les statistiques de chaque fichier pour chaque commit |
| `--shortstat` | N'affiche que les ligne modifiées/insérées/effacées de l'option –stat |
| `--name-only` | Affiche la liste des fichiers modifiés après les informations du commit |
| `--name-status` | Affiche la liste des fichiers affectés accompagnés des informations d'ajout/modification/suppression |
| `--abbrev-commit` | N'affiche que les premiers caractères de la somme de contrôle SHA-1 |
| `--relative-date` | Affiche la date en format relatif (par exemple "2 weeks ago" : il y a deux semaines) au lieu du format de date complet |
| `--graph` | Affiche en caractères ASCII le graphe de branches et fusions en vis-à-vis de l'historique |
| `--pretty` | Affiche les *commits* dans un format alternatif. Les formats incluent `oneline`, `short`, `full`, `fuller`, et `format` (où on peut spécifier son propre format) |
| `--oneline` | Option de convenance correspondant à `--pretty=oneline --abbrev-commit` |

## Limiter la longueur de l'historique

En complément des options de formatage de sortie, `git log` est pourvu de certaines options de limitation utiles — des options qui permettent de restreindre la liste à un sous-ensemble de *commits*. Vous avez déjà vu une de ces options — l'option `-2` qui ne montre que les deux derniers *commits*. En fait, on peut utiliser `-<n>`, où `n` correspond au nombre de *commits* que l'on cherche à visualiser en partant des plus récents. En vérité, il est peu probable que vous utilisiez cette option, parce que Git injecte par défaut sa sortie dans un outil de pagination qui permet de la visualiser page à page.

Cependant, les options de limitation portant sur le temps, telles que `--since` (depuis) et `--until` (jusqu'à) sont très utiles. Par exemple, la commande suivante affiche la liste des *commits* des deux dernières semaines :

```
$ git log --since=2.weeks
```

Cette commande fonctionne avec de nombreux formats — vous pouvez indiquer une date spécifique (2008-01-05) ou une date relative au présent telle que "2 years 1 day 3 minutes ago".

Vous pouvez aussi restreindre la liste aux *commits* vérifiant certains critères de recherche. L'option `--author` permet de filtrer sur un auteur spécifique, et l'option `--grep` permet de chercher des mots clés dans les messages de validation. Notez que si vous spécifiez à la fois `--author` et `--grep`, la commande retournera seulement des *commits* correspondant simultanément aux deux critères.

Si vous souhaitez spécifier plusieurs options `--grep`, vous devez ajouter l'option `--all-match`, car par défaut ces commandes retournent les *commits* vérifiant au moins un critère de recherche.

Un autre filtre vraiment utile est l'option `-S` qui prend une chaîne de caractères et ne retourne que les *commits* qui introduisent des modifications qui ajoutent ou retirent du texte comportant cette chaîne. Par exemple, si vous voulez trouver la dernière validation qui a ajouté ou retiré une référence à une fonction spécifique, vous pouvez lancer :

```
$ git log -Snom_de_fonction
```

La dernière option vraiment utile à `git log` est la spécification d'un chemin. Si un répertoire ou un nom de fichier est spécifié, le journal est limité aux *commits* qui ont introduit des modifications aux fichiers concernés. C'est toujours la dernière option de la commande, souvent précédée de deux tirets ( `--` ) pour séparer les chemins des options précédentes.

Le tableau Options pour limiter la sortie de `git log` récapitule les options que nous venons de voir ainsi que quelques autres pour référence.

Table 3. Options pour limiter la sortie de `git log`

| Option | Description |
|---|---|
| `-(n)` | N'affiche que les n derniers *commits* |
| `--since`, `--after` | Limite l'affichage aux *commits* réalisés après la date spécifiée |
| `--until`, `--before` | Limite l'affichage aux *commits* réalisés avant la date spécifiée |
| `--author` | Ne montre que les *commits* dont le champ auteur correspond à la chaîne passée en argument |
| `--committer` | Ne montre que les *commits* dont le champ validateur correspond à la chaîne passée en argument |
| `--grep` | Ne montre que les *commits* dont le message de validation contient la chaîne de caractères |
| `-S` | Ne montre que les *commits* dont les ajouts ou retraits contient la chaîne de caractères |

Par exemple, si vous souhaitez visualiser quels *commits* modifiant les fichiers de test dans l'historique du code source de Git ont été validés par Junio C Hamano et n'étaient pas des fusions durant le mois d'octobre 2008, vous pouvez lancer ce qui suit :

```
$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn branch
```

À partir des 40 000 *commits* constituant l'historique des sources de Git, cette commande extrait les 6 qui correspondent aux critères.

# Annuler des actions

À tout moment, vous pouvez désirer annuler une de vos dernières actions. Dans cette section, nous allons passer en revue quelques outils de base permettant d'annuler des modifications. Il faut être très attentif car certaines de ces annulations sont définitives (elles ne peuvent pas être elles-mêmes annulées). C'est donc un des rares cas d'utilisation de Git où des erreurs de manipulation peuvent entraîner des pertes définitives de données.

Une des annulations les plus communes apparaît lorsqu'on valide une modification trop tôt en oubliant d'ajouter certains fichiers, ou si on se trompe dans le message de validation. Si vous souhaitez rectifier cette erreur, vous pouvez valider le complément de modification avec l'option `--amend` :

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

Cette commande prend en compte la zone d'index et l'utilise pour le *commit*. Si aucune modification n'a été réalisée depuis la dernière validation (par exemple en lançant cette commande immédiatement après la dernière validation), alors l'instantané sera identique et la seule modification à introduire sera le message de validation.

L'éditeur de message de validation démarre, mais il contient déjà le message de la validation précédente. Vous pouvez éditer ce message normalement, mais il écrasera le message de la validation précédente.

Par exemple, si vous validez une version puis réalisez que vous avez oublié d'indexer les modifications d'un fichier que vous vouliez ajouter à ce *commit*, vous pouvez faire quelque chose comme ceci :

```
$ git commit -m 'validation initiale'
$ git add fichier_oublie
$ git commit --amend
```

Vous n'aurez au final qu'un unique *commit* — la seconde validation remplace le résultat de la première.

## Désindexer un fichier déjà indexé

Les deux sections suivantes démontrent comment bricoler les modifications dans votre zone d'index et votre zone de travail. Un point sympathique est que la commande permettant de connaître l'état de ces deux zones vous rappelle aussi comment annuler les modifications. Par exemple, supposons que vous avez modifié deux fichiers et voulez les valider comme deux modifications indépendantes, mais que vous avez tapé accidentellement `git add *` et donc indexé les deux. Comment annuler l'indexation d'un des fichiers ? La commande `git status` vous le rappelle :

```
$ git add .
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    renommé :   README.md -> README
    modifié :   CONTRIBUTING.md
```

Juste sous le texte « Modifications qui seront validées », elle vous indique d'utiliser `git reset HEAD <fichier>...` pour désindexer un fichier. Utilisons donc ce conseil pour désindexer le fichier `CONTRIBUTING.md` :

```
$ git reset HEAD CONTRIBUTING.md
Modifications non indexées après reset :
M       CONTRIBUTING.md
$ git status
Sur la branche master
```

```
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        renommé :         README.md -> README

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md
```

La commande à taper peut sembler étrange mais elle fonctionne. Le fichier `CONTRIBUTING.md` est modifié mais de retour à l'état non indexé.

| | |
|---|---|
| | Bien que `git reset` *puisse* être une commande dangereuse conjuguée avec l'option `--hard`, dans le cas présent, le fichier dans la copie de travail n'a pas été touché. Appeler `git reset` sans option n'est pas dangereux - cela ne touche qu'à la zone d'index. |

Pour l'instant, cette invocation magique est la seule à connaître pour la commande `git reset`. Nous entrerons plus en détail sur ce que `reset` réalise et comment le maîtriser pour faire des choses intéressantes dans Reset démystifié

## Réinitialiser un fichier modifié

Que faire si vous réalisez que vous ne souhaitez pas conserver les modifications du fichier `CONTRIBUTING.md` ? Comment le réinitialiser facilement, le ramener à son état du dernier instantané (ou lors du clonage, ou dans l'état dans lequel vous l'avez obtenu dans votre copie de travail) ? Heureusement, `git status` vous indique comment procéder. Dans le résultat de la dernière commande, la zone de travail ressemble à ceci :

```
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         CONTRIBUTING.md
```

Ce qui vous indique de façon explicite comment annuler des modifications que vous avez faites. Faisons comme indiqué :

```
$ git checkout -- CONTRIBUTING.md
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        renommé :         README.md -> README
```

Vous pouvez constater que les modifications ont été annulées.

| | |
|---|---|
| | Vous devriez aussi vous apercevoir que c'est une commande dangereuse : toutes les modifications que vous auriez réalisées sur ce fichier ont disparu — vous venez tout juste de l'écraser avec un autre fichier. N'utilisez jamais cette commande à moins d'être vraiment sûr de ne pas vouloir de ces modifications. |

Si vous souhaitez seulement écarter momentanément cette modification, nous verrons comment mettre de côté et créer des branches dans le chapitre Les branches avec Git ; ce sont de meilleures façons de procéder.

Souvenez-vous, tout ce qui a été *validé* dans Git peut quasiment toujours être récupéré. Y compris des *commits* sur des branches qui ont été effacées ou des *commits* qui ont été écrasés par une validation avec l'option `--amend` (se référer au chapitre Récupération de données pour la récupération de données). Cependant, tout ce que vous perdez avant de l'avoir validé n'a aucune chance d'être récupérable via Git.

# Travailler avec des dépôts distants

Pour pouvoir collaborer sur un projet Git, il est nécessaire de savoir comment gérer les dépôts distants. Les dépôts distants sont des versions de votre projet qui sont hébergées sur Internet ou le réseau d'entreprise. Vous pouvez en avoir plusieurs, pour lesquels vous pouvez avoir des droits soit en lecture seule, soit en lecture/écriture. Collaborer avec d'autres personnes consiste à gérer ces dépôts distants, en poussant ou tirant des données depuis et vers ces dépôts quand vous souhaitez partager votre travail. Gérer des dépôts distants inclut savoir comment ajouter des dépôts distants, effacer des dépôts distants qui ne sont plus valides, gérer des branches distantes et les définir comme suivies ou non, et plus encore. Dans cette section, nous traiterons des commandes de gestion distante.

## Afficher les dépôts distants

Pour visualiser les serveurs distants que vous avez enregistrés, vous pouvez lancer la commande `git remote`. Elle liste les noms des différentes références distantes que vous avez spécifiées. Si vous avez cloné un dépôt, vous devriez au moins voir l'origine `origin` — c'est-à-dire le nom par défaut que Git donne au serveur à partir duquel vous avez cloné :

```
$ git clone https://github.com/schacon/ticgit
Clonage dans 'ticgit'...
remote: Counting objects: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Réception d'objets: 100% (1857/1857), 374.35 KiB | 243.00 KiB/s, fait.
Résolution des deltas: 100% (772/772), fait.
Vérification de la connectivité... fait.
$ cd ticgit
$ git remote
origin
```

Vous pouvez aussi spécifier `-v`, qui vous montre l'URL que Git a stockée pour chaque nom court :

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Si vous avez plus d'un dépôt distant, la commande précédente les liste tous. Par exemple, un dépôt avec plusieurs dépôts distants permettant de travailler avec quelques collaborateurs pourrait ressembler à ceci.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Notez que ces dépôts distants sont accessibles au moyen de différents protocoles ; nous traiterons des protocoles au chapitre Installation de Git sur un serveur.

## Ajouter des dépôts distants

J'ai expliqué et donné des exemples d'ajout de dépôts distants dans les chapitres précédents, mais voici spécifiquement comment faire. Pour ajouter un nouveau dépôt distant Git comme nom court auquel il est facile de faire référence, lancez `git remote add [nomcourt] [url]` :

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

Maintenant, vous pouvez utiliser le mot-clé `pb` sur la ligne de commande au lieu de l'URL complète. Par exemple, si vous voulez récupérer toute l'information que Paul a mais que vous ne souhaitez pas l'avoir encore dans votre branche, vous pouvez lancer `git fetch pb` :

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Dépaquetage des objets: 100% (43/43), fait.
Depuis https://github.com/paulboone/ticgit
 * [nouvelle branche] master     -> pb/master
 * [nouvelle branche] ticgit     -> pb/ticgit
```

La branche `master` de Paul est accessible localement en tant que `pb/master` — vous pouvez la fusionner dans une de vos propres branches, ou vous pouvez extraire une branche localement si vous souhaitez l'inspecter. Nous traiterons plus en détail de la nature des branches et de leur utilisation au chapitre Les branches avec Git.

## Récupérer et tirer depuis des dépôts distants

Comme vous venez tout juste de le voir, pour obtenir les données des dépôts distants, vous pouvez lancer :

```
$ git fetch [remote-name]
```

Cette commande s'adresse au dépôt distant et récupère toutes les données de ce projet que vous ne possédez pas déjà. Après cette action, vous possédez toutes les références à toutes les branches contenues dans ce dépôt, que vous pouvez fusionner ou inspecter à tout moment.

Si vous clonez un dépôt, le dépôt distant est automatiquement ajouté sous le nom « origin ». Donc, `git fetch origin` récupère tout ajout qui a été poussé vers ce dépôt depuis que vous l'avez cloné ou la dernière fois que vous avez récupéré les ajouts. Il faut noter que la commande `fetch` tire les données dans votre dépôt local mais sous sa propre branche — elle ne les fusionne pas automatiquement avec aucun de vos travaux ni ne modifie votre copie de travail. Vous devez volontairement fusionner ses modifications distantes dans votre travail lorsque vous le souhaitez.

Si vous avez créé une branche pour suivre l'évolution d'une branche distante (cf. la section suivante et le chapitre Les branches avec Git pour plus d'information), vous pouvez utiliser la commande `git pull` qui récupère et fusionne automatiquement une branche distante dans votre branche locale. Ce comportement peut correspondre à une méthode de travail plus confortable, sachant que par défaut la commande `git clone` paramètre votre branche locale pour qu'elle suive la branche `master` du dépôt que vous avez cloné (en supposant que le dépôt distant ait une branche `master`). Lancer `git pull` récupère généralement les données depuis le serveur qui a été initialement cloné et essaie de les fusionner dans votre branche de travail actuel.

## Pousser son travail sur un dépôt distant

Lorsque votre dépôt vous semble prêt à être partagé, il faut le pousser en amont. La commande pour le faire est simple : `git push [nom-distant] [nom-de-branche]`. Si vous souhaitez pousser votre branche `master` vers le serveur `origin` (pour rappel, cloner un dépôt définit automatiquement ces noms pour vous), alors vous pouvez lancer ceci pour pousser votre travail vers le serveur amont :

```
$ git push origin master
```

Cette commande ne fonctionne que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Si vous et quelqu'un d'autre clonez un dépôt au même moment et que cette autre personne pousse ses modifications et qu'après vous tentez de pousser les vôtres, votre poussée sera rejetée à juste titre. Vous devrez tout d'abord tirer les modifications de l'autre personne et les fusionner avec les vôtres avant de pouvoir pousser. Référez-vous au chapitre Les branches avec Git pour de plus amples informations sur les techniques pour pousser vers un serveur distant.

## Inspecter un dépôt distant

Si vous souhaitez visualiser plus d'informations à propos d'un dépôt distant particulier, vous pouvez utiliser la commande `git remote show [nom-distant]`. Si vous lancez cette commande avec un nom court particulier, tel que `origin`, vous obtenez quelque chose comme :

```
$ git remote show origin
* distante origin
  URL de rapatriement : https://github.com/schacon/ticgit
  URL push : https://github.com/schacon/ticgit
  Branche HEAD : master
  Branches distantes :
    master suivi
    ticgit suivi
  Branche locale configurée pour 'git pull' :
    master fusionne avec la distante master
  Référence locale configurée pour 'git push' :
    master pousse vers master (à jour)
```

Cela donne la liste des URL pour le dépôt distant ainsi que la liste des branches distantes suivies. Cette commande vous informe que si vous êtes sur la branche `master` et si vous lancez `git pull`, il va automatiquement fusionner la branche `master` du dépôt distant après avoir récupéré toutes les références sur le serveur distant. Cela donne aussi la liste des autres références qu'il aura tirées.

Le résultat ci-dessus est un exemple simple mais réaliste de dépôt distant. Lors d'une utilisation plus intense de Git, la commande `git remote show` fournira beaucoup d'information :

```
$ git remote show origin
* distante origin
  URL: https://github.com/my-org/complex-project
  URL de rapatriement : https://github.com/my-org/complex-project
  URL push : https://github.com/my-org/complex-project
  Branche HEAD : master
  Branches distantes :
    master                           suivi
    dev-branch                       suivi
    markdown-strip                   suivi
    issue-43                         nouveau (le prochain rapatriement (fetch) stockera dans remotes/origin)
    issue-45                         nouveau (le prochain rapatriement (fetch) stockera dans remotes/origin)
    refs/remotes/origin/issue-11     dépassé (utilisez 'git remote prune' pour supprimer)
  Branches locales configurées pour 'git pull' :
    dev-branch fusionne avec la distante dev-branch
    master     fusionne avec la distante master
  Références locales configurées pour 'git push' :
    dev-branch                     pousse vers dev-branch       (à jour)
    markdown-strip                 pousse vers markdown-strip   (à jour)
    master                         pousse vers master           (à jour)
```

Cette commande affiche les branches poussées automatiquement lorsqu'on lance `git push` dessus. Elle montre aussi les branches distantes qui n'ont pas encore été rapatriées, les branches distantes présentes localement mais effacées sur le serveur, et toutes les branches qui seront fusionnées quand on lancera `git pull`.

### Retirer et renommer des dépôts distants

Si vous souhaitez renommer une référence, vous pouvez lancer `git remote rename` pour modifier le nom court d'un dépôt distant. Par exemple, si vous souhaitez renommer `pb` en `paul`, vous pouvez le faire avec `git remote rename` :

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Il faut mentionner que ceci modifie aussi les noms de branches distantes. Celle qui était référencée sous `pb/master` l'est maintenant sous `paul/master`.

Si vous souhaitez retirer un dépôt distant pour certaines raisons — vous avez changé de serveur ou vous n'utilisez plus ce serveur particulier, ou peut-être un contributeur a cessé de contribuer — vous pouvez utiliser `git remote rm` :

```
$ git remote rm paul
$ git remote
origin
```

## Étiquetage

À l'instar de la plupart des VCS, Git donne la possibilité d'étiqueter un certain état dans l'historique comme important. Généralement, les gens utilisent cette fonctionnalité pour marquer les états de publication ( `v1.0` et ainsi de suite). Dans cette section, nous apprendrons comment lister les différentes étiquettes (*tag* en anglais), comment créer de nouvelles étiquettes et les différents types d'étiquettes.

### Lister vos étiquettes

Lister les étiquettes existantes dans Git est très simple. Tapez juste `git tag` :

```
$ git tag
v0.1
v1.3
```

Cette commande liste les étiquettes dans l'ordre alphabétique. L'ordre dans lequel elles apparaissent n'a aucun rapport avec l'historique.

Vous pouvez aussi rechercher les étiquettes correspondant à un motif particulier. Par exemple, le dépôt des sources de Git contient plus de 500 étiquettes. Si vous souhaitez ne visualiser que les séries 1.8.5, vous pouvez lancer ceci :

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

## Créer des étiquettes

Git utilise deux types principaux d'étiquettes : légères et annotées.

Une étiquette légère ressemble beaucoup à une branche qui ne change pas, c'est juste un pointeur sur un *commit* spécifique.

Les étiquettes annotées, par contre, sont stockées en tant qu'objets à part entière dans la base de données de Git. Elles ont une somme de contrôle, contiennent le nom et l'adresse e-mail du créateur, la date, un message d'étiquetage et peuvent être signées et vérifiées avec GNU Privacy Guard (GPG). Il est généralement recommandé de créer des étiquettes annotées pour générer toute cette information mais si l'étiquette doit rester temporaire ou l'information supplémentaire n'est pas désirée, les étiquettes légères peuvent suffire.

## Les étiquettes annotées

Créer des étiquettes annotées est simple avec Git. Le plus simple est de spécifier l'option `-a` à la commande `tag` :

```
$ git tag -a v1.4 -m 'ma version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

L'option `-m` permet de spécifier le message d'étiquetage qui sera stocké avec l'étiquette. Si vous ne spécifiez pas de message en ligne pour une étiquette annotée, Git lance votre éditeur pour pouvoir le saisir.

Vous pouvez visualiser les données de l'étiquette à côté du *commit* qui a été marqué en utilisant la commande `git show` :

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

ma version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Cette commande affiche le nom du créateur, la date de création de l'étiquette et le message d'annotation avant de montrer effectivement l'information de validation.

## Les étiquettes légères

Une autre manière d'étiqueter les *commits* est d'utiliser les étiquettes légères. Celles-ci se réduisent à stocker la somme de contrôle d'un *commit* dans un fichier, aucune autre information n'est conservée. Pour créer une étiquette légère, il suffit de n'utiliser aucune des options `-a` , `-s` ou `-m` :

```
$ git tag v1.4-lg
$ git tag
v0.1
v1.3
v1.4
v1.4-lg
v1.5
```

Cette fois-ci, en lançant `git show` sur l'étiquette, on ne voit plus aucune information complémentaire. La commande ne montre que l'information de validation :

```
$ git show v1.4-lg
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

## Étiqueter après coup

Vous pouvez aussi étiqueter des *commits* plus anciens. Supposons que l'historique des *commits* ressemble à ceci :

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Fusion branche 'experimental'
a6b4c97498bd301d84096da251c98a07c7723e65 Début de l'écriture support
0d52aaab4479697da7686c15f77a3d64d9165190 Un truc de plus
6d52a271eda8725415634dd79daabbc4d9b6008e Fusion branche 'experimental'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc ajout d'une fonction de validatn
4682c3261057305bdd616e23b64b0857d832627b ajout fichier afaire
166ae0c4d3f420721acbb115cc33848dfcc2121a début de l'ecriture support
9fceb02d0ae598e95dc970b74767f19372d61af8 mise à jour rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc validation afaire
8a5cbc430f1a9c3d00faaeffd07798508422908a mise à jour lisezmoi
```

Maintenant, supposons que vous avez oublié d'étiqueter le projet à la version `v1.2` qui correspondait au *commit* « mise à jour rakefile ». Vous pouvez toujours le faire après l'évènement. Pour étiqueter ce *commit*, vous spécifiez la somme de contrôle du *commit* (ou une partie) en fin de commande :

```
$ git tag -a v1.2 9fceb02
```

Le *commit* a été étiqueté :

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lg
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

## Partager les étiquettes

Par défaut, la commande `git push` ne transfère pas les étiquettes vers les serveurs distants. Il faut explicitement pousser les étiquettes après les avoir créées localement. Ce processus s'apparente à pousser des branches distantes — vous pouvez lancer `git push origin [nom-du-tag]`.

```
$ git push origin v1.5
Décompte des objets: 14, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (12/12), fait.
```

```
Écriture des objets: 100% (14/14), 2.05KiB | 0 bytes/s, fait.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

Si vous avez de nombreuses étiquettes que vous souhaitez pousser en une fois, vous pouvez aussi utiliser l'option `--tags` avec la commande `git push`. Ceci transférera toutes les nouvelles étiquettes vers le serveur distant.

