**GIT**(DVCS)

Contenu

[**I.** **POUR MIEUX CONNAITRE GIT.** 3](#_Toc323044663)

[a) Gestionnaire de version 3](#_Toc323044664)

[b) DVCS (Distributed Version Control Systems) 3](#_Toc323044666)

[c) Les avantages de Git 3](#_Toc323044668)

[**II.** **UTILISATION DE GIT** 5](#_Toc323044671)

[a) Enregistrer des modifications dans le dépôt 5](#_Toc323044672)

[b) Vérifier l’état des fichiers 5](#_Toc323044673)

[c) Placer de nouveaux fichiers sous suivi de version ou indexer des fichiers modifiés 5](#_Toc323044674)

[d) Ignorer des fichiers 5](#_Toc323044675)

[e) Inspecter les modifications indexées et non indexées 5](#_Toc323044676)

[f) Valider vos modifications 6](#_Toc323044677)

[g) Éliminer la phase d’indexation 6](#_Toc323044678)

[h) Effacer des fichiers 6](#_Toc323044679)

[i) Déplacer des fichiers 6](#_Toc323044680)

[j) Visualiser l’historique des validations 6](#_Toc323044681)

[k) Limiter la longueur de l’historique 8](#_Toc323044683)

[l) Utiliser une interface graphique pour visualiser l’historique 8](#_Toc323044684)

[**III. ANNULER DES ACTIONS** 9](#_Toc323044685)

[a) Modifier le dernier commit 9](#_Toc323044686)

[b) Désindexer un fichier déjà indexé 9](#_Toc323044687)

[c) Réinitialiser un fichier modifié 9](#_Toc323044688)

[**IV. TRAVAILLER AVEC DES DEPOTS DISTANTS** 10](#_Toc323044689)

[a) Afficher les dépôts distants 10](#_Toc323044690)

[b) Ajouter des dépôts distants 10](#_Toc323044691)

[c) Récupérer et tirer depuis des dépôts distants 10](#_Toc323044692)

[d) Pousser son travail sur un dépôt distant 10](#_Toc323044693)

[e) Inspecter un dépôt distant 10](#_Toc323044694)

[f) Retirer et déplacer des branches distantes 11](#_Toc323044695)

[**V. BALISAGE** 12](#_Toc323044696)

[a) Lister vos balises 12](#_Toc323044697)

[b) Créer des balises 12](#_Toc323044698)

[c) Vérifier des balises 12](#_Toc323044699)

[d) Baliser après coup 12](#_Toc323044700)

[e) Partager les balises 13](#_Toc323044701)

[**VI.** **LES BRANCHES AVEC GIT** 14](#_Toc323044702)

[a) Ce qu’est une branche 14](#_Toc323044703)

[b) Brancher et fusionner : les bases 15](#_Toc323044704)

[c) Les bases de la fusion 16](#_Toc323044705)

[d) Conflits de fusion 17](#_Toc323044706)

[e) Gestion de branches 18](#_Toc323044708)

[VII. TRAVAILLER AVEC LES BRANCHES 19](#_Toc323044709)

[**Branches au long cours** 19](#_Toc323044710)

1. **POUR MIEUX CONNAITRE GIT.**
2. Gestionnaire de version

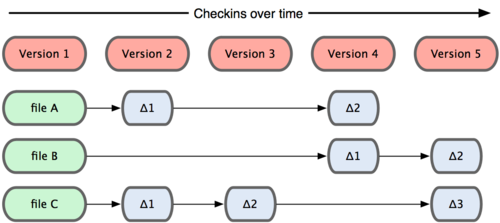
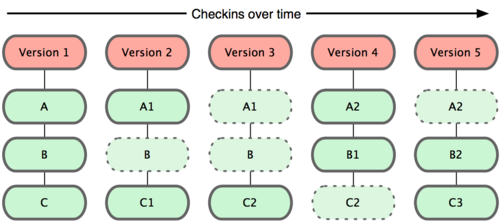
Un gestionnaire de gestion enregistre l’évolution d’un fichier ou d’un ensemble de fichiers au cours du temps de manière à ce qu’on puisse rappeler une version antérieure d’un fichier à tout moment.

1. DVCS (Distributed Version Control Systems)

Le dépôt est complètement dupliqué. 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.

1. Les avantages de Git

À chaque fois que vous commitez 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 qui n’a pas été modifié.



Système de gestion de données des autres VCS Système de gestion de données de Git

La plupart des opérations de **Git ne nécessite que des fichiers et ressources locales** les opérations semblent donc instantanées et donc il y a très peu de choses que vous ne puissiez réaliser si vous n’êtes pas connecté.