```
$ git push origin --tags
Décompte des objets: 1, fait.
Écriture des objets: 100% (1/1), 160 bytes | 0 bytes/s, fait.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lg -> v1.4-lg
```

À présent, lorsqu'une autre personne clone ou tire depuis votre dépôt, elle obtient aussi les étiquettes.

### Extraire une étiquette

Il n'est pas vraiment possible d'extraire une étiquette avec Git, puisque les étiquettes ne peuvent pas être modifiées. Si vous souhaitez ressortir dans votre copie de travail une version de votre dépôt correspondant à une étiquette spécifique, le plus simple consiste à créer une branche à partir de cette étiquette :

```
$ git checkout -b version2 v2.0.0
Extraction des fichiers: 100% (602/602), fait.
Basculement sur la nouvelle branche 'version2'
```

Bien sûr, toute validation modifiera la branche `version2` par rapport à l'étiquette `v2.0.0` puisqu'elle avancera avec les nouvelles modifications. Soyez donc prudent sur l'identification de cette branche.

## Les alias Git

Avant de clore ce chapitre sur les bases de Git, il reste une astuce qui peut rendre votre apprentissage de Git plus simple, facile ou familier : les alias. Nous n'y ferons pas référence ni ne les considèrerons utilisés dans la suite du livre, mais c'est un moyen de facilité qui mérite d'être connu.

Git ne complète pas votre commande si vous ne la tapez que partiellement. Si vous ne voulez pas avoir à taper l'intégralité du texte de chaque commande, vous pouvez facilement définir un alias pour chaque commande en utilisant `git config`. Voici quelques exemples qui pourraient vous intéresser :

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Ceci signifie que, par exemple, au lieu de taper `git commit`, vous n'avez plus qu'à taper `git ci`. Au fur et à mesure de votre utilisation de Git, vous utiliserez probablement d'autres commandes plus fréquemment. Dans ce cas, n'hésitez pas à créer de nouveaux alias.

Cette technique peut aussi être utile pour créer des commandes qui vous manquent. Par exemple, pour corriger le problème d'ergonomie que vous avez rencontré lors de la désindexation d'un fichier, vous pourriez créer un alias pour désindexer :

```
$ git config --global alias.unstage 'reset HEAD --'
```

Cela rend les deux commandes suivantes équivalentes :

```
$ git unstage fileA
$ git reset HEAD fileA
```

Cela rend les choses plus claires. Il est aussi commun d'ajouter un alias `last`, de la manière suivante :

```
$ git config --global alias.last 'log -1 HEAD'
```

Ainsi, vous pouvez visualiser plus facilement le dernier *commit* :

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

Pour explication, Git remplace simplement la nouvelle commande par tout ce que vous lui aurez demandé d'aliaser. Si par contre vous souhaitez lancer une commande externe plutôt qu'une sous-commande Git, vous pouvez commencer votre commande par un caractère `!`. C'est utile si vous écrivez vos propres outils pour travailler dans un dépôt Git. On peut par exemple aliaser `git visual` pour lancer `gitk` :

```
$ git config --global alias.visual "!gitk"
```

# Résumé

À présent, vous pouvez réaliser toutes les opérations locales de base de Git — créer et cloner un dépôt, faire des modifications, les indexer et les valider, visualiser l'historique de ces modifications. Au prochain chapitre, nous traiterons de la fonctionnalité unique de Git : son modèle de branches.

# Les branches avec Git


- Les branches en bref
  - Créer une nouvelle branche
  - Basculer entre les branches
- Branches et fusions : les bases
  - Branches
  - Fusions (*Merges*)
  - Conflits de fusions (*Merge conflicts*)
- Gestion des branches
- Travailler avec les branches
  - Branches au long cours
  - Les branches thématiques
- Branches de suivi à distance
  - Pousser les branches
  - Suivre les branches
  - Tirer une branche (*Pulling*)
  - Suppression de branches distantes
- Rebaser (*Rebasing*)
  - Les bases
  - Rebases plus intéressants
  - Les dangers du rebasage
  - Rebaser quand vous rebasez
  - Rebaser ou Fusionner
- Résumé


Presque tous les VCS proposent une certaine forme de gestion de branches. Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne. Pour de nombreux VCS, il s'agit d'un processus coûteux qui nécessite souvent la création d'une nouvelle copie du répertoire de travail, ce qui peut prendre longtemps dans le cas de gros projets.

Certaines personnes considèrent le modèle de gestion de branches de Git comme ce qu'il a de plus remarquable et il offre sûrement à Git une place à part au sein de la communauté des VCS. En quoi est-il si spécial ? La manière dont Git gère les branches est incroyablement légère et permet de réaliser les opérations sur les branches de manière quasi instantanée et, généralement, de basculer entre les branches aussi rapidement. À la différence de nombreux autres VCS, Git encourage des méthodes qui privilégient la création et la fusion fréquentes de branches, jusqu'à plusieurs fois par jour. Bien comprendre et maîtriser cette fonctionnalité vous permettra de faire de Git un outil puissant et unique et peut totalement changer votre manière de développer.

## Les branches en bref

Pour réellement comprendre la manière dont Git gère les branches, nous devons revenir en arrière et examiner de plus près comment Git stocke ses données. Si vous vous souvenez bien du chapitre Démarrage rapide, Git ne stocke pas ses données comme une série de modifications ou de différences successives mais plutôt comme une série d'instantanés (appelés *snapshots*).

Lorsque vous faites un commit, Git stocke un objet *commit* qui contient un pointeur vers l'instantané (*snapshot*) du contenu que vous avez indexé. Cet objet contient également les noms et prénoms de l'auteur, le message que vous avez renseigné ainsi que des pointeurs vers le ou les *commits* qui précèdent directement ce *commit* : aucun parent pour le *commit* initial, un parent pour un *commit* normal et de multiples parents pour un *commit* qui résulte de la fusion d'une ou plusieurs branches.

Pour visualiser ce concept, supposons que vous avez un répertoire contenant trois fichiers que vous indexez puis validez. L'indexation des fichiers calcule une empreinte (*checksum*) pour chacun (via la fonction de hachage SHA-1 mentionnée au chapitre Démarrage rapide), stocke cette version du fichier dans le dépôt Git (Git les nomme *blobs*) et ajoute cette empreinte à la zone d'index (*staging area*) :

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Lorsque vous créez le *commit* en lançant la commande `git commit`, Git calcule l'empreinte de chaque sous-répertoire (ici, seulement pour le répertoire racine) et stocke ces objets de type arbre dans le dépôt Git. Git crée alors un objet *commit* qui contient les méta-données et un pointeur vers l'arbre de la racine du projet de manière à pouvoir recréer l'instantané à tout moment.

Votre dépôt Git contient à présent cinq objets : un *blob* pour le contenu de chacun de vos trois fichiers, un arbre (*tree*) qui liste le contenu du répertoire et spécifie quels noms de fichiers sont attachés à quels *blobs* et enfin un objet *commit* portant le pointeur vers l'arbre de la racine ainsi que toutes les méta-données attachées au *commit*.

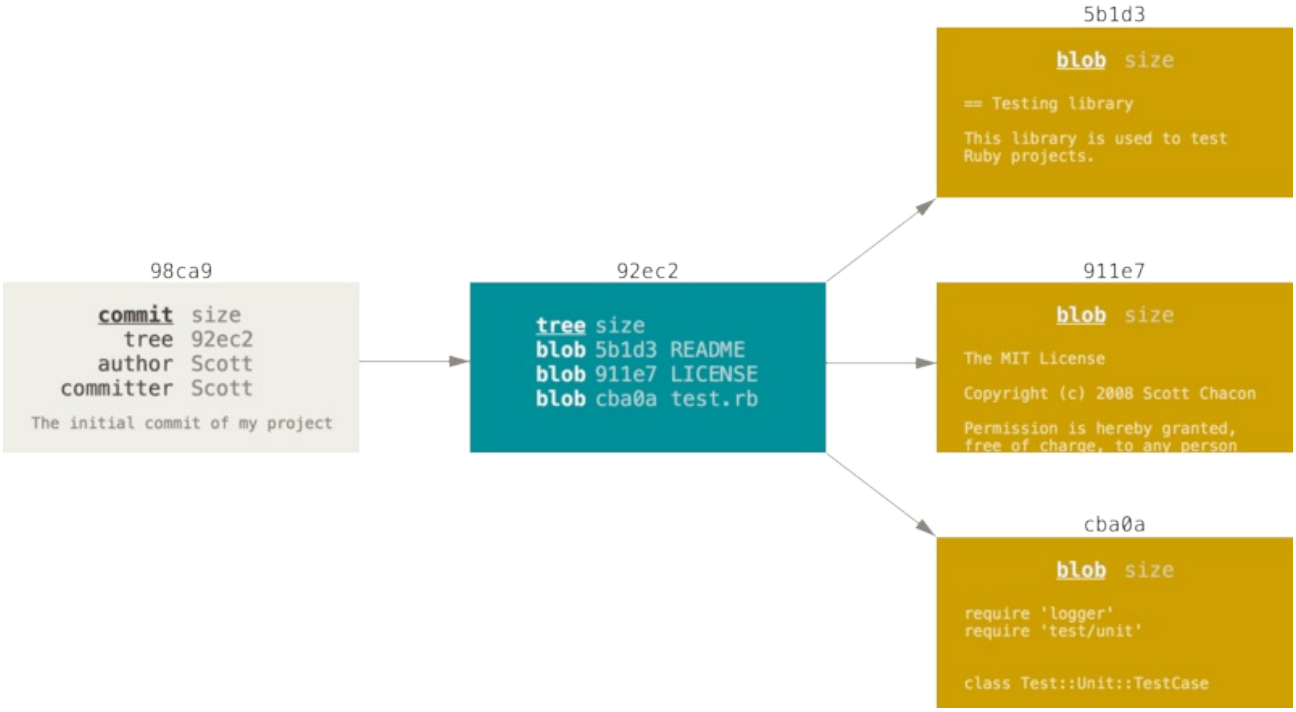


*Figure 10 : Un commit et son arbre.*

Figure 9. Un commit et son arbre

Si vous faites des modifications et validez à nouveau, le prochain *commit* stocke un pointeur vers le *commit* le précédant immédiatement.

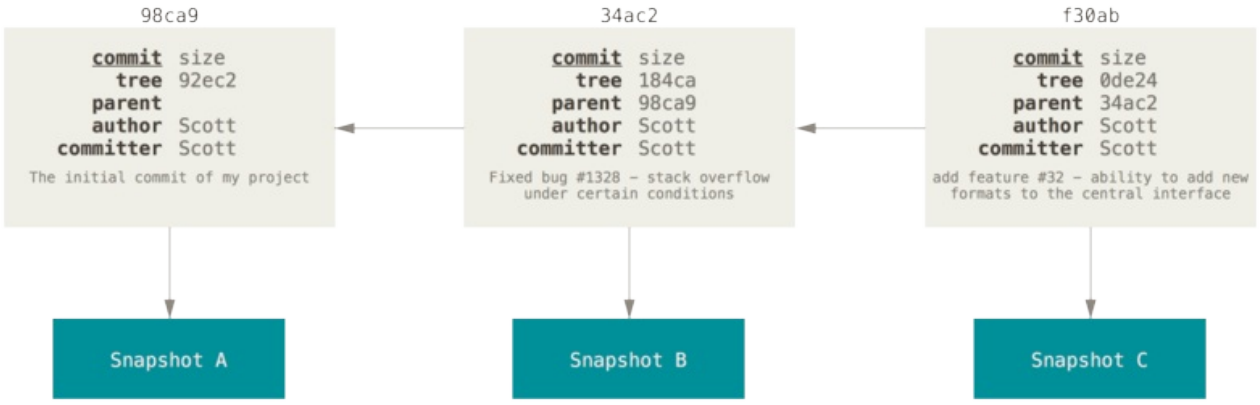


*Figure 11 : Commits et leurs parents.*

Figure 10. Commits et leurs parents

Une branche dans Git est simplement un pointeur léger et déplaçable vers un de ces *commits*. La branche par défaut dans Git s'appelle `master`. Au fur et à mesure des validations, la branche `master` pointe vers le dernier des *commits* réalisés. À chaque validation, le pointeur de la branche `master` avance automatiquement.

| | |
|---|---|
| | La branche ``master`` n'est pas une branche spéciale. Elle est identique à toutes les autres branches. La seule raison pour laquelle chaque dépôt en a une est que la commande `git init` la crée par défaut et que la plupart des gens ne s'embêtent pas à la changer. |

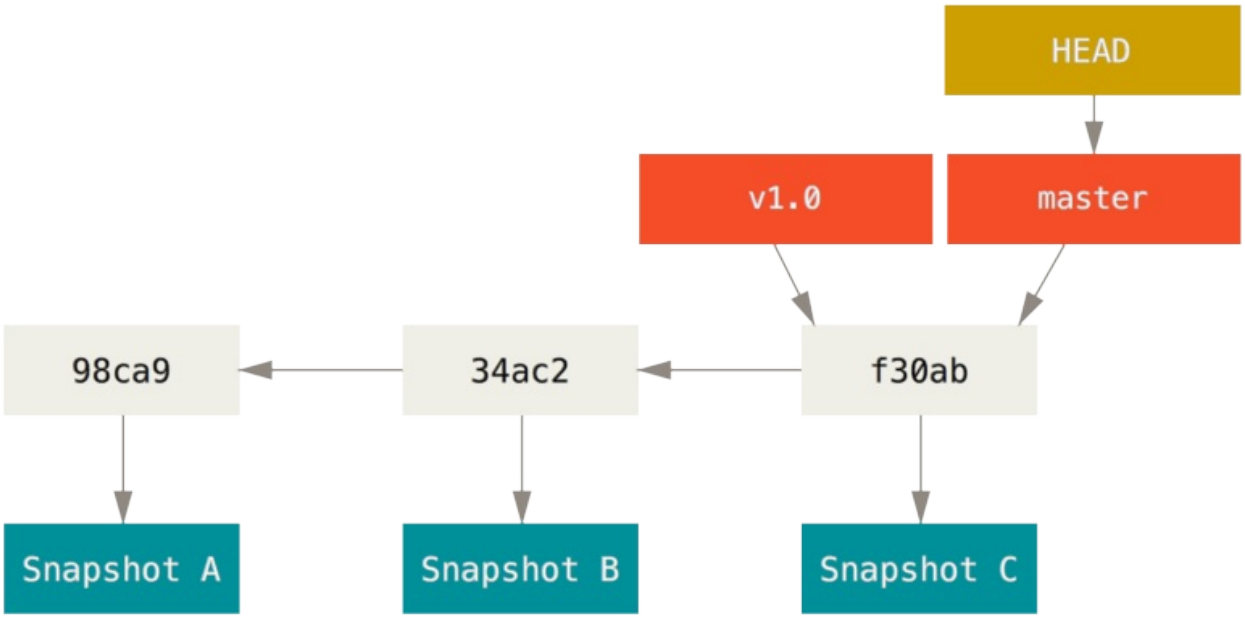


*Figure 12 : Une branche et l'historique de ses _commits_.*

Figure 11. Une branche et l'historique de ses *commits*

## Créer une nouvelle branche

Que se passe-t-il si vous créez une nouvelle branche ? Eh bien, cela crée un nouveau pointeur pour vous. Supposons que vous créez une nouvelle branche nommée `test`. Vous utilisez pour cela la commande `git branch` :

```
$ git branch testing
```

Cela crée un nouveau pointeur vers le *commit* courant.

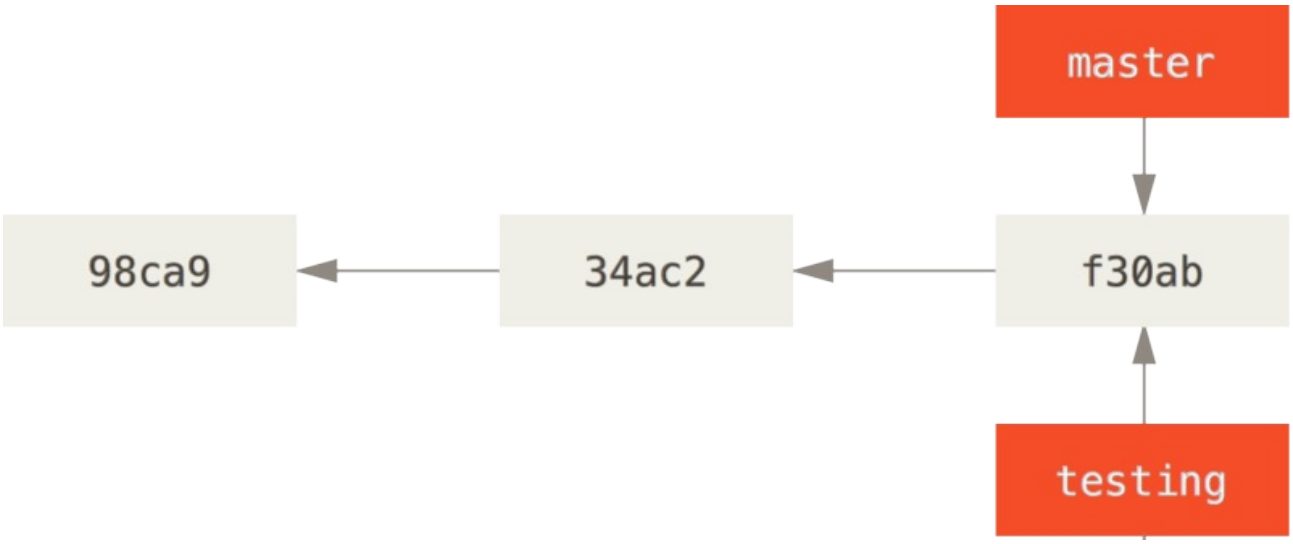


*Figure 13 : Deux branches pointant vers la même série de _commits_.*

Figure 12. Deux branches pointant vers la même série de *commits*

Comment Git connaît-il alors la branche sur laquelle vous vous trouvez ? Il conserve à cet effet un pointeur spécial appelé `HEAD`. Vous remarquez que sous cette appellation se cache un concept très différent de celui utilisé dans les autres VCS tels que Subversion ou CVS. Dans Git, il s'agit simplement d'un pointeur sur la branche locale où vous vous trouvez. Dans ce cas, vous vous trouvez toujours sur `master`. En effet, la commande `git branch` n'a fait que créer une nouvelle branche — elle n'a pas fait basculer la copie de travail vers cette branche.

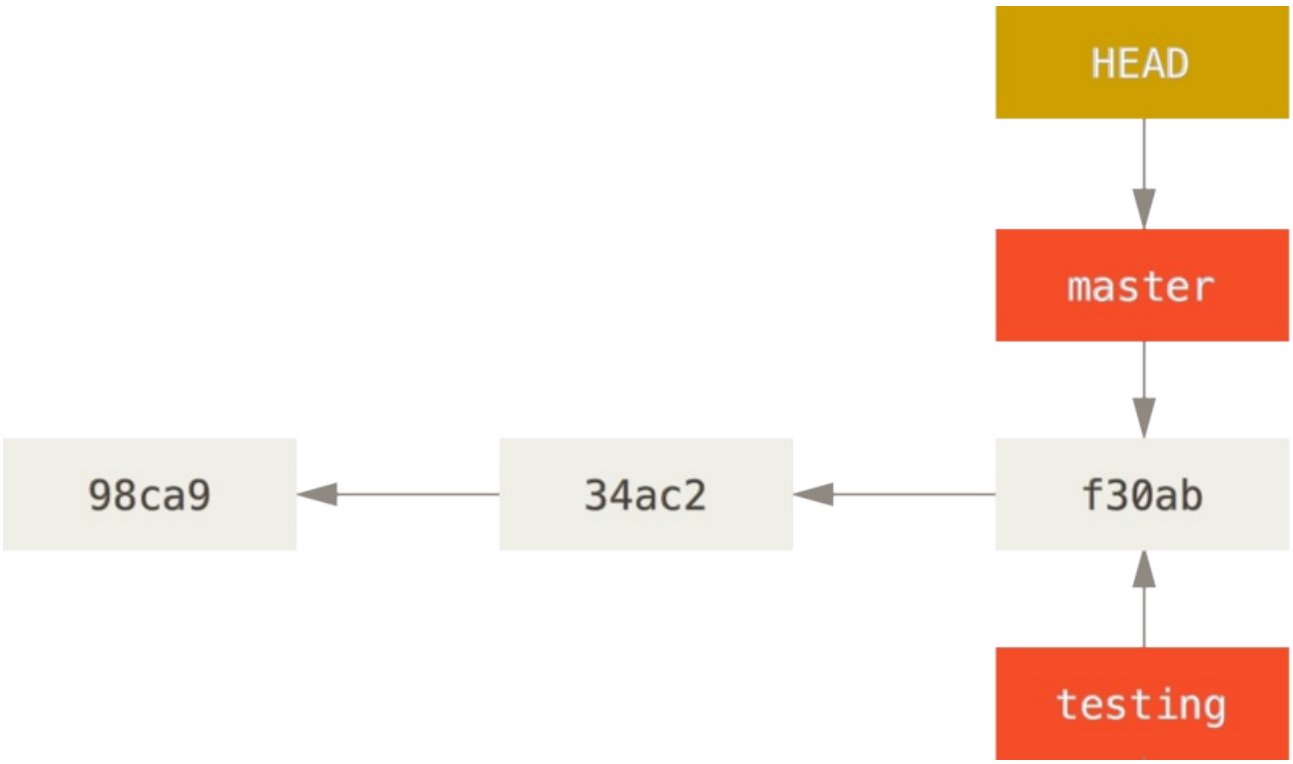


*Figure 14 : HEAD pointant vers une branche.*

Figure 13. HEAD pointant vers une branche

Vous pouvez vérifier cela facilement grâce à la commande `git log` qui vous montre vers quoi les branches pointent. Il s'agit de l'option `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD, master, test) add feature #32 - ability to add new
34ac2 fixed bug #ch1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Vous pouvez voir les branches ``master`` et ``test`` qui se situent au niveau du *commit* `f30ab`.

## Basculer entre les branches

Pour basculer sur une branche existante, il suffit de lancer la commande `git checkout`. Basculons sur la nouvelle branche `testing` :