Si vous n’êtes pas connecté vous pouvez continuer à gérer vos versions sans soucis en attendant de pouvoir de nouveau vous connecter pour partager votre travail.

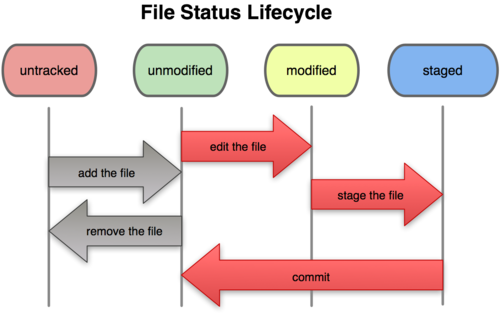
**Git gère l’intégrité** : Dans Git, tout est vérifié par une somme de contrôle avant d’être stocké et par la suite cette somme de contrôle, signature unique, sert de référence. Cela signifie qu’il est impossible de modifier le contenu d’un fichier ou d’un répertoire sans que Git ne s’en aperçoive.

**Git ne fait qu’ajouter des données :** 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. il est très difficile de perdre des données, spécialement si en plus vous synchronisez votre base de données locale avec un dépôt distant.

Les trois états : Git gère trois états dans lequel les fichiers peuvent résider : commité, modifié et indexé. Commité 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 commité 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.

1. **UTILISATION DE GIT ( en ligne de commande )**

## Enregistrer des modifications dans le dépôt

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 appartenait déjà au dernier instantané ; ils peuvent être inchangés, modifiés ou indexés. Tous les autres fichiers sont non suivis. Si vous créer un nouveau fichier Git ne le placera sous suivi en 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.

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.

## Vérifier l’état des fichiers

L’outil principal pour déterminer quels fichiers sont dans quel état est la commande git status.

## Placer de nouveaux fichiers sous suivi de version ou indexer des fichiers modifiés

Utiliser la commande git add Monfichier*. Monfichier* est indexé car il apparaît dans la section “Changes to be committed” (Modifications à enregistrer).Si vous enregistrez à ce moment, la version du fichier à l’instant où vous lancez git add est celle qui appartiendra à l’instantané. (Cette commande s’applique aussi bien sur les répertoires, dans ce cas elle ajoute récursivement tous les fichiers.)

## Ignorer des fichiers

Spécifier des patrons dans le fichier .gitignore.

## Inspecter les modifications indexées et non indexées

La commande git diff. Elle sera utilisée le plus souvent pour répondre aux questions suivantes : qu’est-ce qui a été modifié mais pas encore indexé ? Quelle modifications 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 diff --cached pour visualiser ce qui a déjà été indexé dans le fichier.

## Valider vos modifications

Souvenez-vous que tout ce qui encore non indexé tous les fichiers qui ont été créés ou modifiés mais n’ont pas subi degit add depuis ne feront pas partie de la prochaine validation.

La commande git commit permet de valider les modifications,  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.

## Éliminer la phase d’indexation

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 .

## 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. Pour  abandonner le suivi de version d’un fichier tout en le conservant dans la copie de travail la commande est git rm --cached monfichier.txt

1. Déplacer 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. 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 cela fonctionne Git gère le renommage des fichiers.

## Visualiser l’historique des validations

La commande git log est l’outil le plus basique et puissant pour cet objet.

Quelques options  pour la commande git log:

|  |  |
| --- | --- |
| Options | Explications |
| *-p* | montre les différences introduites entre chaque validation. Affiche le patch appliqué par chaque commit. |
| *-2* | limite la sortie de la commande aux deux entrées les plus récentes |
| *--stat* | pour visualiser des statistiques résumées pour chaque commit |
| *--pretty* | pour modifier le journal vers un format différent |
| *oneline* | affiche chaque commit sur une seule ligne |
| *short, full*,*fuller* | montre le résultat à peu de choses près dans le même format mais avec de plus en plus d’information |
| *--graph* | Cette option ajoute un joli graphe en caractères ASCII pour décrire l’historique des branches et fusions |
| *--shortstat* | N'affiche que les lignes 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 |

Liste des options de formatage pour l’option --pretty les plus utiles :

|  |  |
| --- | --- |
| options | explications |
| %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 |

1. Limiter la longueur de l’historique

Voici quelques options permettant de limiter la longueur du résultat de git log :

|  |  |
| --- | --- |
| options | explications |
| --grep | permet de chercher des mots clés dans les messages de validation |
| -(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 |

1. Utiliser une interface graphique pour visualiser l’historique

Gitk est un outil graphique mimant les fonctionnalités de git log*.* Il suffit de taper gitk en ligne de commande pour y avoir accès.