```
$ git checkout testing
```

Cela déplace `HEAD` pour le faire pointer vers la branche `testing`.

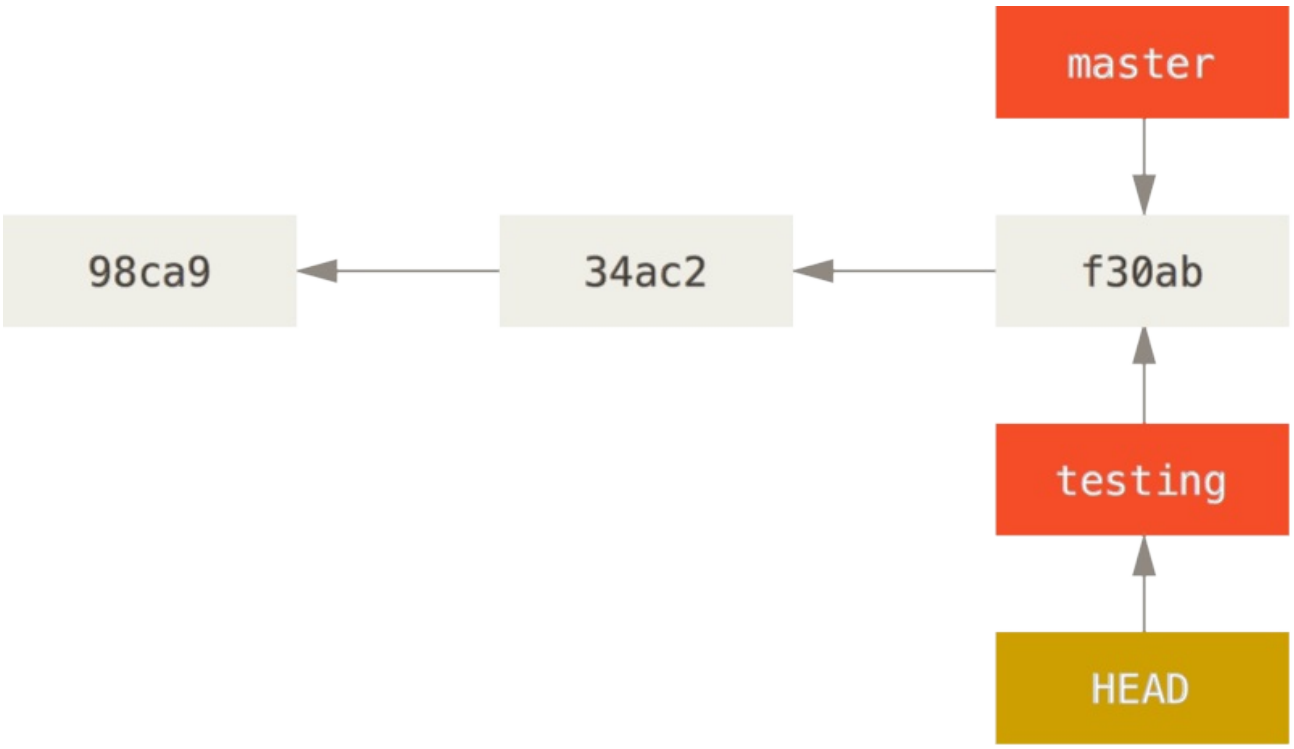


*Figure 15 : HEAD pointe vers la branche courante.*

Figure 14. HEAD pointe vers la branche courante

Qu'est-ce que cela signifie ? Et bien, faisons une autre validation :

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

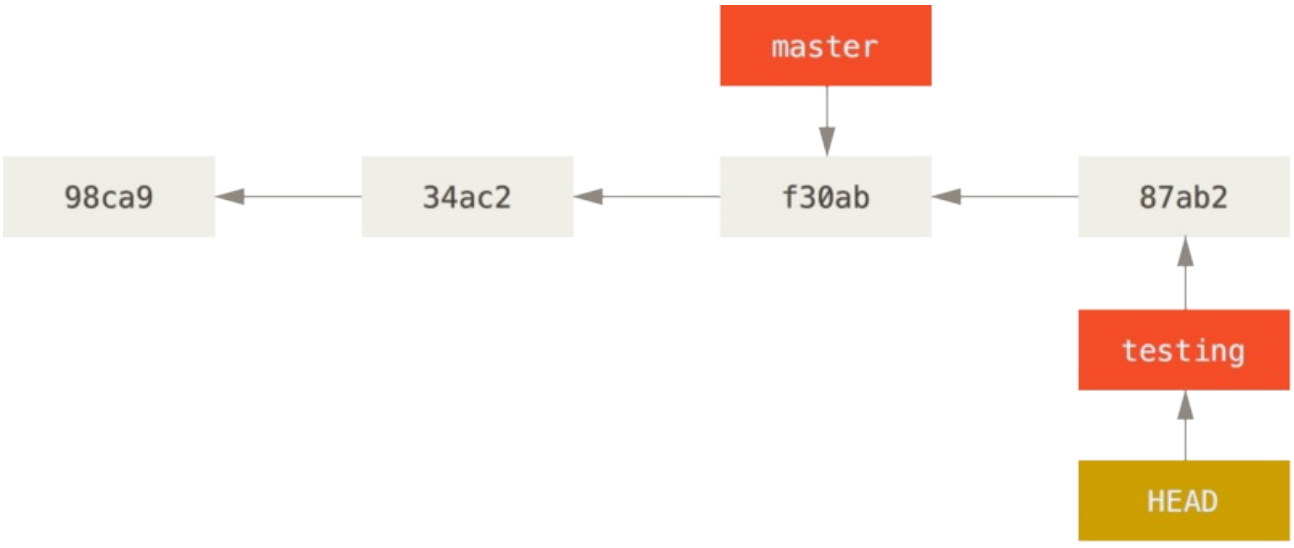


*Figure 16 : La branche HEAD avance à chaque _commit_.*

Figure 15. La branche HEAD avance à chaque *commit*

C'est intéressant parce qu'à présent, votre branche `test` a avancé tandis que la branche `master` pointe toujours sur le *commit* sur lequel vous étiez lorsque vous avez lancé la commande `git checkout` pour changer de branche. Retournons sur la branche `master` :

```
$ git checkout master
```

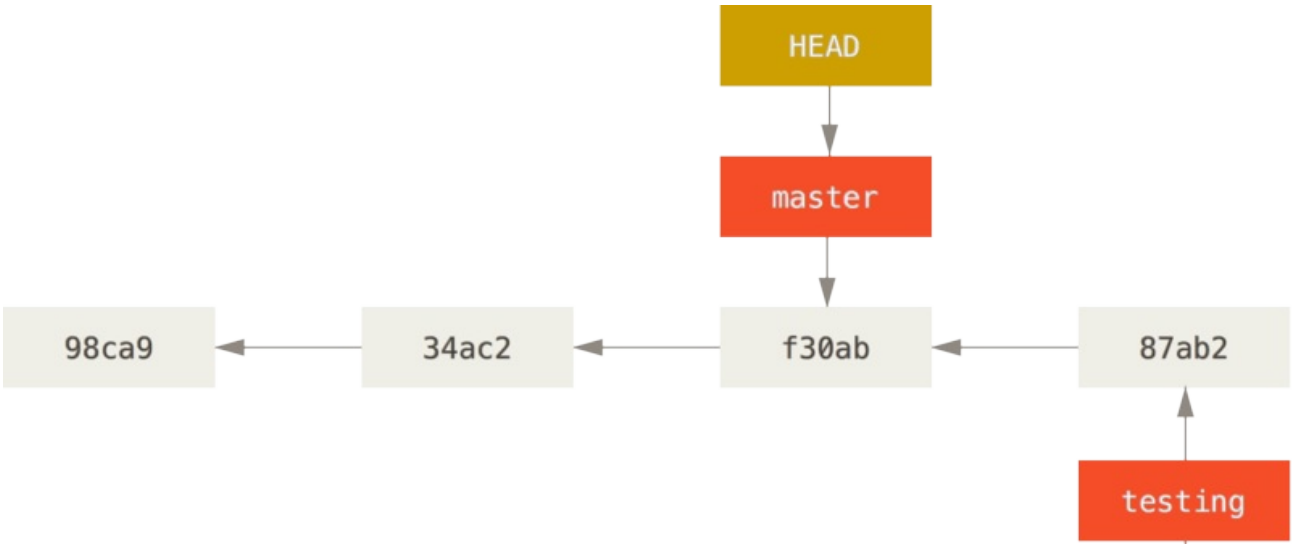


*Figure 17 : HEAD est déplacé lors d'un _checkout_.*

Figure 16. HEAD est déplacé lors d'un *checkout*

Cette commande a réalisé deux actions. Elle a remis le pointeur `HEAD` sur la branche `master` et elle a replacé les fichiers de votre répertoire de travail dans l'état du *snapshot* pointé par `master`. Cela signifie aussi que les modifications que vous réalisez à partir de ce point divergeront de l'ancienne version du projet. Cette commande annule les modifications réalisées dans la branche `test` pour vous permettre de repartir dans une autre direction.

| | |
|---|---|
| | Changer de branche modifie les fichiers dans votre répertoire de travail<br>Il est important de noter que lorsque vous changez de branche avec Git, les fichiers de votre répertoire de travail sont modifiés. Si vous basculez vers une branche plus ancienne, votre répertoire de travail sera remis dans l'état dans lequel il était lors du dernier *commit* sur cette branche. Si git n'est pas en mesure d'effectuer cette action proprement, il ne vous laissera pas changer de branche. |

Réalisons quelques autres modifications et validons à nouveau :

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Maintenant, l'historique du projet a divergé (voir Divergence d'historique). Vous avez créé une branche et basculé dessus, y avez réalisé des modifications, puis vous avez rebasculé sur la branche principale et réalisé d'autres modifications. Ces deux modifications sont isolées dans des branches séparées : vous pouvez basculer d'une branche à l'autre et les fusionner quand vous êtes prêt. Et vous avez fait tout ceci avec de simples commandes : `branch`, `checkout` et `commit`.

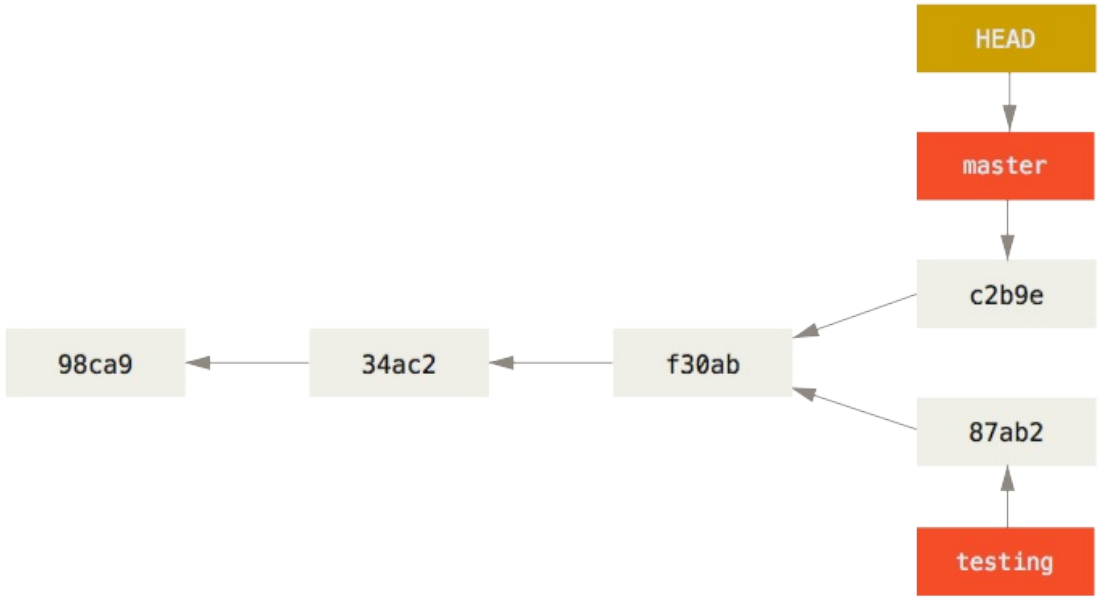


*Figure 18 : Divergence d'historique.*

Figure 17. Divergence d'historique

Vous pouvez également voir ceci grâce à la commande `git log`. La commande `git log --oneline --decorate --graph --all` va afficher l'historique de vos *commits*, affichant les endroits où sont positionnés vos pointeurs de branche ainsi que la manière dont votre historique a divergé.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (test) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #ch1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Parce qu'une branche Git n'est en fait qu'un simple fichier contenant les 40 caractères de l'empreinte SHA-1 du *commit* sur lequel elle pointe, les branches ne coûtent quasiment rien à créer et à détruire. Créer une branche est aussi simple et rapide qu'écrire 41 caractères dans un fichier (40 caractères plus un retour chariot).

C'est une différence de taille avec la manière dont la plupart des VCS gèrent les branches, qui implique de copier tous les fichiers du projet dans un second répertoire. Cela peut durer plusieurs secondes ou même quelques minutes selon la taille du projet, alors que pour Git, le processus est toujours instantané. De plus, comme nous enregistrons les parents quand nous validons les modifications, la détermination de l'ancêtre commun approprié pour la fusion est réalisée automatiquement pour nous et est généralement une opération très facile. Ces fonctionnalités encouragent naturellement les développeurs à créer et utiliser souvent des branches.

Voyons pourquoi vous devriez en faire autant.

# Branches et fusions : les bases

Prenons un exemple simple faisant intervenir des branches et des fusions (*merges*) que vous pourriez trouver dans le monde réel. Vous effectuez les tâches suivantes :

1. vous travaillez sur un site web ;
2. vous créez une branche pour un nouvel article en cours ;
3. vous commencez à travailler sur cette branche.

À cette étape, vous recevez un appel pour vous dire qu'un problème critique a été découvert et qu'il faut le régler au plus tôt. Vous faites donc ce qui suit :

1. vous basculez sur la branche de production ;
2. vous créez une branche pour y ajouter le correctif ;
3. après l'avoir testé, vous fusionnez la branche du correctif et poussez le résultat en production ;
4. vous rebasculez sur la branche initiale et continuez votre travail.

## Branches

Commençons par supposer que vous travaillez sur votre projet et avez déjà quelques *commits*.

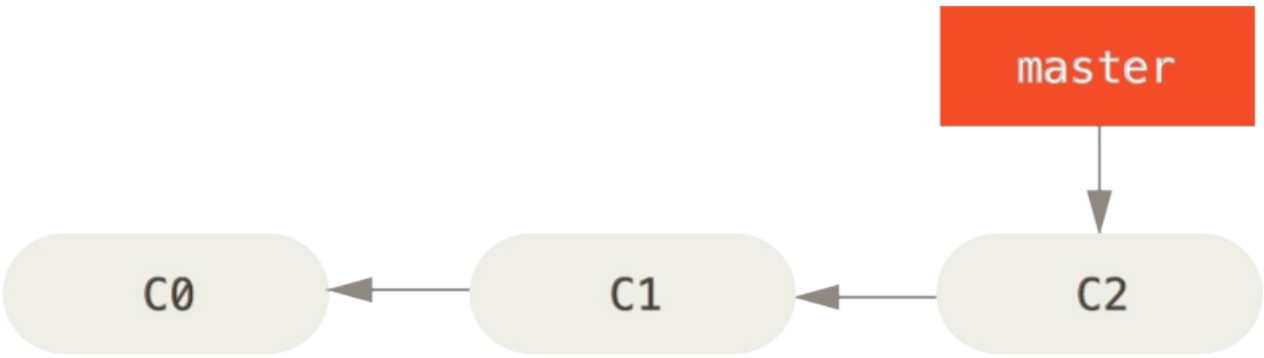


*Figure 19 : Historique de _commits_ simple.*

Figure 18. Historique de *commits* simple

Vous avez décidé de travailler sur le problème numéroté #53 dans l'outil de gestion des tâches que votre entreprise utilise, quel qu'il soit. Pour créer une branche et y basculer tout de suite, vous pouvez lancer la commande `git checkout` avec l'option `-b` :

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Cette commande est un raccourci pour :

```
$ git branch iss53
$ git checkout iss53
```

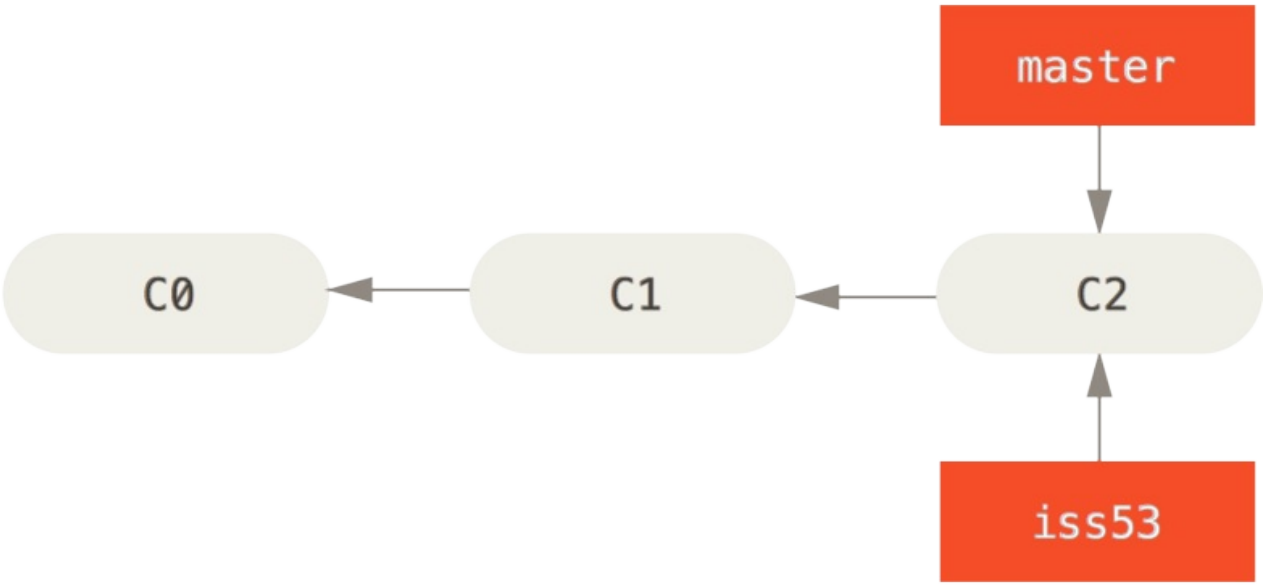


*Figure 20 : Création d'un nouveau pointeur de branche.*

Figure 19. Création d'un nouveau pointeur de branche

Vous travaillez sur votre site web et validez vos modifications. Ce faisant, la branche `iss53` avance parce que vous l'avez extraite (c'est-à-dire que votre pointeur `HEAD` pointe dessus) :

```
$ vim index.html
$ git commit -a -m "ajout d'un pied de page [problème 53]"
```

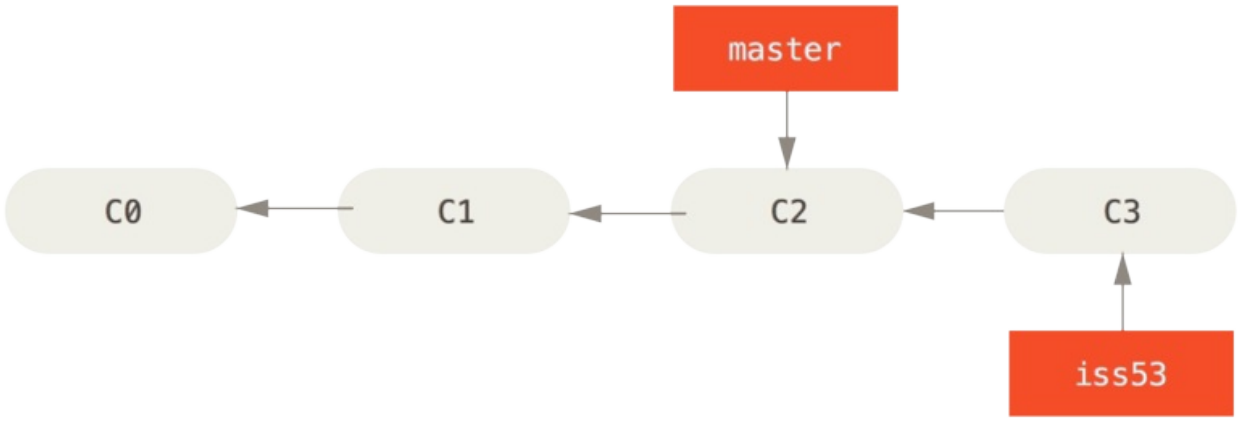


*Figure 21 : La branche iss53 a avancé avec votre travail.*

Figure 20. La branche iss53 a avancé avec votre travail

À ce moment-là, vous recevez un appel qui vous apprend qu'il y a un problème sur le site web qu'il faut résoudre immédiatement. Avec Git, vous n'avez pas à déployer en même temps votre correctif et les modifications déjà validées pour `iss53` et vous n'avez pas non plus à vous fatiguer à annuler ces modifications avant de pouvoir appliquer votre correctif sur ce qu'il y a en production. Tout ce que vous avez à faire, c'est simplement de rebasculer sur la branche `master` .

Cependant, avant de le faire, notez que si votre copie de travail ou votre zone d'index contiennent des modifications non validées qui sont en conflit avec la branche que vous extrayez, Git ne vous laissera pas changer de branche. Le mieux est d'avoir votre copie de travail propre au moment de changer de branche. Il y a des moyens de contourner ceci (précisément par le remisage et l'amendement de *commit*) dont nous parlerons plus loin, au chapitre Remisage et nettoyage. Pour l'instant, nous supposons que vous avez validé tous vos changements et que vous pouvez donc rebasculer vers votre branche `master` :

```
$ git checkout master
Switched to branch 'master'
```

À cet instant, votre répertoire de copie de travail est exactement dans l'état dans lequel vous l'aviez laissé avant de commencer à travailler sur le problème #53 et vous pouvez vous consacrer à votre correctif. C'est un point important à garder en mémoire : quand vous changez de branche, Git réinitialise votre répertoire de travail pour qu'il soit le même que la dernière fois que vous avez effectué un *commit* sur cette branche. Il ajoute, retire et modifie automatiquement les fichiers de manière à s'assurer que votre copie de travail soit identique à ce qu'elle était lors de votre dernier *commit* sur cette branche.

Vous avez ensuite un correctif à faire. Pour ce faire, créons une branche `correctif` sur laquelle travailler jusqu'à résolution du problème :

```
$ git checkout -b correctif
Switched to a new branch 'correctif'
$ vim index.html
$ git commit -a -m "correction de l'adresse email incorrecte"
[correctif 1fb7853] "correction de l'adresse email incorrecte"
 1 file changed, 2 insertions(+)
```

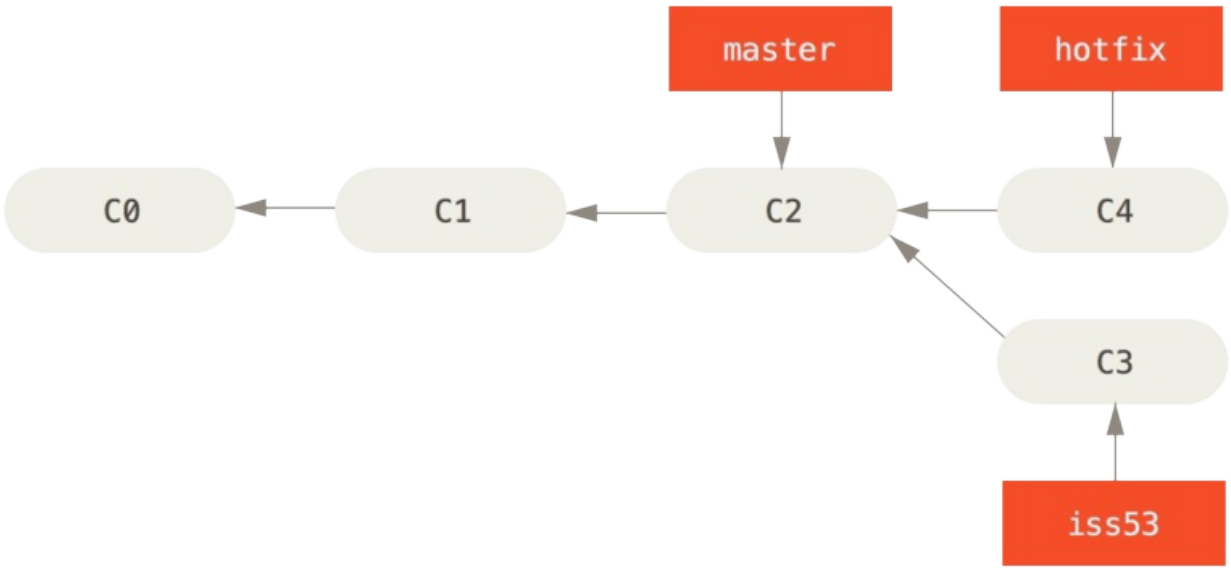


*Figure 22 : Branche de correctif basée sur \`master\`.*

Figure 21. Branche de correctif basée sur `master`

Vous pouvez lancer vos tests, vous assurer que la correction est efficace et la fusionner dans la branche `master` pour la déployer en production. Vous réalisez ceci au moyen de la commande `git merge` :

```
$ git checkout master
$ git merge correctif
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
```

```
 1 file changed, 2 insertions(+)
```

Vous noterez la mention `fast-forward` lors de cette fusion (*merge*). Comme le *commit* `C4` pointé par la branche `hotfix` que vous avez fusionnée était directement devant le *commit* `C2` sur lequel vous vous trouvez, Git a simplement déplacé le pointeur (vers l'avant). Autrement dit, lorsque l'on cherche à fusionner un *commit* qui peut être atteint en parcourant l'historique depuis le *commit* d'origine, Git se contente d'avancer le pointeur car il n'y a pas de travaux divergents à fusionner — ceci s'appelle un `fast-forward` (avance rapide).

Votre modification est maintenant dans l'instantané (*snapshot*) du *commit* pointé par la branche `master` et vous pouvez déployer votre correctif.

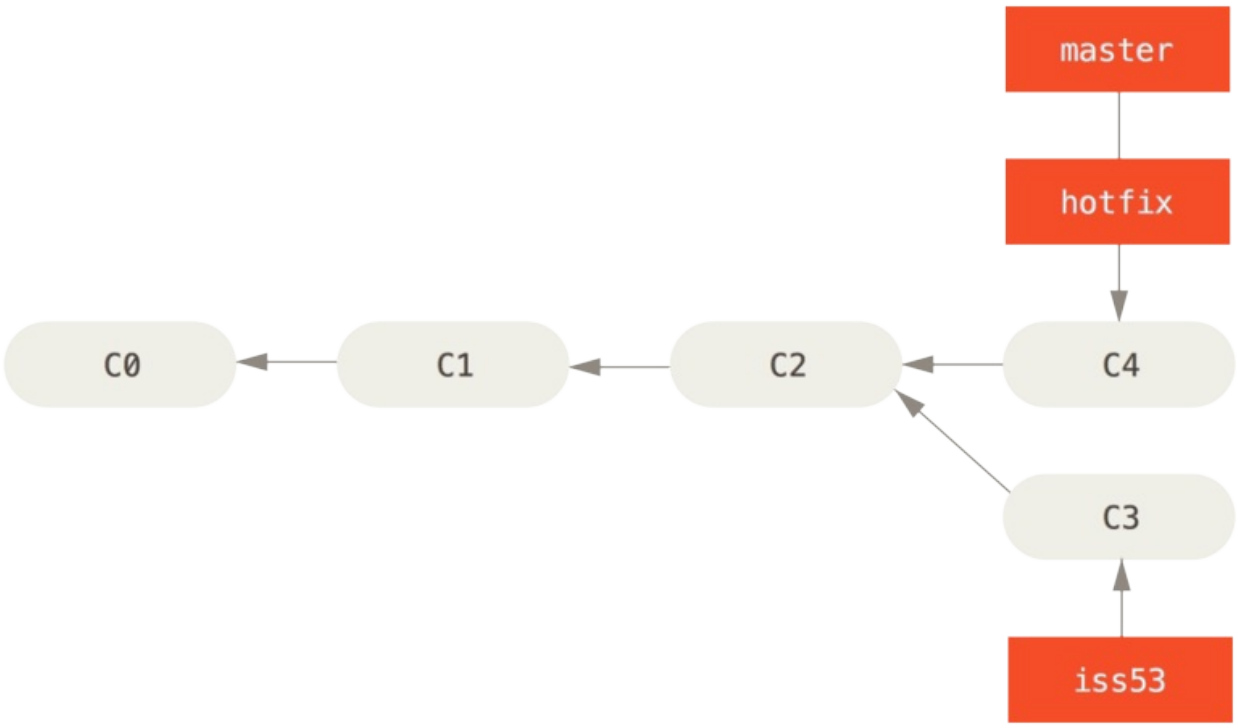


*Figure 23 : Avancement du pointeur de \`master\` sur \`correctif\`.*

Figure 22. Avancement du pointeur de `master` sur `correctif`

Après le déploiement de votre correctif super-important, vous voilà prêt à retourner travailler sur le sujet qui vous occupait avant l'interruption. Cependant, vous allez avant cela effacer la branche `correctif` dont vous n'avez plus besoin puisque la branche `master` pointe au même endroit. Vous pouvez l'effacer avec l'option `-d` de la commande `git branch` :

```
$ git branch -d correctif
Deleted branch correctif (3a0874c).
```

Maintenant, vous pouvez retourner travailler sur la branche qui contient vos travaux en cours pour le problème #53 :

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Nouveau pied de page terminé [issue 53]'
[iss53 ad82d7a] Nouveau pied de page terminé [issue 53]
1 file changed, 1 insertion(+)
```

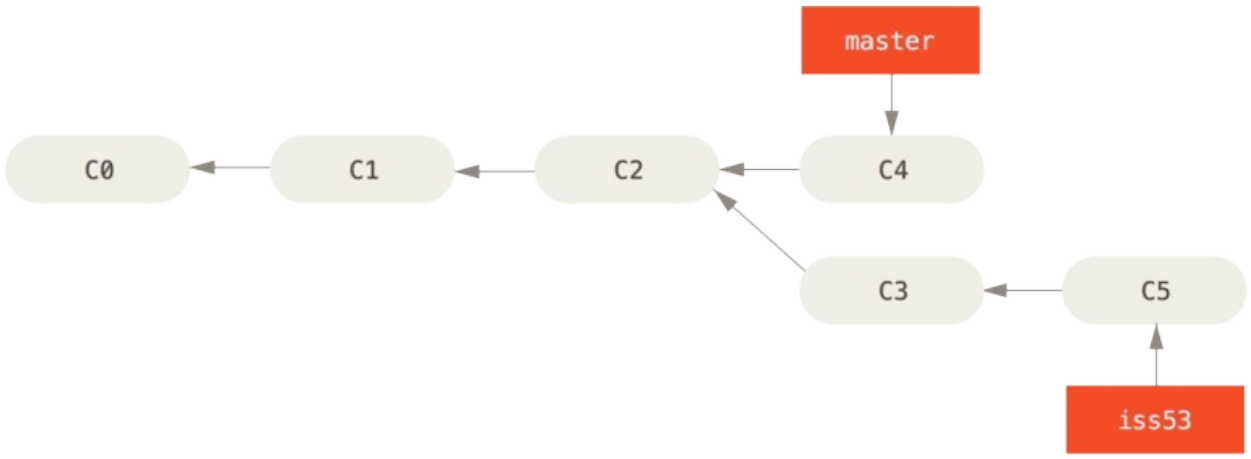


*Figure 24 : Le travail continue sur \`iss53\`.*

Figure 23. Le travail continue sur `iss53`

Il est utile de noter que le travail réalisé dans la branche `correctif` n'est pas contenu dans les fichiers de la branche `iss53`. Si vous avez besoin de les y rapatrier, vous pouvez fusionner la branche `master` dans la branche `iss53` en lançant la commande `git merge master`, ou vous pouvez retarder l'intégration de ces modifications jusqu'à ce que vous décidiez plus tard de rapatrier la branche `iss53` dans `master`.

## Fusions (*Merges*)

Supposons que vous ayez décidé que le travail sur le problème #53 était terminé et prêt à être fusionné dans la branche `master`. Pour ce faire, vous allez fusionner votre branche `iss53` de la même manière que vous l'avez fait plus tôt pour la branche `correctif`. Tout ce que vous avez à faire est d'extraire la branche dans laquelle vous souhaitez fusionner et lancer la commande `git merge` :

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
README |    1 +
1 file changed, 1 insertion(+)
```

Le comportement semble légèrement différent de celui observé pour la fusion précédente de la branche `correctif`. Dans ce cas, à un certain moment, l'historique de développement a divergé. Comme le *commit* sur la branche sur laquelle vous vous trouvez n'est plus un ancêtre direct de la branche que vous cherchez à fusionner, Git doit effectuer quelques actions. Dans ce cas, Git réalise une simple fusion à trois sources (*three-way merge*), en utilisant les deux instantanés pointés par les sommets des branches ainsi que leur plus proche ancêtre commun.

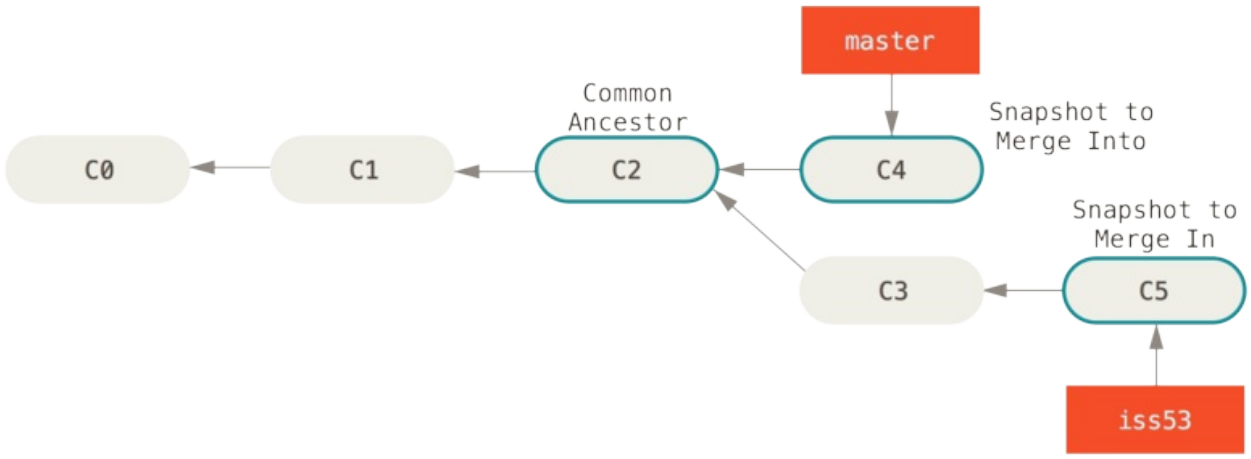


*Figure 25 : Trois instantanés utilisés dans une fusion classique.*

Figure 24. Trois instantanés utilisés dans une fusion classique

Au lieu d'avancer simplement le pointeur de branche, Git crée un nouvel instantané qui résulte de la fusion à trois sources et crée automatiquement un nouveau *commit* qui pointe dessus. On appelle ceci un *commit* de fusion (*merge commit*) qui est spécial en cela qu'il a plus d'un parent.

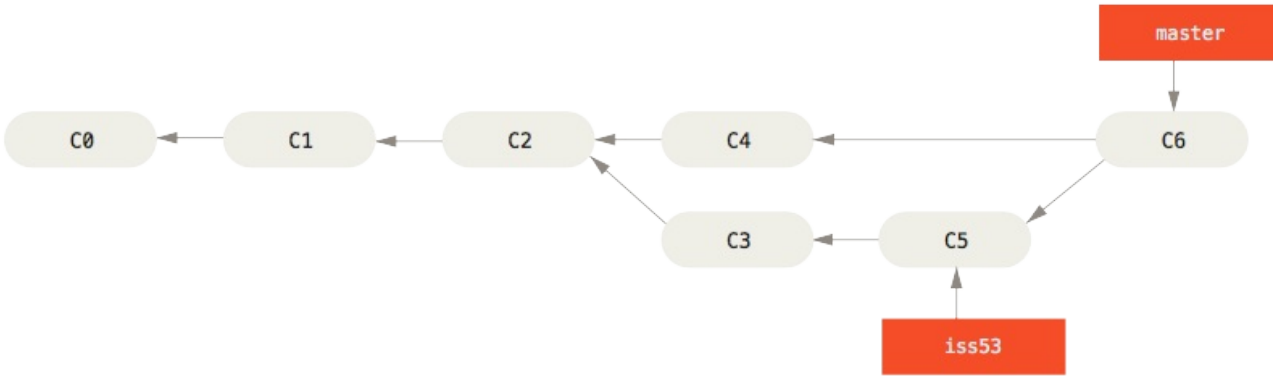


*Figure 26 : Un _commit_ de fusion.*

Figure 25. Un *commit* de fusion

À présent que votre travail a été fusionné, vous n'avez plus besoin de la branche `iss53`. Vous pouvez fermer le ticket dans votre outil de suivi des tâches et supprimer la branche :

```
$ git branch -d iss53
```

## Conflits de fusions (*Merge conflicts*)

Quelques fois, le processus ci-dessus ne se déroule pas aussi bien. Si vous avez modifié différemment la même partie du même fichier dans les deux branches que vous souhaitez fusionner, Git ne sera pas capable de réaliser proprement la fusion. Si votre résolution du problème #53 a modifié la même section de fichier que le `correctif`, vous obtiendrez un conflit qui ressemblera à ceci :

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git n'a pas automatiquement créé le *commit* de fusion. Il a arrêté le processus le temps que vous résolviez le conflit. Si vous voulez vérifier, à tout moment après l'apparition du conflit, quels fichiers n'ont pas été fusionnés, vous pouvez lancer la commande `git status` :

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Tout ce qui comporte des conflits et n'a pas été résolu est listé comme `unmerged`. Git ajoute des marques de résolution de conflit standards dans les fichiers qui comportent des conflits, pour que vous puissiez les ouvrir et résoudre les conflits manuellement. Votre fichier contient des sections qui ressemblent à ceci :

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Cela signifie que la version dans `HEAD` (votre branche `master`, parce que c'est celle que vous aviez extraite quand vous avez lancé votre commande de fusion) est la partie supérieure de ce bloc (tout ce qui se trouve au-dessus de la ligne `=======`), tandis que la version de votre branche `iss53` se trouve en dessous. Pour résoudre le conflit, vous devez choisir une partie ou l'autre ou bien fusionner leurs contenus vous-même. Par exemple, vous pourriez choisir de résoudre ce conflit en remplaçant tout le bloc par ceci :

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Cette résolution comporte des éléments de chaque section et les lignes `<<<<<<<`, `=======` et `>>>>>>>` ont été complètement effacées. Après avoir résolu chacune de ces sections dans chaque fichier comportant un conflit, lancez `git add` sur chaque fichier pour le marquer comme résolu. Placer le fichier dans l'index marque le conflit comme résolu pour Git.

Si vous souhaitez utiliser un outil graphique pour résoudre ces conflits, vous pouvez lancer `git mergetool` qui démarre l'outil graphique de fusion approprié et vous permet de naviguer dans les conflits :

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
```

```
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Si vous souhaitez utiliser un outil de fusion autre que celui par défaut (Git a choisi `opendiff` dans ce cas car la commande a été lancée depuis un Mac), vous pouvez voir tous les outils supportés après l'indication « *of the following tools:* ». Entrez simplement le nom de l'outil que vous préféreriez utiliser.

| | |
|---|---|
| | Si vous avez besoin d'outils plus avancés pour résoudre des conflits complexes, vous trouverez davantage d'informations au chapitre Fusion avancée. |

Après avoir quitté l'outil de fusion, Git vous demande si la fusion a été réussie. Si vous répondez par la positive à l'outil, il ajoute le fichier dans l'index pour le marquer comme résolu.

Vous pouvez lancer à nouveau la commande `git status` pour vérifier que tous les conflits ont été résolus :

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Si cela vous convient et que vous avez vérifié que tout ce qui comportait des conflits a été ajouté à l'index, vous pouvez entrer la commande `git commit` pour finaliser le *commit* de fusion. Le message de validation par défaut ressemble à ceci :

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

Vous pouvez modifier ce message pour inclure les détails sur la manière dont le conflit a été résolu si vous pensez que cela peut être utile lors d'une revue ultérieure. Indiquez pourquoi vous avez fait ces choix, si ce n'est pas clair.

# Gestion des branches

Maintenant que vous avez créé, fusionné et supprimé des branches, regardons de plus près les outils de gestion des branches qui s'avèreront utiles lors d'une utilisation intensive des branches.

La commande `git branch` permet en fait bien plus que la simple création et suppression de branches. Si vous la lancez sans argument, vous obtenez la liste des branches courantes :

```
$ git branch
  iss53
* master
  test
```

Notez le caractère `*` qui préfixe la branche `master` : il indique la branche courante (c'est-à-dire la branche sur laquelle le pointeur `HEAD` se situe). Ceci signifie que si, dans cette situation, vous validez des modifications (grâce à `git commit`), le pointeur de la branche `master` sera mis à jour pour inclure vos modifications. Pour visualiser la liste des derniers *commits* sur chaque branche, vous pouvez utiliser le commande `git branch -v` :

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  test 782fd34 add scott to the author list in the readmes
```

`--merged` et `--no-merged` sont des options très utiles qui permettent de filtrer les branches de cette liste selon que vous les avez ou ne les avez pas encore fusionnées avec la branche courante. Pour voir quelles branches ont déjà été fusionnées dans votre branche courante, lancez `git branch --merged` :

```
$ git branch --merged
  iss53
* master
```

Comme vous avez déjà fusionné `iss53` un peu plus tôt, vous la voyez dans votre liste. Les branches de cette liste qui ne comportent pas le préfixe `*` peuvent généralement être effacées sans risque au moyen de `git branch -d` puisque vous avez déjà intégré leurs modifications dans une autre branche et ne risquez donc pas de perdre quoi que ce soit.

Pour visualiser les branches qui contiennent des travaux qui n'ont pas encore été fusionnés, vous pouvez utiliser la commande `git branch --no-merged` :

```
$ git branch --no-merged
  test
```

Ceci affiche votre autre branche. Comme elle contient des modifications qui n'ont pas encore été intégrées, essayer de les supprimer par la commande `git branch -d` se solde par un échec :

```
$ git branch -d test
error: The branch 'test' is not fully merged.
If you are sure you want to delete it, run 'git branch -D test'.
```

Si vous souhaitez réellement supprimer cette branche et perdre ainsi le travail réalisé, vous pouvez tout de même forcer la suppression avec l'option `-D`, comme l'indique le message.

# Travailler avec les branches

Maintenant que vous avez acquis les bases concernant les branches et les fusions (*merges*), que pouvez-vous ou devez-vous en faire ? Ce chapitre traite des différents processus que cette gestion de branche légère permet de mettre en place, de manière à vous aider à décider si vous souhaitez en incorporer un dans votre cycle de développement.

## Branches au long cours

Comme Git utilise une *fusion à 3 sources*, fusionner une même branche dans une autre plusieurs fois sur une longue période est généralement facile. Cela signifie que vous pouvez avoir plusieurs branches ouvertes en permanence pour différentes phases de votre cycle de développement. Vous pourrez fusionner régulièrement ces branches entre elles.