# ANNULER DES ACTIONS

1. Modifier le dernier commit

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.

## Désindexer un fichier déjà indexé

git reset HEAD <fichier> Le fichier est modifié mais de retour à l’état non indexé.

## Réinitialiser un fichier modifié

git checkout -- <fichier> Pour réinitialiser le fichier et le ramener à l’état qu’il avait dans le dernier instantané (ou lors du clonage, ou dans l’état dans lequel vous l’avez obtenu dans votre copie de travail).

**/ ! \ C’est une commande dangereuse** : toutes les modifications que vous auriez réalisées sur ce fichier ont disparu

# TRAVAILLER AVEC DES DEPOTS DISTANTS

1. Afficher les dépôts distants

git remote. Elle liste les noms des différentes étiquettes distantes que vous avez spécifiées.

-v, qui vous montre l’URL que Git a stocké pour nom court à étendre.

## Ajouter des dépôts distants

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 git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

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 ne souhaitez pas l’avoir encore dans votre branche, vous pouvez lancer git fetch pb. 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.

1. Récupérer et tirer depuis des dépôts distants

git fetch [nom-distant] 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.

1. 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].

## 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].

## Retirer et déplacer des branches distantes

Si vous souhaitez renommer une référence, dans les versions récentes de Git, vous pouvez lancer git remote rename pour modifier le nom court d’un dépôt distant. 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 une référence pour certaines raisons vous pouvez utiliser git remote rm .

# BALISAGE

Git donne la possibilité de baliser 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).

1. Lister vos balises

Lister les balises existantes dans Git est très simple. Tapez juste git tag. Cette commande liste les balises dans l’ordre alphabétique. L’ordre dans lequel elles apparaissent n’a aucun rapport avec l’historique.

1. Créer des balises

Git utilise deux types principaux de balises : **légères et annotées**.

Une **balise légère** ressemble beaucoup à une branche qui ne change pas, c’est juste un pointeur sur un commit spécifique. Celles-ci se réduisent à stocker la somme de contrôle d’un commit dans un fichier.

**Les balises 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 de balisage et peuvent être signées et vérifiées avec GNU Privacy Guard (GPG). Il est généralement recommandé de créer des balises annotées pour générer toute cette information mais si la balise doit rester temporaire ou l’information supplémentaire n’est pas désirée, il reste toujours les balises légères.

**Les balises annotées :** Créer des balises annotées est simple avec Git. Le plus simple est de spécifier l’option -a à la commande tag. L’option -m permet de spécifier le message de balisage qui sera stocké avec la balise. Si vous ne spécifiez pas de message en ligne pour une balise annotée, Git lance votre éditeur pour pouvoir le saisir. Vous pouvez visualiser les données de la balise à côté du commit qui a été marqué en utilisant la commande git show.

**Les balises signées :** Vous pouvez aussi signer vos balises avec GPG, à condition d’avoir une clé privée. Il suffit de spécifier l’option -s au lieu de -a.

**Les balises légères :** Pour créer une balise légère, il suffit de n’utiliser aucune des options -a, -sou -m à la commande tag.

1. Vérifier des balises

Pour vérifier une balise signée, il faut utiliser git tag -v [nom-de-balise].

1. Baliser après coup

Vous pouvez aussi baliser des commits plus anciens. Pour cela il suffit de spécifier la somme de contrôle du commit (ou une partie) en fin de commande : git tag -a v1.2 9fceb02.

1. Partager les balises

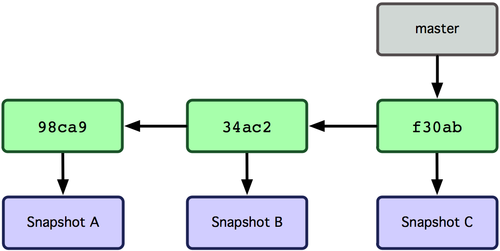
Par défaut, la commande git push ne transfère pas les balises vers les serveurs distants. Il faut explicitement pousser les balises après les avoir créées localement. Ce processus s’apparente à pousser des branches distantes, vous pouvez lancer gitpush origin [nom-du-tag]. Si vous avez de nombreuses balises que vous souhaitez pousser en une fois, vous pouvez aussi utiliser l’option --tags avec la commande git push.

1. **LES BRANCHES AVEC GIT**

Faire une branche signifie diverger de la ligne principale de développement et continuer à travailler sans se préoccuper de cette ligne principale.  La méthode de Git pour gérer les branches est particulièrement légère, permettant de réaliser des embranchements quasi instantanément et de basculer de branche généralement aussi rapidement.