De nombreux développeurs travaillent avec Git selon une méthode qui utilise cette approche. Il s'agit, par exemple, de n'avoir que du code entièrement stable et testé dans leur branche `master` ou bien même uniquement du code qui a été ou sera publié au sein d'une *release*. Ils ont alors en parallèle une autre branche appelée `develop` ou `next`. Cette branche accueille les développements en cours qui font encore l'objet de tests de stabilité — cette branche n'est pas nécessairement toujours stable mais quand elle le devient, elle peut être intégrée (via une fusion) dans `master`. Cette branche permet d'intégrer des branches thématiques (*topic branches* : branches de faible durée de vie telles que votre branche `iss53`), une fois prêtes, de manière à s'assurer qu'elles passent l'intégralité des tests et n'introduisent pas de bugs.

En réalité, nous parlons de pointeurs qui se déplacent le long des lignes des *commits* réalisés. Les branches stables sont plus basses dans l'historique des *commits* tandis que les branches des derniers développements sont plus hautes dans l'historique.

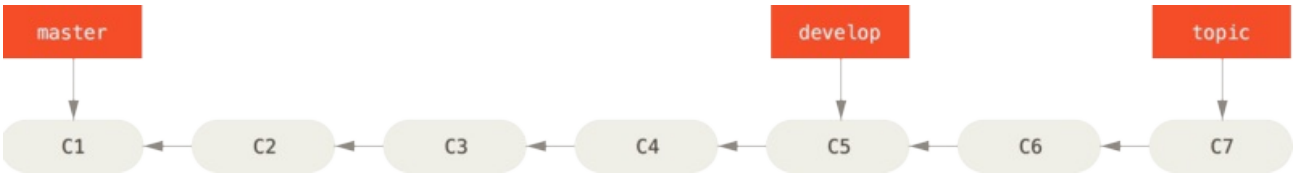


*Figure 27 : Vue linéaire de branches dans un processus de _stabilité progressive_.*

Figure 26. Vue linéaire de branches dans un processus de *stabilité progressive*

Il est généralement plus simple d'y penser en termes de silos de tâches où un ensemble de *commits* évolue progressivement vers un silo plus stable une fois qu'il a été complètement testé.

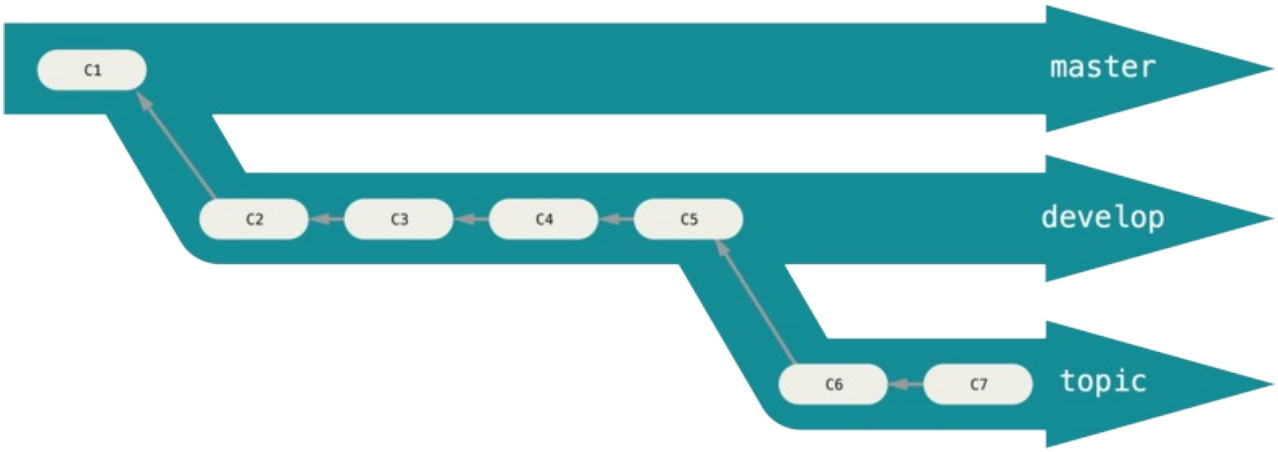


*Figure 28 : Vue _en silo_ de branches dans un processus de _stabilité progressive_.*

Figure 27. Vue *en silo* de branches dans un processus de *stabilité progressive*

Vous pouvez reproduire ce schéma sur plusieurs niveaux de stabilité. Des projets plus gros ont aussi une branche `proposed` ou `pu` (*proposed updates*) qui intègre elle-même des branches qui ne sont pas encore prêtes à être intégrées aux branches `next` ou `master`. L'idée est que les branches évoluent à différents niveaux de stabilité : quand elles atteignent un niveau plus stable, elles peuvent être fusionnées dans la branche de stabilité supérieure. Une fois encore, disposer de multiples branches au long cours n'est pas nécessaire mais s'avère souvent utile, spécialement dans le cadre de projets importants et complexes.

## Les branches thématiques

Les branches thématiques, elles, sont utiles quelle que soit la taille du projet. Une branche thématique est une branche ayant une courte durée de vie créée et utilisée pour une fonctionnalité ou une tâche particulière. C'est une méthode que vous n'avez probablement jamais utilisée avec un autre VCS parce qu'il y est généralement trop lourd de créer et fusionner des branches. Mais dans Git, créer, développer, fusionner et supprimer des branches plusieurs fois par jour est monnaie courante.

Vous avez déjà vu ces branches dans la section précédente avec les branches `iss53` et `correctif` que vous avez créées. Vous y avez réalisé quelques *commits* et vous les avez supprimées immédiatement après les avoir fusionnées dans votre branche principale. Cette technique vous permet de changer de contexte rapidement et complètement. Parce que votre travail est isolé dans des silos où toutes les modifications sont liées à une thématique donnée, il est beaucoup plus simple de réaliser des revues de code. Vous pouvez conserver vos modifications dans ces branches pendant des minutes, des jours ou des mois puis les fusionner quand elles sont prêtes, indépendamment de l'ordre dans lequel elles ont été créées ou traitées.

Prenons l'exemple suivant : alors que vous développez (sur `master`), vous créez une nouvelle branche pour un problème (`prob91`), travaillez un peu sur ce problème puis créez une seconde branche pour essayer de trouver une autre manière de le résoudre (`prob91v2`). Vous retournez ensuite sur la branche `master` pour y travailler pendant un moment puis finalement créez une dernière branche (`ideeidiote`) contenant une idée dont vous doutez de la pertinence. Votre historique de *commits* pourrait ressembler à ceci :

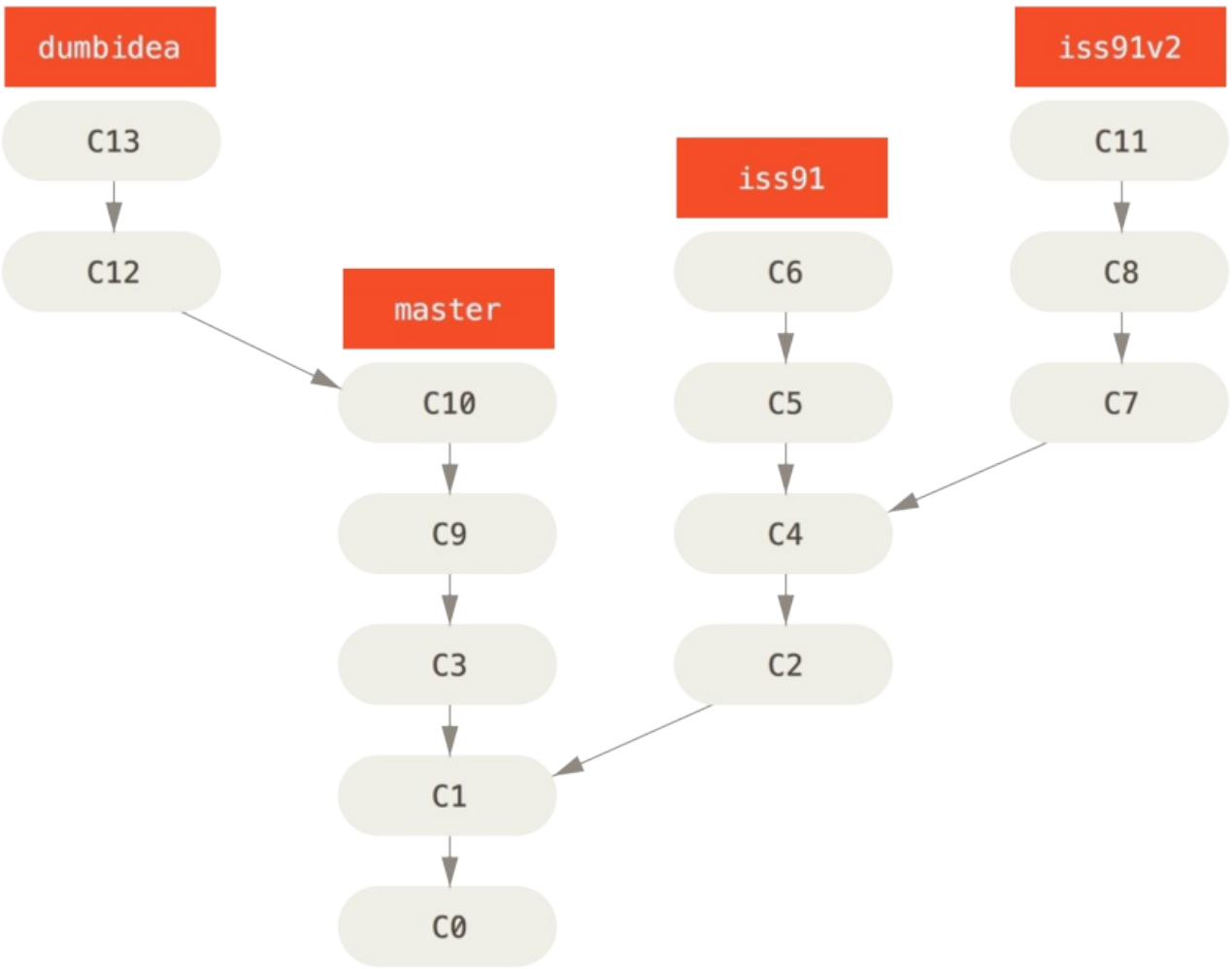


*Figure 29 : Branches thématiques multiples.*

Figure 28. Branches thématiques multiples

Maintenant, supposons que vous décidiez que vous préférez la seconde solution pour le problème ( `prob91v2` ) et que vous ayez montré la branche `ideeidiote` à vos collègues qui vous ont dit qu'elle était géniale. Vous pouvez jeter la branche `prob91` originale (perdant ainsi les *commits* `C5` et `C6` ) et fusionner les deux autres branches. Votre historique ressemble à présent à ceci :

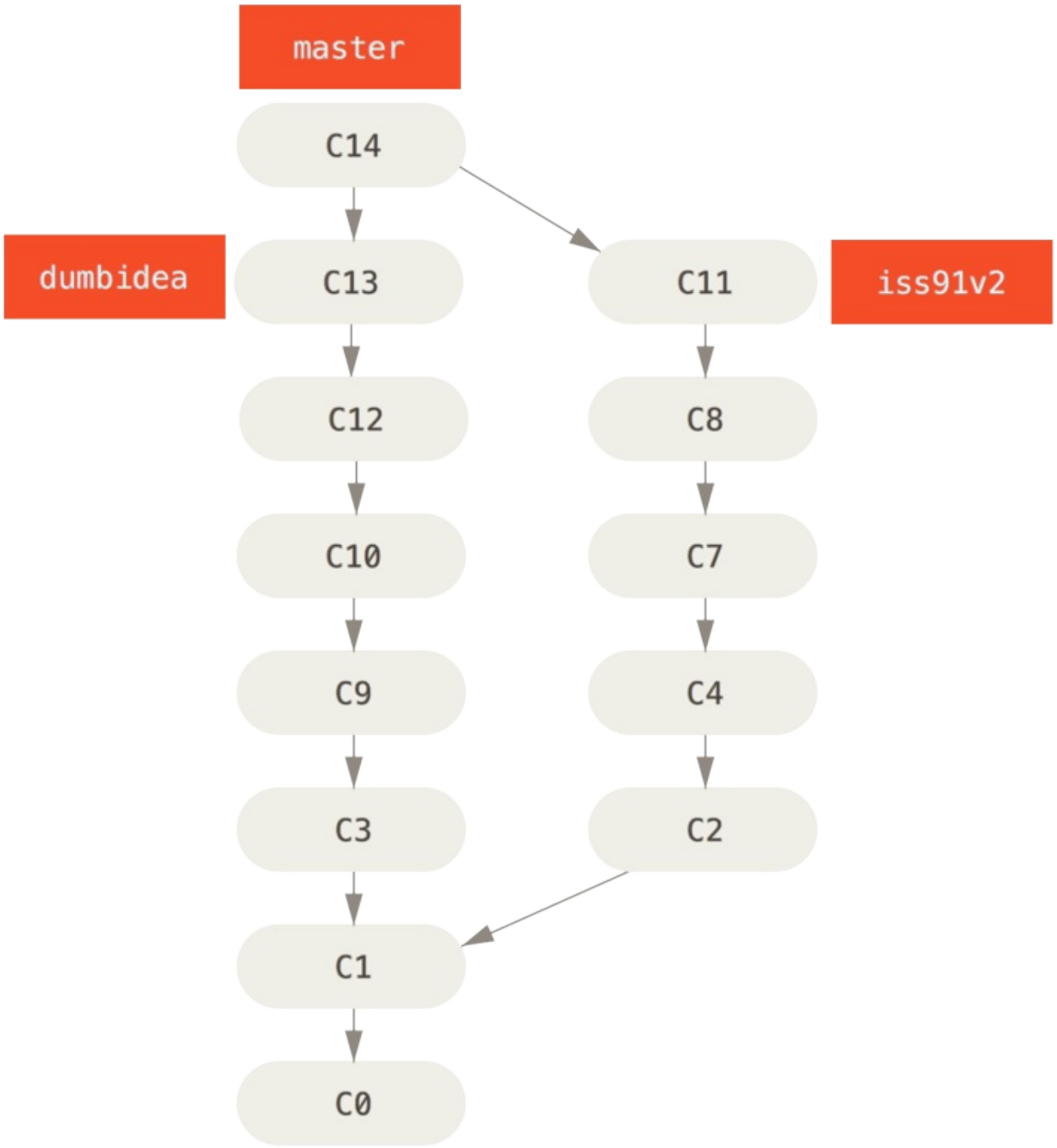


*Figure 30 : Historique après la fusion de \`ideeidiote\` et \`prob91v2\`.*

Figure 29. Historique après la fusion de `ideeidiote` et `prob91v2`

Nous verrons au chapitre Git distribué, d'autres méthodes et processus possibles pour vos projets Git. Nous vous invitons à prendre connaissance de ce chapitre avant de vous décider pour une méthode particulière de gestion de vos branches pour votre prochain projet.

Il est important de se souvenir que lors de la réalisation de toutes ces actions, ces branches sont complètement locales. Lorsque vous créez et fusionnez des branches, ceci est réalisé uniquement dans votre dépôt Git local et aucune communication avec un serveur n'a lieu.

# Branches de suivi à distance

Les références distantes sont des références (pointeurs) vers les éléments de votre dépôt distant tels que les branches, les tags, etc… Vous pouvez obtenir la liste complète de ces références distantes avec la commade `git ls-remote (remote)`, ou `git remote show (remote)`. Néanmoins, une manière plus courante consiste à tirer parti des branches de suivi à distance.

Les branches de suivi à distance sont des références (des pointeurs) vers l'état des branches sur votre dépôt distant. Ce sont des branches locales qu'on ne peut pas modifier ; elles sont modifiées automatiquement pour vous lors de communications réseau. Les branches de suivi à distance agissent comme des marques-pages pour vous indiquer l'état des branches sur votre dépôt distant lors de votre dernière connexion.

Elles prennent la forme de `(distant)/(branche)`. Par exemple, si vous souhaitiez visualiser l'état de votre branche `master` sur le dépôt distant `origin` lors de votre dernière communication, il vous suffirait de vérifier la branche `origin/master`. Si vous étiez en train de travailler avec un collègue et qu'il avait publié la branche `iss53`, vous pourriez avoir votre propre branche `iss53` ; mais la branche sur le serveur pointerait sur le *commit* de `origin/iss53`.

Cela peut être un peu déconcertant, essayons d'éclaircir les choses par un exemple. Supposons que vous avez un serveur Git sur le réseau à l'adresse `git.notresociete.com`. Si vous clonez à partir de ce serveur, la commande `clone` de Git le nomme automatiquement `origin`, tire tout son historique, crée un pointeur sur l'état actuel de la branche `master` et l'appelle localement `origin/master`. Git crée également votre propre branche `master` qui démarre au même endroit que la branche `master` d'origine, pour que vous puissiez commencer à travailler.

| | |
|---|---|
| | `origin` n'est pas spécial<br>De la même manière que le nom de branche `master` n'a aucun sens particulier pour Git, le nom `origin` n'est pas spécial. Tandis que `master` est le nom attribué par défaut à votre branche initiale lorsque vous lancez la commande `git init` et c'est la seule raison pour laquelle ce nom est utilisé aussi largement, `origin` est le nom utilisé par défaut pour un dépôt distant lorsque vous lancez `git clone`. Si vous lancez à la place `git clone -o booyah`, votre branche de suivi à distance par défaut s'appellera `booyah/master`. |

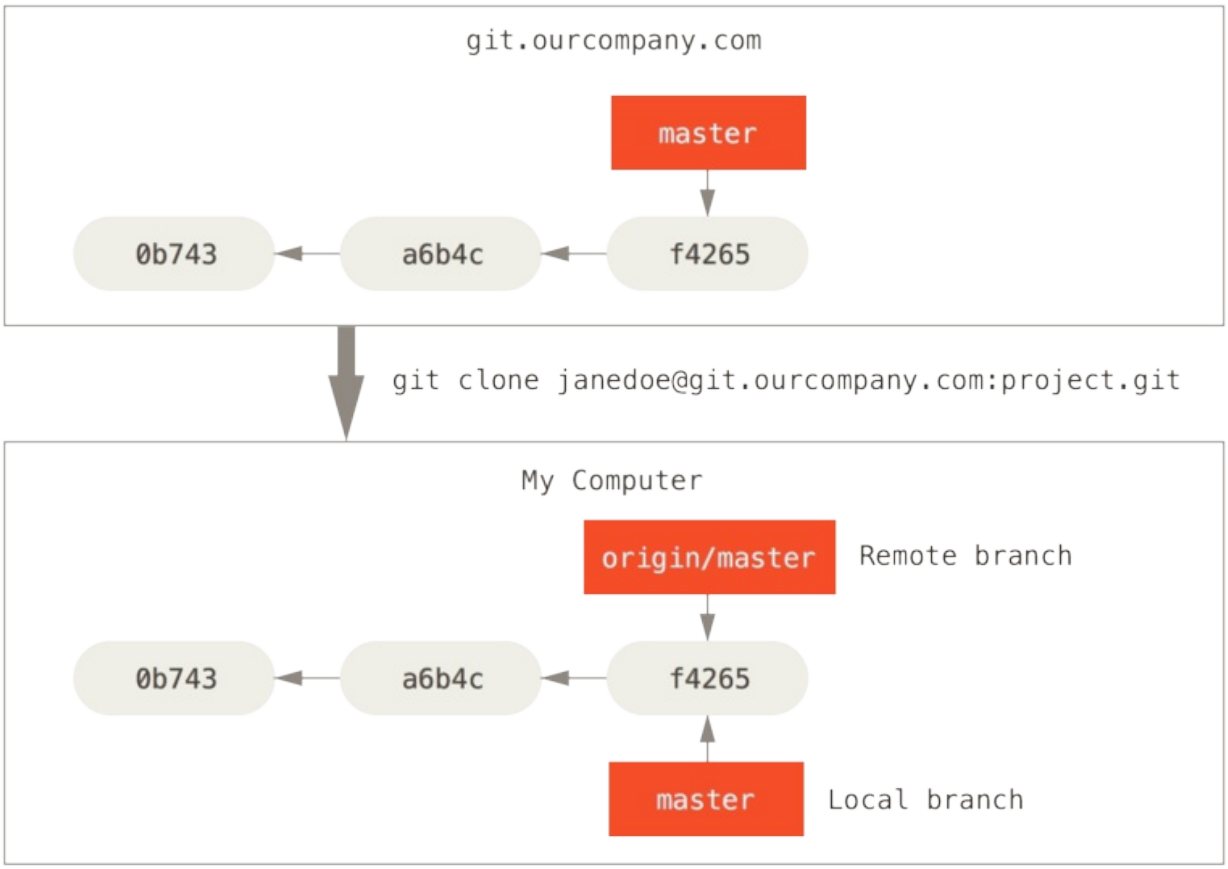


*Figure 31 : Dépôts distant et local après un _clone_.*

Figure 30. Dépôts distant et local après un *clone*

Si vous travaillez sur votre branche locale `master` et que dans le même temps, quelqu'un publie sur `git.notresociete.com` et met à jour cette même branche `master`, alors vos deux historiques divergent. Tant que vous restez sans contact avec votre serveur distant, votre pointeur vers `origin/master` n'avance pas.