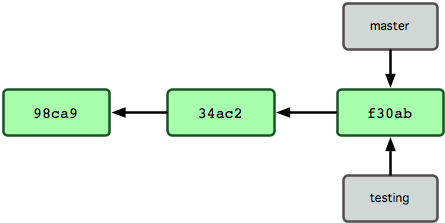
## Ce qu’est une branche

Lorsqu’on valide dans Git, Git stocke un objet commit qui contient un pointeur vers l’instantané du contenu qui a été indexé, les méta-données d’auteur et de message et zéro ou plusieurs pointeurs vers le ou les commits qui sont les parents directs de ce commit : zéro parent pour la première validation, un parent pour un commit normal et des parents multiples pour des commits qui sont le résultat de la fusion d’une ou plusieurs branches.



La branche master avance automatiquement au rythme des validations.

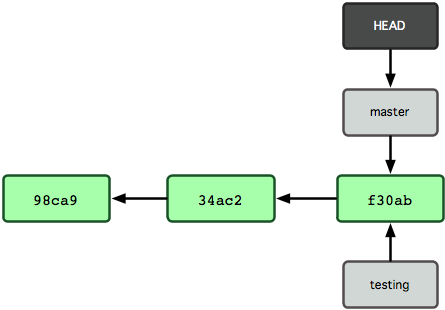
Une branche dans Git est tout simplement un pointeur mobile léger vers un de ces objets commit. La branche par défaut dans Git s’appelle master. Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés. À chaque validation, le pointeur de la branche master avance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

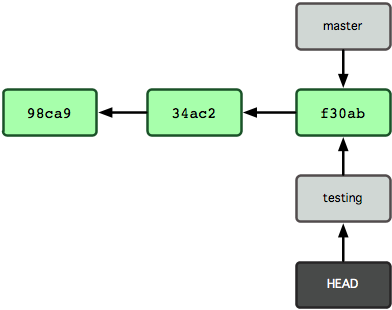


Que se passe-t-il si vous créez une nouvelle branche ? Et bien, cela crée un nouveau pointeur à déplacer. Supposons que vous créez une nouvelle branche nommée testing. Vous utilisez la commande *git branch* *<nomdelabranche>:*

Une nouvelle branche est créée

Cela crée un nouveau pointeur vers le commit actuel.



Comment Git connaît-il la branche sur laquelle vous vous trouvez ? Il conserve un pointeur spécial appelé HEAD. Dans Git, c’est un pointeur sur la branche locale où vous vous trouvez. Dans notre cas, vous vous trouvez toujours sur master. 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.

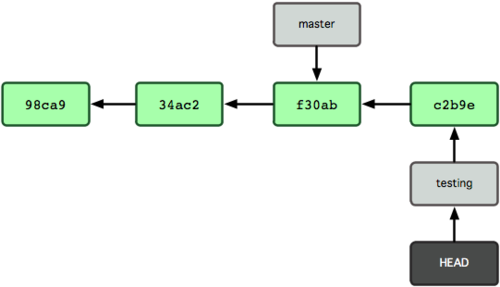
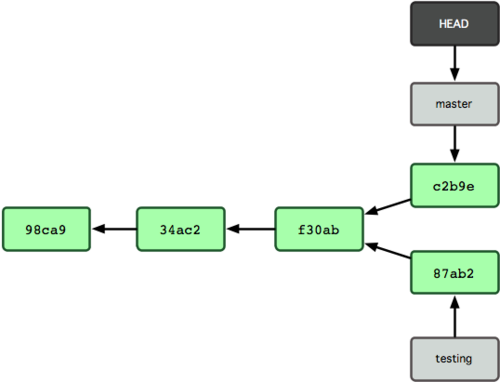
Un pointeur head indique sur quelle branche nous travaillons

Pour basculer vers une branche existante, il suffit de lancer la commande git checkout <nomdelabranche>. Basculons vers la nouvelle branche testing .

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

Qu’est-ce que cela signifie ? Et bien, faisons une autre validation.

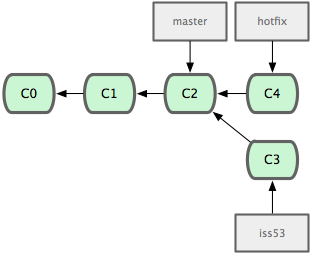
Pour pouvoir travailler sur la branche testing in bascule le pointeur head dessus

C’est intéressant parce qu’à présent, votre branche testing a avancé, tandis que la branche master pointe toujours sur le commit sur lequel vous étiez lorsque vous avez lancé git checkout pour basculer de branche. Retournons sur la branche master. Cette commande va réaliser deux actions. Elle a remis le pointeur HEAD sur la branche master et elle a replacé les fichiers de la copie de travail dans l’état pointé par master. Cela signifie aussi que les modifications que vous réalisez à partir de maintenant divergeront de l’ancienne version du projet. Cette commande retire les modifications réalisées dans la branche testing pour vous permettre de repartir dans une autre direction de développement.