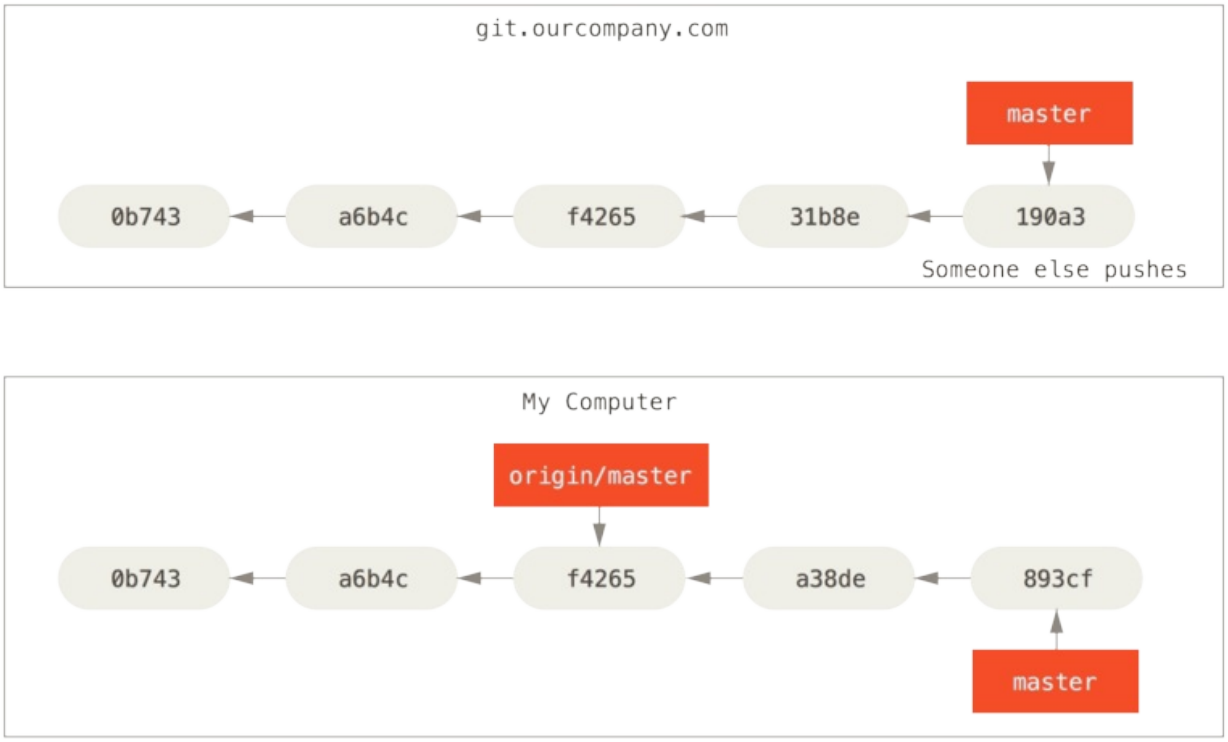


*Figure 32 : Les travaux locaux et distants peuvent diverger.*

Figure 31. Les travaux locaux et distants peuvent diverger

Lancez la commande `git fetch origin` pour synchroniser vos travaux. Cette commande recherche le serveur hébergeant `origin` (dans notre cas, `git.notresociete.com`), y récupère toutes les nouvelles données et met à jour votre base de donnée locale en déplaçant votre pointeur `origin/master` vers une nouvelle position, plus à jour.

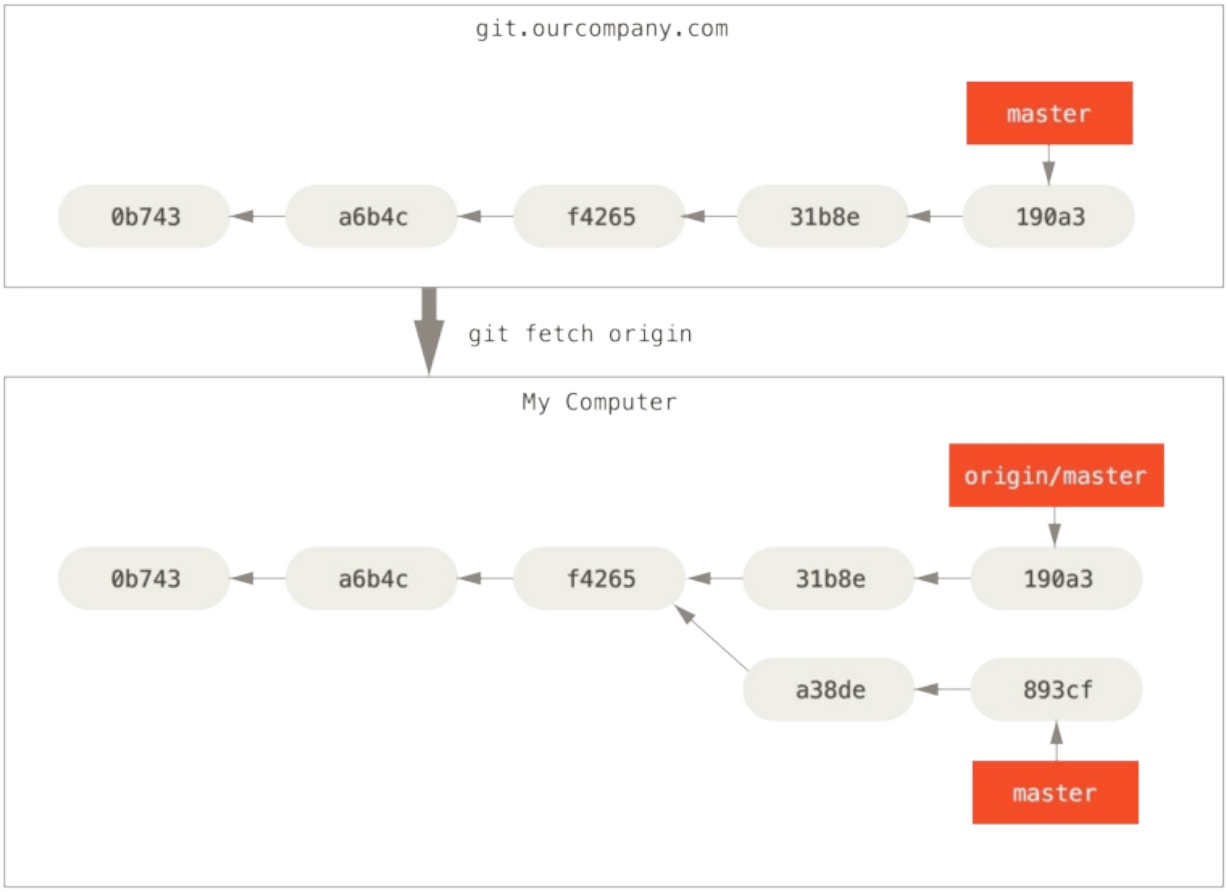


*Figure 33 : \`git fetch\` met à jour vos branches de suivi à distance.*

Figure 32. `git fetch` met à jour vos branches de suivi à distance

Pour démontrer l'usage de multiples serveurs distants et le fonctionnement des branches de suivi à distance pour ces projets distants, supposons que vous avez un autre serveur Git interne qui n'est utilisé que par une équipe de développeurs. Ce serveur se trouve sur `git.equipe1.notresociete.com` . Vous pouvez l'ajouter aux références distantes de votre projet en lançant la commande `git remote add` comme nous l'avons décrit au chapitre Les bases de Git. Nommez ce serveur distant `equipeun` qui sera le raccourci pour l'URL complète.

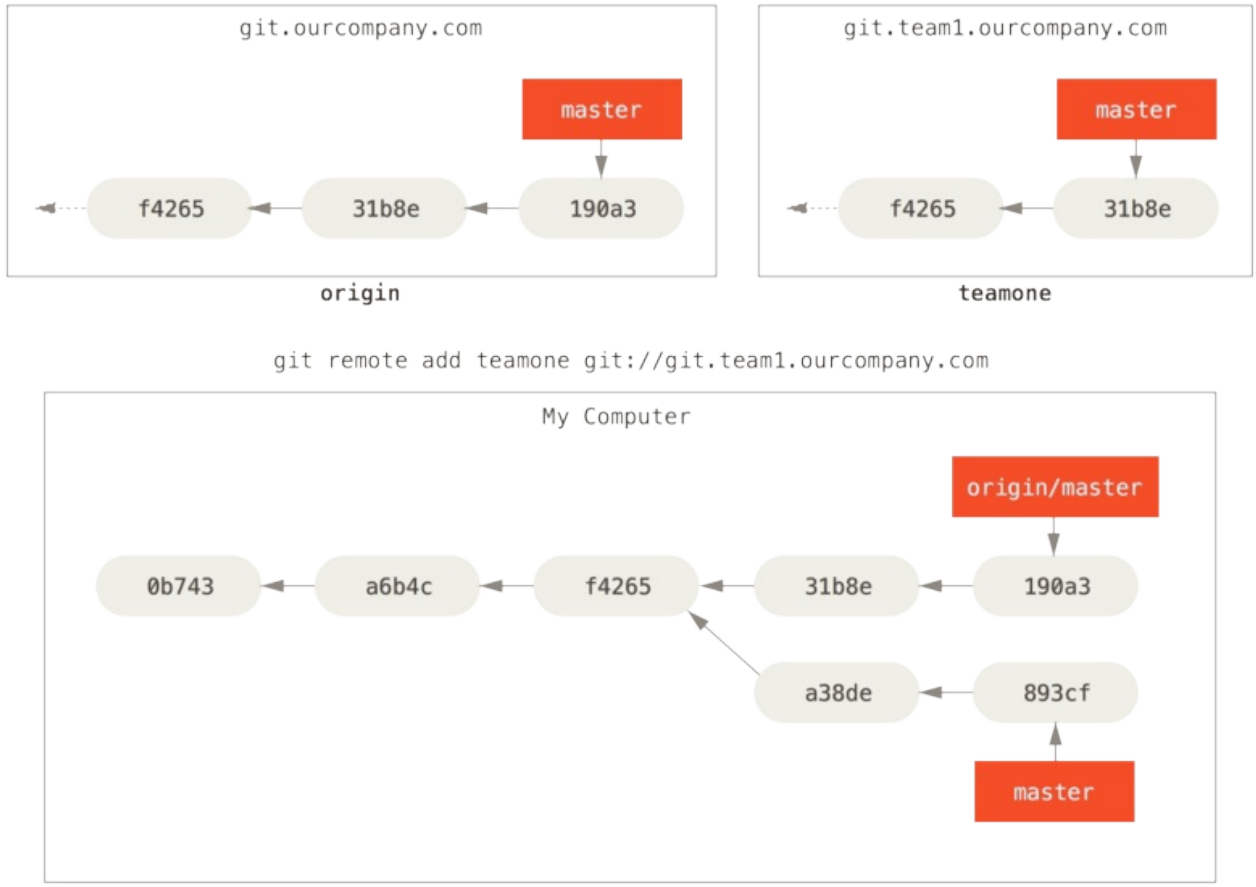


*Figure 34 : Ajout d'un nouveau serveur en tant que référence distante.*

Figure 33. Ajout d'un nouveau serveur en tant que référence distante

Maintenant, vous pouvez lancer `git fetch equipeun` pour récupérer l'ensemble des informations du serveur distant `equipeun` que vous ne possédez pas. Comme ce serveur contient déjà un sous-ensemble des données du serveur `origin` , Git ne récupère aucune donnée mais initialise une branche de suivi à distance appelée `equipeun/master` qui pointe sur le même *commit* que celui vers lequel pointe la branche `master` de `equipeun` .

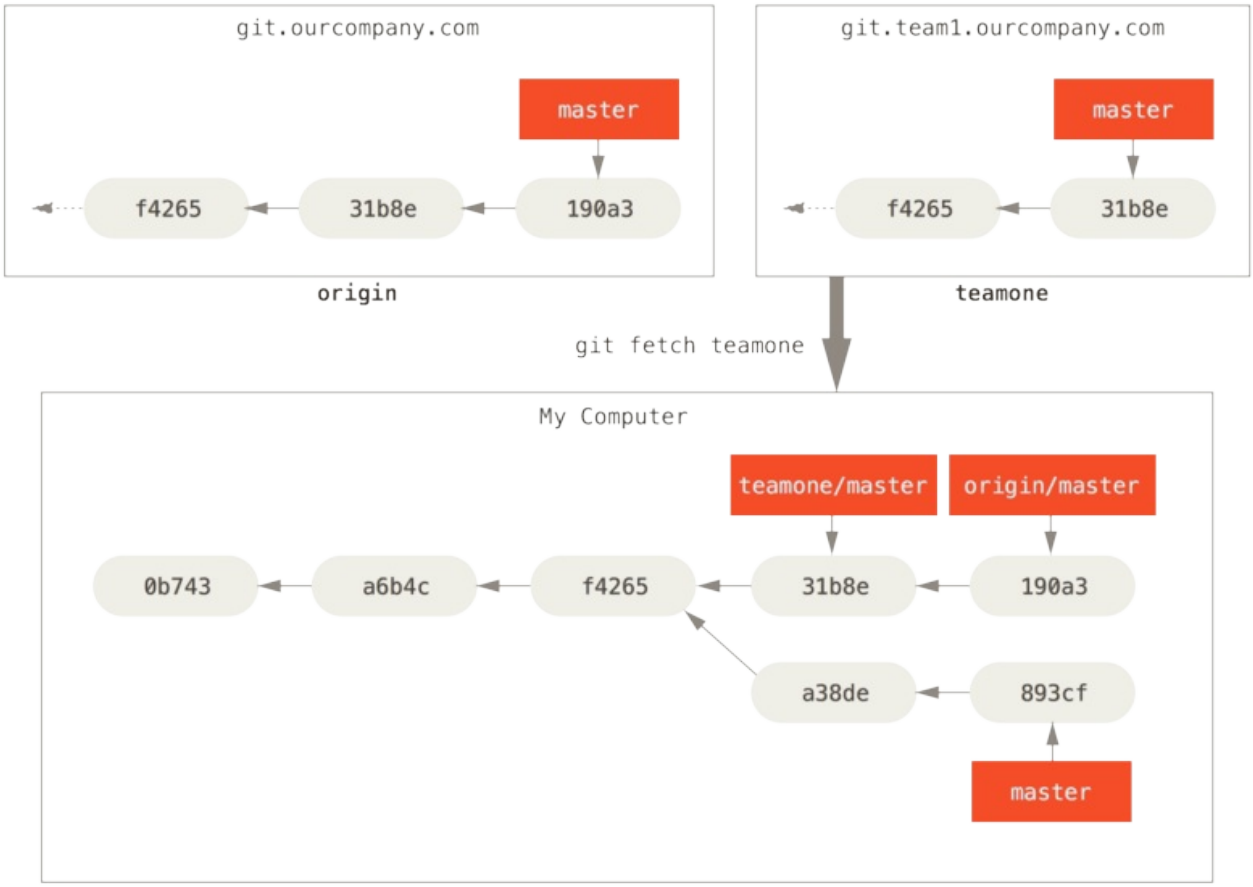


*Figure 35 : Branche de suivi à distance \`equipeun/master\`.*

Figure 34. Branche de suivi à distance `equipeun/master`

## Pousser les branches

Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur un serveur distant sur lequel vous avez accès en écriture. Vos branches locales ne sont pas automatiquement synchronisées sur les serveurs distants — vous devez pousser explicitement les branches que vous souhaitez partager. De cette manière, vous pouvez utiliser des branches privées pour le travail que vous ne souhaitez pas partager et ne pousser que les branches sur lesquelles vous souhaitez collaborer.

Si vous possédez une branche nommée `correctionserveur` sur laquelle vous souhaitez travailler avec d'autres, vous pouvez la pousser de la même manière que vous avez poussé votre première branche. Lancez `git push (serveur distant) (branche)` :

```
$ git push origin correctionserveur
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      correctionserveur -> correctionserveur
```

Il s'agit en quelque sorte d'un raccourci. Git développe automatiquement le nom de branche `correctionserveur` en `refs/heads/correctionserveur:refs/heads/correctionserveur`, ce qui signifie "Prendre ma branche locale `correctionserveur` et la pousser pour mettre à jour la branche distante `correctionserveur`". Nous traiterons plus en détail la partie `refs/heads/` au chapitre Les tripes de Git mais généralement, vous pouvez l'oublier. Vous pouvez aussi lancer `git push origin`

`correctionserveur:correctionserveur`, qui réalise la même chose — ce qui signifie « Prendre ma branche `correctionserveur` et en faire la branche `correctionserveur` distante ». Vous pouvez utiliser ce format pour pousser une branche locale vers une branche distante nommée différemment. Si vous ne souhaitez pas l'appeler `correctionserveur` sur le serveur distant, vous pouvez lancer à la place `git push origin correctionserveur:branchegeniale` pour pousser votre branche locale `correctionserveur` sur la branche `branchegeniale` sur le dépôt distant.

| | |
|---|---|
| | Ne renseignez pas votre mot de passe à chaque fois<br>Si vous utilisez une URL en HTTPS, le serveur Git vous demandera votre nom d'utilisateur et votre mot de passe pour vous authentifier. Par défaut, vous devez entrer ces informations sur votre terminal et le serveur pourra alors déterminer si vous être autorisé à pousser.<br><br>Si vous ne voulez pas entrer ces informations à chaque fois que vous poussez, vous pouvez mettre en place un "cache d'identification" (*credential cache*). Son fonctionnement le plus simple consiste à garder ces informations en mémoire pour quelques minutes mais vous pouvez configurer ce délai en lançant la commande `git config --global credential.helper cache`.<br><br>Pour davantage d'informations sur les différentes options de cache d'identification disponibles, vous pouvez vous référer au chapitre Stockage des identifiants. |

La prochaine fois qu'un de vos collègues récupère les données depuis le serveur, il récupérera, au sein de la branche de suivi à distance `origin/correctionserveur`, une référence vers l'état de la branche `correctionserveur` sur le serveur :

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      correctionserveur    -> origin/correctionserveur
```

Il est important de noter que lorsque vous récupérez une nouvelle branche depuis un serveur distant, vous ne créez pas automatiquement une copie locale éditable. En d'autres termes, il n'y a pas de branche `correctionserveur`, seulement un pointeur sur la branche `origin/correctionserveur` qui n'est pas modifiable.

Pour fusionner ce travail dans votre branche de travail actuelle, vous pouvez lancer la commande `git merge origin/correctionserveur`. Si vous souhaitez créer votre propre branche `correctionserveur` pour pouvoir y travailler, vous pouvez faire qu'elle repose sur le pointeur distant :

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

Cette commande vous fournit une branche locale modifiable basée sur l'état actuel de `origin/correctionserveur`.

## Suivre les branches

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement ce qu'on appelle une "branche de suivi" (*tracking branch*) et la branche qu'elle suit est appelée "branche amont" (*upstream branch*). Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante. Si vous vous trouvez sur une branche de suivi et que vous tapez `git push`, Git sélectionne automatiquement le serveur vers lequel pousser vos modifications. De même, un `git pull` sur une de ces branches récupère toutes les références distantes et fusionne automatiquement la branche distante correspondante dans la branche actuelle.

Lorsque vous clonez un dépôt, il crée généralement automatiquement une branche `master` qui suit `origin/master`. C'est pourquoi les commandes `git push` et `git pull` fonctionnent directement sans autre configuration. Vous pouvez néanmoins créer d'autres branches de suivi si vous le souhaitez, qui suivront des branches sur d'autres dépôts distants ou ne suivront pas la branche `master`. Un cas d'utilisation simple est l'exemple précédent, en lançant `git checkout -b [branche] [nomdistant]/[branche]`. C'est une opération suffisamment courante pour que Git propose l'option abrégée `--track` :

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

Pour créer une branche locale avec un nom différent de celui de la branche distante, vous pouvez simplement utiliser la première version avec un nom différent de branche locale :

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

À présent, votre branche locale `cs` poussera vers et tirera automatiquement depuis `origin/correctionserveur`.

Si vous avez déjà une branche locale et que vous voulez l'associer à une branche distante que vous venez de récupérer ou que vous voulez changer la branche distante que vous suivez, vous pouvez ajouter l'option `-u` ou `--set-upstream-to` à la commande `git branch` à tout moment.

```
$ git branch -u origin/correctionserveur
Branch correctionserveur set up to track remote branch correctionserveur from origin.
```

| | |
|---|---|
| | Raccourci vers *upstream*<br>Quand vous avez une branche de suivi configurée, vous pouvez faire référence à sa branche amont grâce au raccourci `@{upstream}` ou `@{u}`. Ainsi, si vous êtes sur la branche `master` qui suit `origin/master`, vous pouvez utiliser quelque chose comme `git merge @{u}` au lieu de `git merge origin/master` si vous le souhaitez. |

Si vous voulez voir quelles branches de suivi vous avez configurées, vous pouvez passer l'option `-vv` à `git branch`. Celle-ci va lister l'ensemble de vos branches locales avec quelques informations supplémentaires, y compris quelle est la branche suivie et si votre branche locale est devant, derrière ou les deux à la fois.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* correctionserveur f8674d9 [equipe1/correction-serveur-ok: ahead 3, behind 1] this should do it
  test   5ea463a trying something new
```

Vous pouvez constater ici que votre branche `iss53` suit `origin/iss53` et est *"devant de deux"*, ce qui signifie qu'il existe deux *commits* locaux qui n'ont pas été poussés au serveur. On peut aussi voir que la branche `master` suit `origin/master` et est à jour. On peut voir ensuite que notre branche `correctionserveur` suit la branche `correction-serveur-ok` sur notre serveur `equipe1` et est *"devant de trois"* et *"derrière de un"*, ce qui signifie qu'il existe un *commit* qui n'a pas été encore intégré localement et trois *commits* locaux qui n'ont pas été poussés. Finalement, on peut voir que notre branche `test` ne suit aucune branche distante.

Il est important de noter que ces nombres se basent uniquement sur l'état de votre branche distante la dernière fois qu'elle a été synchronisée depuis le serveur. Cette commande n'effectue aucune recherche sur les serveurs et ne travaille que sur les données locales qui ont été mises en cache depuis ces serveurs. Si vous voulez mettre complètement à jour ces nombres, vous devez préalablement synchroniser (*fetch*) toutes vos branches distantes depuis les serveurs. Vous pouvez le faire de cette façon : `$ git fetch --all; git branch -vv`.

### Tirer une branche (*Pulling*)

Bien que la commande `git fetch` récupère l'ensemble des changements présents sur serveur et qui n'ont pas déjà été rapatriés localement, elle ne modifie en rien votre répertoire de travail. Cette commande récupère simplement les données pour vous et vous laisse les fusionner par vous-même. Cependant, il existe une commande appelée `git pull` qui consiste essentiellement en un `git fetch` immédiatement suivi par un `git merge` dans la plupart des cas. Si vous disposez d'une branche de suivi configurée comme illustré dans le chapitre précédent, soit par une configuration explicite soit en ayant laissé les commandes `clone` ou `checkout` les créer pour vous, `git pull` va examiner quel serveur et quelle branche votre branche courante suit actuellement, synchroniser depuis ce serveur et ensuite essayer de fusionner cette branche distante avec la vôtre.

Il est généralement préférable de simplement utiliser les commandes `fetch` et `merge` explicitement plutôt que de laisser faire la magie de `git pull` qui peut s'avérer source de confusion.

### Suppression de branches distantes

Supposons que vous en avez terminé avec une branche distante – disons que vous et vos collaborateurs avez terminé une fonctionnalité et l'avez fusionnée dans la branche `master` du serveur distant (ou la branche correspondant à votre code stable). Vous pouvez effacer une branche distante en ajoutant l'option `--delete` à `git push`. Si vous souhaitez effacer votre branche `correctionserveur` du serveur, vous pouvez lancer ceci :

```
$ git push origin --delete correctionserveur
To https://github.com/schacon/simplegit
 - [deleted]         correctionserveur
```

En résumé, cela ne fait que supprimer le pointeur sur le serveur. Le serveur Git garde généralement les données pour un temps jusqu'à ce qu'un processus de nettoyage (*garbage collection*) passe. De cette manière, si une suppression accidentelle a eu lieu, les données sont souvent très facilement récupérables.

## Rebaser (*Rebasing*)

Dans Git, il y a deux façons d'intégrer les modifications d'une branche dans une autre : en fusionnant ( `merge` ) et en rebasant ( `rebase` ). Dans ce chapitre, vous apprendrez la signification de rebaser, comment le faire, pourquoi c'est un outil incroyable et dans quels cas il est déconseillé de l'utiliser.

### Les bases

Si vous revenez à un exemple précédent du chapitre Fusions (*Merges*), vous remarquerez que votre travail a divergé et que vous avez ajouté des *commits* sur deux branches différentes.

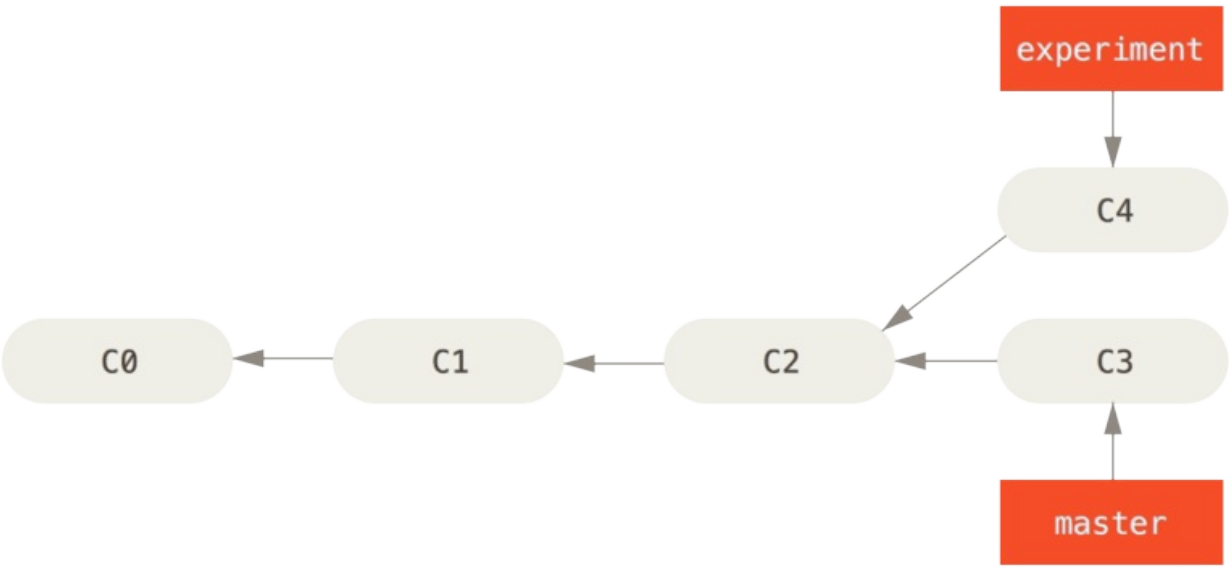


*Figure 36 : Historique divergeant simple.*

Figure 35. Historique divergeant simple

Comme nous l'avons déjà expliqué, le moyen le plus simple pour intégrer ces branches est la fusion via la commande `merge`. Cette commande réalise une *fusion à trois branches* entre les deux derniers instantanés (*snapshots*) de chaque branche (C3 et C4) et l'ancêtre commun le plus récent (C2), créant un nouvel instantané (et un *commit*).

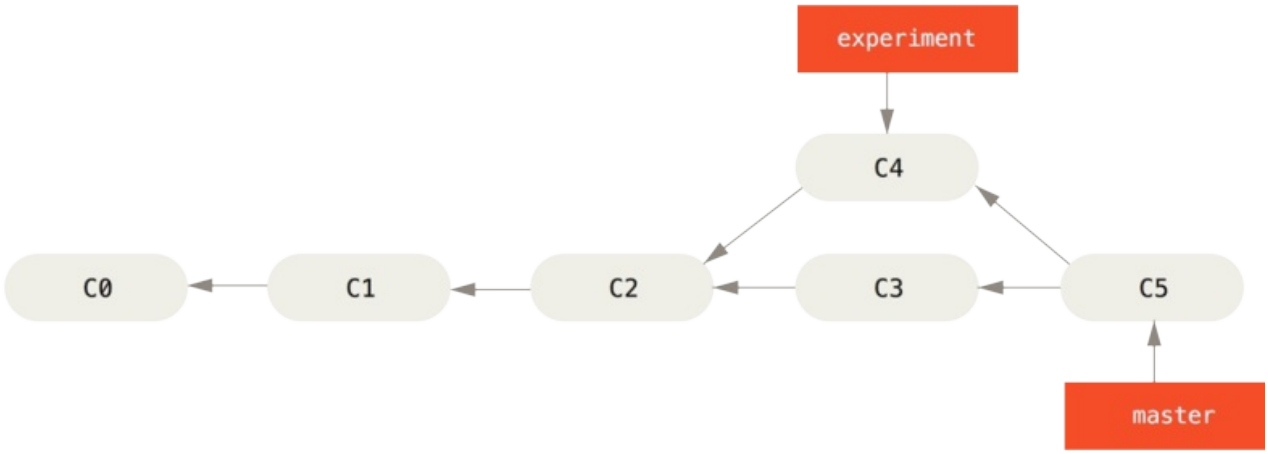


*Figure 37 : Fusion pour intégrer des travaux aux historiques divergeants.*

Figure 36. Fusion pour intégrer des travaux aux historiques divergeants

Cependant, il existe un autre moyen : vous pouvez prendre le *patch* de la modification introduite en `C4` et le réappliquer sur `C3`. Dans Git, cette action est appelée "rebaser" (*rebasing*). Avec la commande `rebase`, vous pouvez prendre toutes les modifications qui ont été validées sur une branche et les rejouer sur une autre.

Dans cet exemple, vous lanceriez les commandes suivantes :

```
$ git checkout experience
$ git rebase master
First, rewinding head to replay your work on top of it...
```

```
Applying: added staged command
```

Cela fonctionne en cherchant l'ancêtre commun le plus récent des deux branches (celle sur laquelle vous vous trouvez et celle sur laquelle vous rebasez), en récupérant toutes les différences introduites par chaque *commit* de la branche courante, en les sauvant dans des fichiers temporaires, en réinitialisant la branche courante sur le même *commit* que la branche de destination et en appliquant finalement chaque modification dans le même ordre.

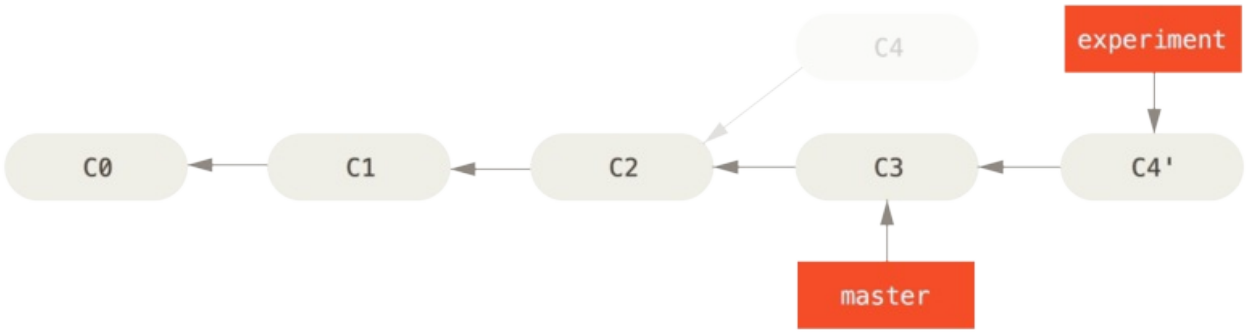


*Figure 38 : Rebasage des modifications introduites par \`C4\` sur \`C3\`.*

Figure 37. Rebasage des modifications introduites par `C4` sur `C3`

À ce moment, vous pouvez retourner sur la branche `master` et réaliser une fusion en avance rapide (*fast-forward merge*).

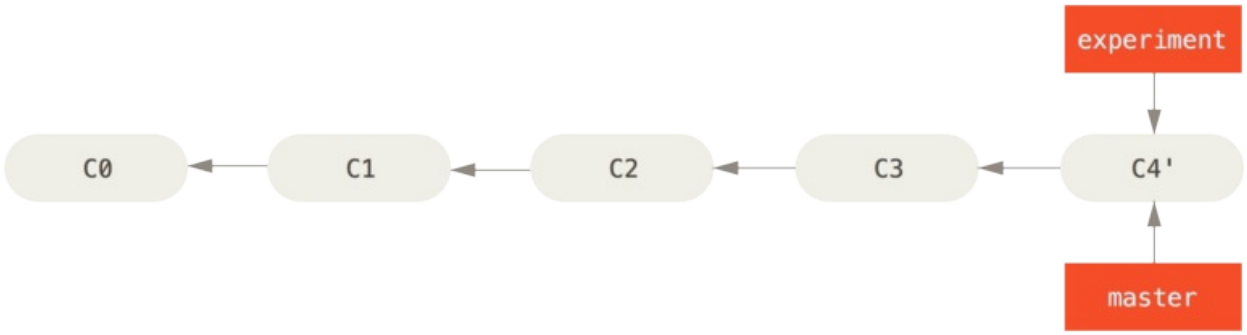


*Figure 39 : Avance rapide de la branche \`master\`.*

Figure 38. Avance rapide de la branche `master`

À présent, l'instantané pointé par `C4'` est exactement le même que celui pointé par `C5` dans l'exemple de fusion. Il n'y a pas de différence entre les résultats des deux types d'intégration, mais rebaser rend l'historique plus clair. Si vous examinez le journal de la branche rebasée, elle est devenue linéaire : toutes les modifications apparaissent en série même si elles ont eu lieu en parallèle.

Vous aurez souvent à faire cela pour vous assurer que vos *commits* s'appliquent proprement sur une branche distante — par exemple, sur un projet où vous souhaitez contribuer mais que vous ne maintenez pas. Dans ce cas, vous réaliseriez votre travail dans une branche puis vous rebaseriez votre travail sur `origin/master` quand vous êtes prêt à soumettre vos patchs au projet principal. De cette manière, le mainteneur n'a pas à réaliser de travail d'intégration — juste une avance rapide ou simplement une application propre.

Il faut noter que l'instantané pointé par le *commit* final, qu'il soit le dernier des *commits* d'une opération de rebasage ou le *commit* final issu d'une fusion, sont en fait le même instantané — c'est juste que l'historique est différent. Rebaser rejoue les modifications d'une ligne de *commits* sur une autre dans l'ordre d'apparition, alors que la fusion joint et fusionne les deux

têtes.

## Rebases plus intéressants

Vous pouvez aussi faire rejouer votre rebasage sur autre chose qu'une branche. Prenez un historique tel que Un historique avec deux branches thématiques qui sortent l'une de l'autre par exemple. Vous avez créé une branche thématique ( `serveur` ) pour ajouter des fonctionnalités côté serveur à votre projet et avez réalisé un *commit*. Ensuite, vous avez créé une branche pour ajouter des modifications côté client ( `client` ) et avez validé plusieurs fois. Finalement, vous avez rebasculé sur la branche `serveur` et avez réalisé quelques *commits* supplémentaires.

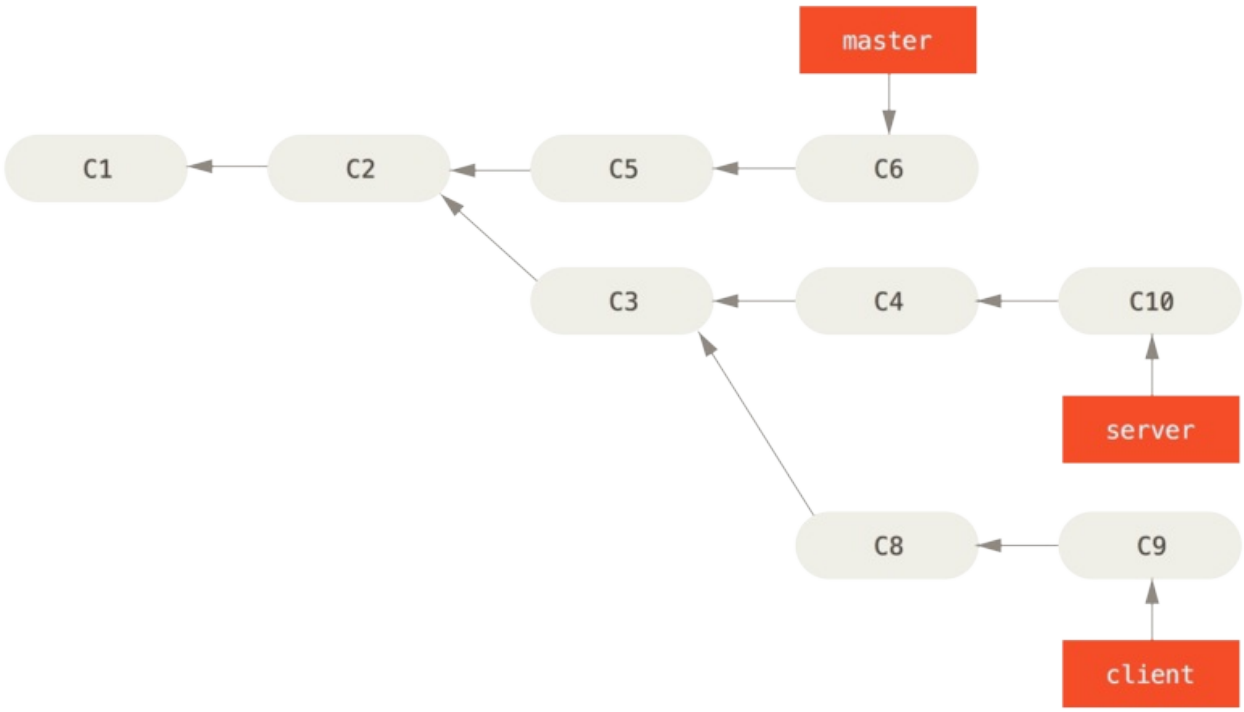


*Figure 40 : Un historique avec deux branches thématiques qui sortent l'une de l'autre.*

Figure 39. Un historique avec deux branches thématiques qui sortent l'une de l'autre

Supposons que vous décidez que vous souhaitez fusionner vos modifications du côté client dans votre ligne principale pour une publication (*release*) mais vous souhaitez retenir les modifications de la partie serveur jusqu'à ce qu'elles soient un peu mieux testées. Vous pouvez récupérer les modifications du côté client qui ne sont pas sur le serveur ( `C8` et `C9` ) et les rejouer sur la branche `master` en utilisant l'option `--onto` de `git rebase` :

```
$ git rebase --onto master serveur client
```

Cela signifie en substance "Extraire la branche client, déterminer les patchs depuis l'ancêtre commun des branches `client` et `serveur` puis les rejouer sur `master` ". C'est assez complexe, mais le résultat est assez impressionnant.

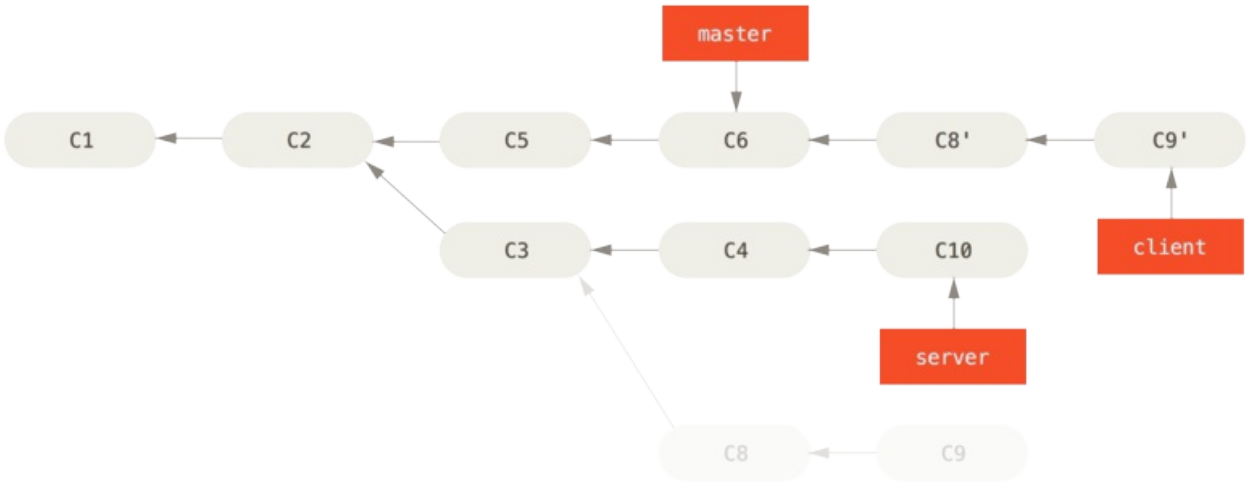


*Figure 41 : Rebaser deux branches thématiques l'une sur l'autre.*

Figure 40. Rebaser deux branches thématiques l'une sur l'autre

Maintenant, vous pouvez faire une avance rapide sur votre branche `master` (cf. Avance rapide sur votre branche `master` pour inclure les modifications de la branche client):

```
$ git checkout master
$ git merge client
```

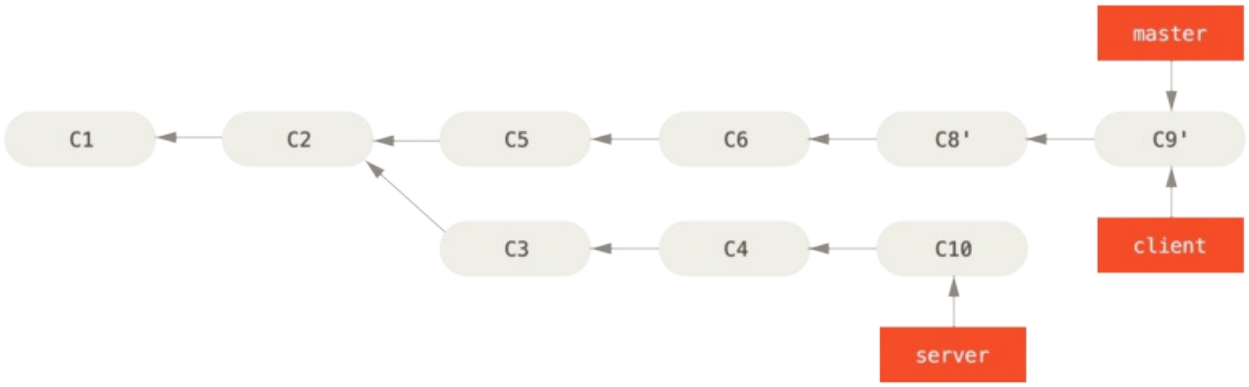


*Figure 42 : Avance rapide sur votre branche \`master\` pour inclure les modifications de la branche client.*

Figure 41. Avance rapide sur votre branche `master` pour inclure les modifications de la branche client

Supposons que vous décidiez de tirer (*pull*) votre branche `serveur` aussi. Vous pouvez rebaser la branche `serveur` sur la branche `master` sans avoir à l'extraire avant en utilisant `git rebase [branchedebase] [branchethematique]` — qui extrait la branche thématique (dans notre cas, `serveur`) pour vous et la rejoue sur la branche de base (`master`) :

```
$ git rebase master serveur
```

Cette commande rejoue les modifications de `serveur` sur le sommet de la branche `master`, comme indiqué dans Rebasage de la branche serveur sur le sommet de la branche `master`..

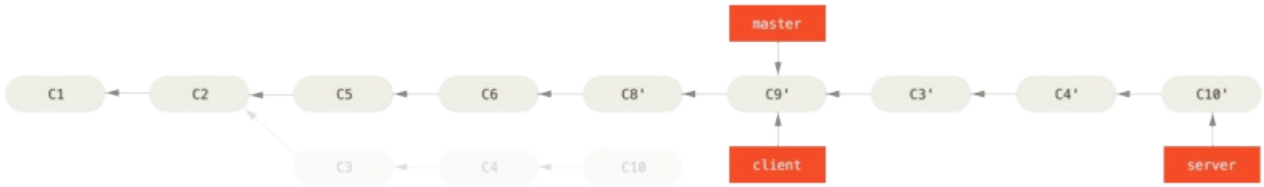


*Figure 43 : Rebasage de la branche serveur sur le sommet de la branche \`master\`.*

Figure 42. Rebasage de la branche serveur sur le sommet de la branche `master` .

Vous pouvez ensuite faire une avance rapide sur la branche de base ( `master` ) :

```
$ git checkout master
$ git merge serveur
```

Vous pouvez effacer les branches `client` et `serveur` une fois que tout le travail est intégré et que vous n'en avez plus besoin, éliminant tout l'historique de ce processus, comme visible sur Historique final des *commits* :

```
$ git branch -d client
$ git branch -d serveur
```


*Figure 44 : Historique final des _commits_.*

Figure 43. Historique final des *commits*

## Les dangers du rebasage

Ah… mais les joies de rebaser ne viennent pas sans leurs contreparties, qui peuvent être résumées en une ligne :

**Ne rebasez jamais des *commits* qui ont déjà été poussés sur un dépôt public.**

Si vous suivez ce conseil, tout ira bien. Sinon, de nombreuses personnes vont vous haïr et vous serez méprisé par vos amis et votre famille.

Quand vous rebasez des données, vous abandonnez les *commits* existants et vous en créez de nouveaux qui sont similaires mais différents. Si vous poussez des *commits* quelque part, que d'autres les tirent et se basent dessus pour travailler, et qu'après coup, vous réécrivez ces *commits* à l'aide de `git rebase` et les poussez à nouveau, vos collaborateurs devront re-fusionner leur travail et les choses peuvent rapidement devenir très désordonnées quand vous essaierez de tirer leur travail dans votre dépôt.

Examinons un exemple expliquant comment rebaser un travail déjà publié sur un dépôt public peut générer des gros problèmes. Supposons que vous clonez un dépôt depuis un serveur central et réalisez quelques travaux dessus. Votre historique de *commits* ressemble à ceci :

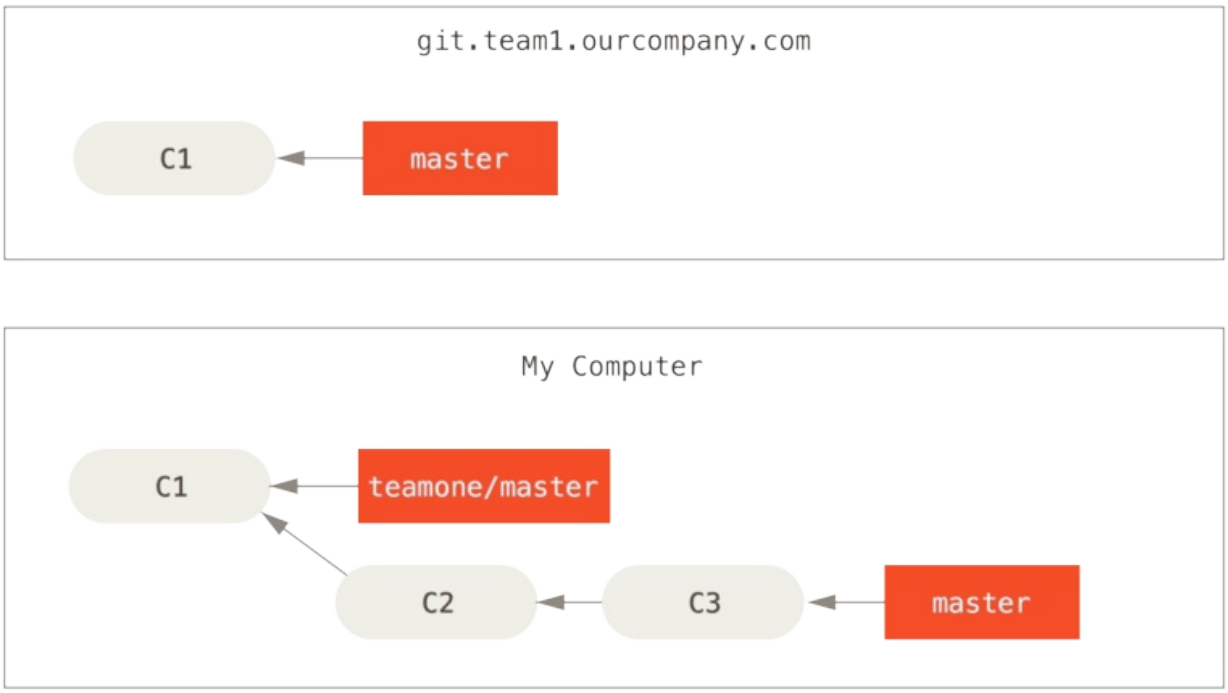


*Figure 45 : Cloner un dépôt et baser du travail dessus.*

Figure 44. Cloner un dépôt et baser du travail dessus

À présent, une autre personne travaille et inclut une fusion, puis elle pousse ce travail sur le serveur central. Vous le récupérez et vous fusionnez la nouvelle branche distante dans votre copie, ce qui donne l'historique suivant :

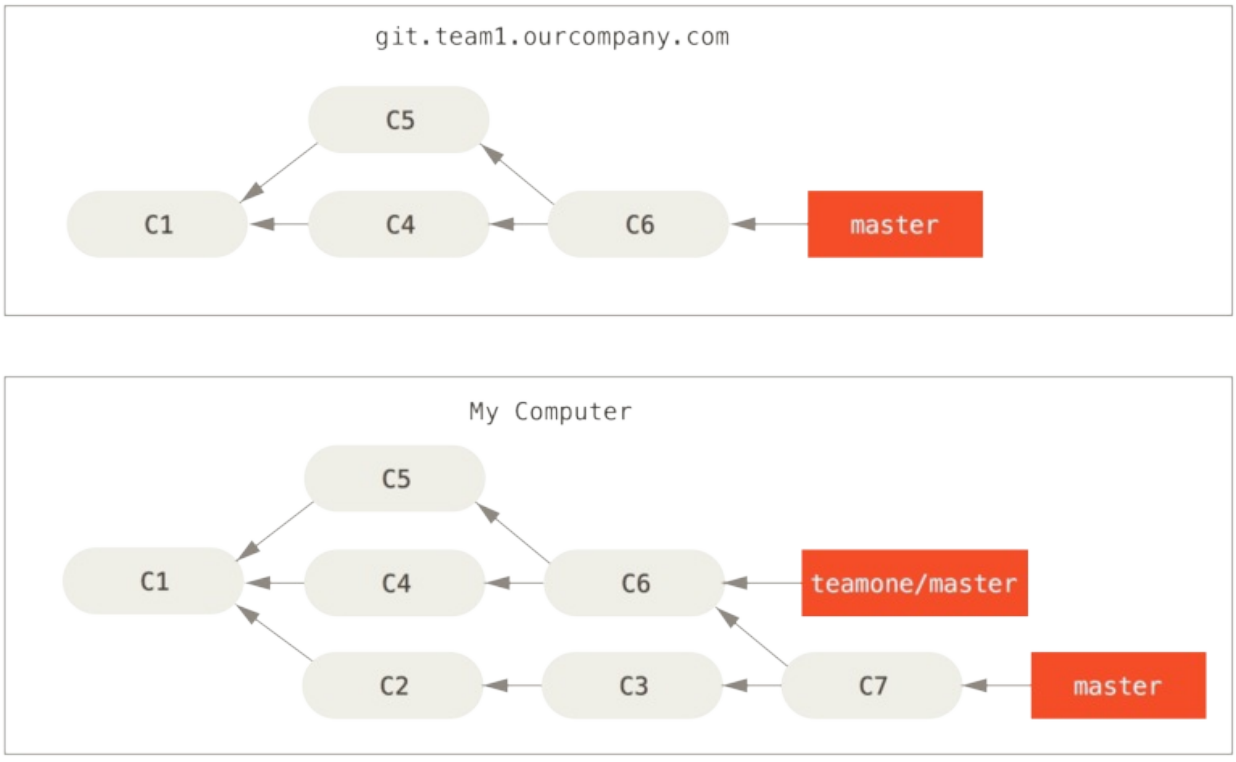

*Figure 46 : Récupération de _commits_ et fusion dans votre copie.*

Figure 45. Récupération de *commits* et fusion dans votre copie

Ensuite, la personne qui a poussé le travail que vous venez de fusionner décide de faire marche arrière et de rebaser son travail. Elle lance un `git push --force` pour forcer l'écrasement de l'historique sur le serveur. Vous récupérez alors les données du serveur, qui vous amènent les nouveaux *commits*.

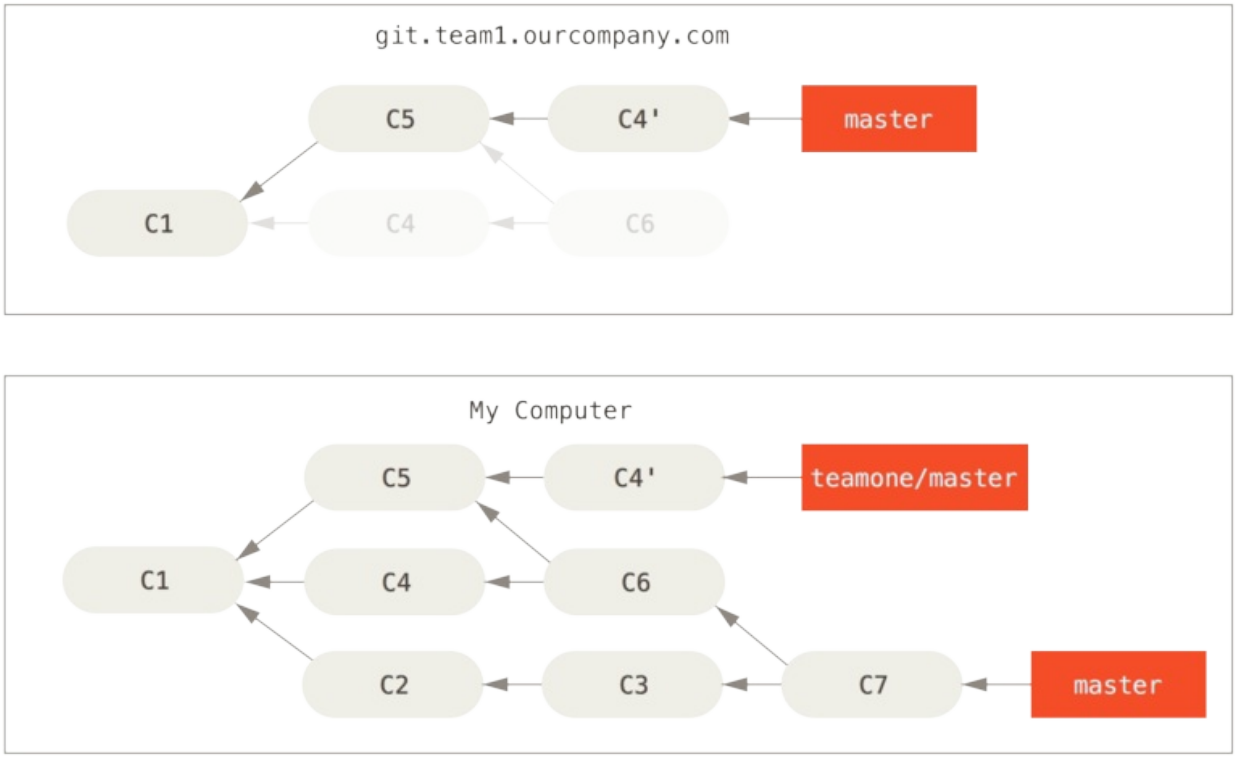


*Figure 47 : Quelqu'un pousse des _commits_ rebasés, en abandonnant les _commits_ sur lesquels vous avez fondé votre travail.*

Figure 46. Quelqu'un pousse des *commits* rebasés, en abandonnant les *commits* sur lesquels vous avez fondé votre travail

Vous êtes désormais tous les deux dans le pétrin. Si vous faites un `git pull`, vous allez créer un *commit* de fusion incluant les deux historiques et votre dépôt ressemblera à ça :

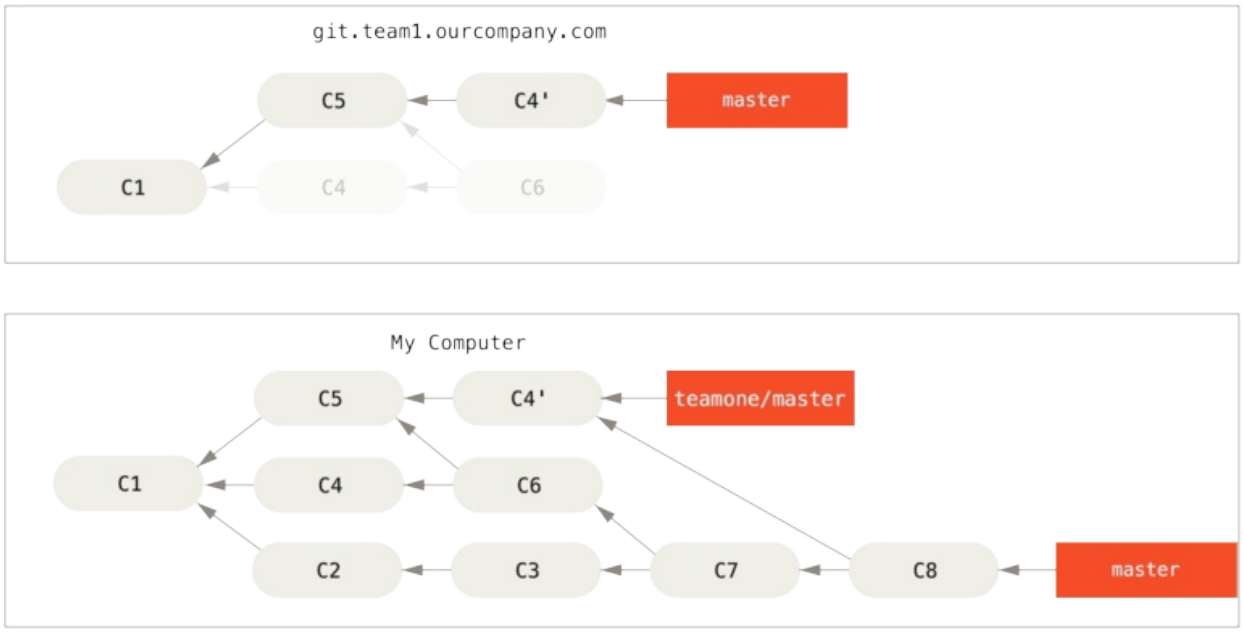


*Figure 48 : Vous fusionnez le même travail une nouvelle fois dans un nouveau _commit_ de fusion.*

Figure 47. Vous fusionnez le même travail une nouvelle fois dans un nouveau *commit* de fusion

Si vous lancez `git log` lorsque votre historique ressemble à ceci, vous verrez deux *commits* qui ont la même date d'auteur et les mêmes messages, ce qui est déroutant. De plus, si vous poussez cet historique sur le serveur, vous réintroduirez tous ces *commits* rebasés sur le serveur central, ce qui va encore plus dérouter les autres développeurs. C'est plutôt logique de présumer que l'autre développeur ne souhaite pas voir apparaître `C4` et `C6` dans l'historique. C'est la raison pour laquelle il avait effectué un rebasage initialement.

## Rebaser quand vous rebasez

Si vous vous retrouvez effectivement dans une situation telle que celle-ci, Git dispose d'autres fonctions magiques qui peuvent vous aider. Si quelqu'un de votre équipe pousse de force des changements qui écrasent des travaux sur lesquels vous vous êtes basés, votre défi est de déterminer ce qui est à vous et ce qui a été réécrit.

Il se trouve qu'en plus de l'empreinte SHA du *commit*, Git calcule aussi une empreinte qui est uniquement basée sur le patch introduit avec le commit. Ceci est appelé un "identifiant de patch" (*patch-id*).

Si vous tirez des travaux qui ont été réécrits et les rebasez au-dessus des nouveaux *commits* de votre collègue, Git peut souvent déterminer ceux qui sont uniquement les vôtres et les réappliquer au sommet de votre nouvelle branche.

Par exemple, dans le scénario précédent, si au lieu de fusionner quand nous étions à l'étape Quelqu'un pousse des *commits* rebasés, en abandonnant les *commits* sur lesquels vous avez fondé votre travail nous exécutons la commande `git rebase equipe1/master`, Git va :

- Déterminer quels travaux sont uniques à notre branche (C2, C3, C4, C6, C7)
- Déterminer ceux qui ne sont pas des *commits* de fusion (C2, C3, C4)
- Déterminer ceux qui n'ont pas été réécrits dans la branche de destination (uniquement C2 et C3 puisque C4 est le même *patch* que C4')
- Appliquer ces *commits* au sommet de `equipe1/master`

Ainsi, au lieu du résultat que nous avons observé au chapitre Vous fusionnez le même travail une nouvelle fois dans un nouveau *commit* de fusion, nous aurions pu finir avec quelque chose qui ressemblerait davantage à Rebaser au-dessus de travaux rebasés puis que l'on a poussé en forçant..

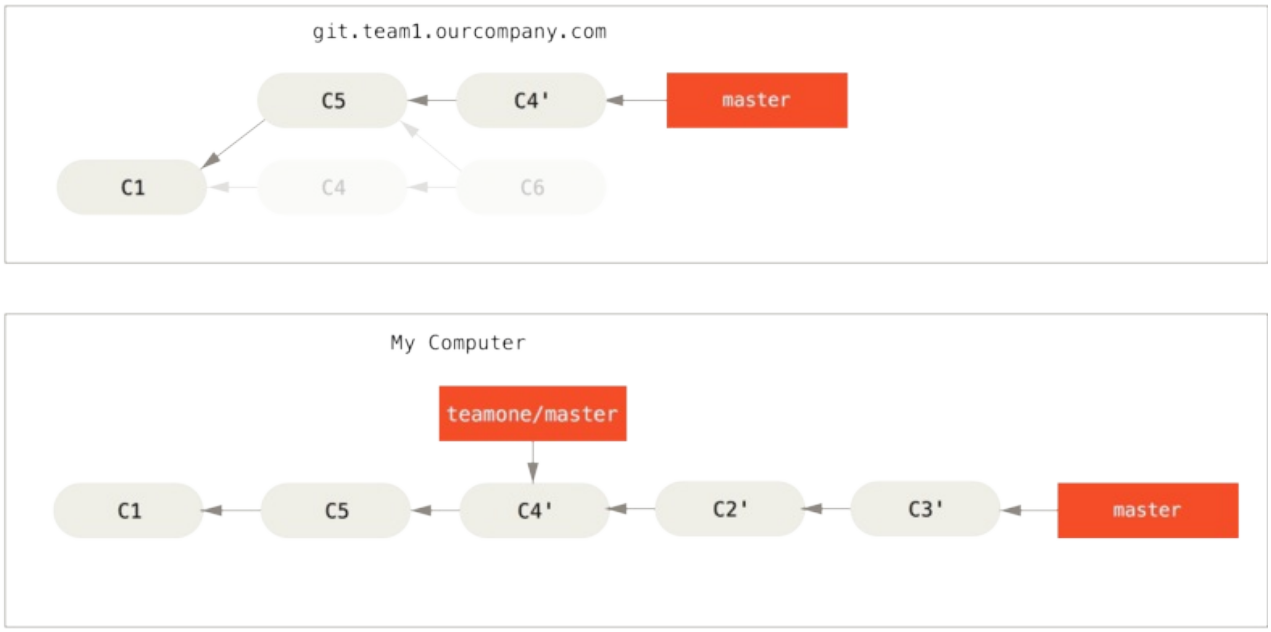


*Figure 49 : Rebaser au-dessus de travaux rebasés puis que l'on a poussé en forçant.*

Figure 48. Rebaser au-dessus de travaux rebasés puis que l'on a poussé en forçant.

Cela fonctionne seulement si les *commits* C4 et C4' de votre collègue correspondent presque exactement aux mêmes modifications. Autrement, le rebasage ne sera pas capable de déterminer qu'il s'agit d'un doublon et va ajouter un autre *patch* similaire à C4 (ce qui échouera probablement puisque les changements sont au moins partiellement déjà présents).

Vous pouvez également simplifier tout cela en lançant un `git pull --rebase` au lieu d'un `git pull` normal. Vous pouvez encore le faire manuellement à l'aide d'un `git fetch` suivi d'un `git rebase team1/master` dans le cas présent.

Si vous utilisez `git pull` et voulez faire de `--rebase` le traitement par défaut, vous pouvez changer la valeur du paramètre de configuration `pull.rebase` par `git config --global pull.rebase true`.

Si vous considérez le fait de rebaser comme un moyen de nettoyer et réarranger des *commits* avant de les pousser et si vous vous en tenez à ne rebaser que des *commits* qui n'ont jamais été publiés, tout ira bien. Si vous tentez de rebaser des *commits* déjà publiés sur lesquels les gens ont déjà basé leur travail, vous allez au devant de gros problèmes et votre équipe vous en tiendra rigueur.

Si vous ou l'un de vos collègues y trouve cependant une quelconque nécessité, assurez-vous que tout le monde sache lancer un `git pull --rebase` pour essayer de rendre les choses un peu plus faciles.

## Rebaser ou Fusionner

Maintenant que vous avez vu concrètement ce que signifient rebaser et fusionner, vous devez vous demander ce qu'il est préférable d'utiliser. Avant de pouvoir répondre à cela, revenons quelque peu en arrière et parlons un peu de ce que signifie un historique.

On peut voir l'historique des *commits* de votre dépôt comme un **enregistrement de ce qu'il s'est réellement passé**. Il s'agit d'un document historique qui a une valeur en tant que tel et ne doit pas être altéré. Sous cet angle, modifier l'historique des *commits* est presque blasphématoire puisque vous *mentez* sur ce qu'il s'est réellement passé. Dans ce cas, que faire dans le

cas d'une série de *commits* de fusions désordonnés ? Cela reflète ce qu'il s'est passé et le dépôt devrait le conserver pour la postérité.

Le point de vue inverse consiste à considérer que l'historique des *commits* est **le reflet de la façon dont votre projet a été construit**. Vous ne publieriez jamais le premier brouillon d'un livre et le manuel de maintenance de votre projet mérite une révision attentive. Ceci constitue le camp de ceux qui utilisent des outils tels que le rebasage et les branches filtrées pour raconter une histoire de la meilleure des manières pour les futurs lecteurs.

Désormais, nous espérons que vous comprenez qu'il n'est pas si simple de répondre à la question portant sur le meilleur outil entre fusion et rebasage. Git est un outil puissant et vous permet beaucoup de manipulations sur et avec votre historique mais chaque équipe et chaque projet sont différents. Maintenant que vous savez comment fonctionnent ces deux outils, c'est à vous de décider lequel correspond le mieux à votre situation en particulier.

De manière générale, la manière de profiter au mieux des deux mondes consiste à rebaser des modifications locales que vous avez effectuées mais qui n'ont pas encore été partagées avant de les pousser de manière à obtenir un historique propre mais sans jamais rebaser quoi que ce soit que vous ayez déjà poussé quelque part.

# Résumé

Nous avons traité les bases des branches et des fusions dans Git. Vous devriez désormais être à l'aise pour créer et basculer sur de nouvelles branches, basculer entre branches et fusionner des branches locales. Vous devriez aussi être capable de partager vos branches en les poussant sur un serveur partagé, de travailler avec d'autres personnes sur des branches partagées et de re-baser vos branches avant de les partager.