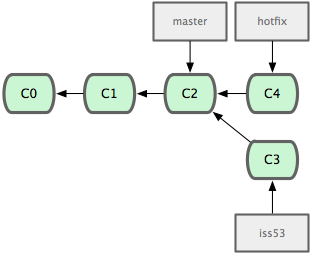
On créer cette fois une nouvelle validation sur la branche master. Les deux branches divergent.

On créée une nouvelle validation qui n'appartiendra qu'à la branche testing

## Brancher et fusionner : les bases

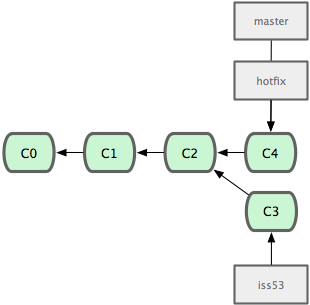
*git checkout* avec l’option -*b* : créer une branche et y basculer tout de suite.

Imaginons que nous travaillons sur un développement web. Nous avons déjà réalisé plusieurs commit. Nous souhaitons maintenant développer une branche pour travailler sur le problème 53. Nous plaçons le pointeur head sur cette nouvelle branche et faisons une nouvelle validation. Cependant nous souhaitons maintenant travailler sur un autre problème. Avec Git, vous n’avez pas besoin de déployer les modifications déjà  validée pour prob53 avec les correctifs du problème et vous n’avez pas non plus à suer pour éliminer ces modifications avant de pouvoir appliquer les correctifs du problème en production. Tout ce que vous avez à faire, c’est simplement rebasculer sur la branche master.

Remarque : Avant de le faire, notez que si votre copie de travail contient des modifications non validées qui sont en conflit avec la branche que vous extrayez, Git ne vous laissera pas basculer de branche. Le mieux est d’avoir votre copie de travail dans un état propre au moment de basculer de branche. Lorsque vous basculez, Git réinitialise le répertoire de travail pour qu’il ressemble à  l’instantané de la validation sur laquelle la branche que vous extrayez pointe. Il ajoute, retire et modifie les fichiers automatiquement.

Création d'une nouvelle branche

Vous avez un correctif à  faire. Créons une branche de correctif sur laquelle travailler jusqu’à  ce que ce soit terminé.

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*.

Comme le commit pointé par la branche que vous avez fusionné était directement descendant du commit sur lequel vous vous trouvez, Git a avancé le pointeur en avant. Autrement dit, lorsque l’on cherche à  fusionner un commit qui peut être joint en suivant l’historique depuis le commit d’origine, Git avance simplement le pointeur car il n’y a pas de travaux divergeant à  réellement fusionner ceci s’appelle **l’avance rapide.**

Fusion de la branche hotfix avec la branche master : avance rapide

## Les bases de la fusion

Supposons que vous ayez décidé que le travail sur le problème #53 est terminé et se trouve donc prêt à  être fusionné dans la branche master. Pour ce faire il suffit de basculer sur la branche master puis de fusionner la branche prob53 a l’aide d’un git merge.

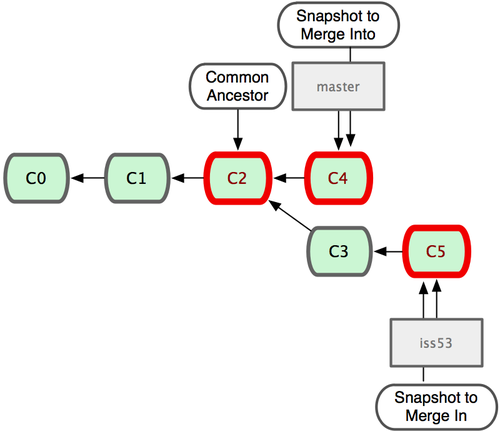
$ git checkout master

$ git merge prob53

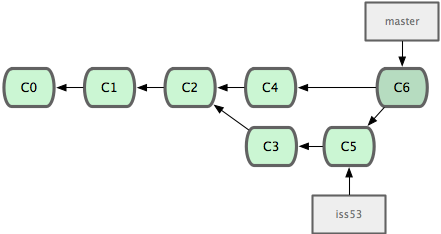
Merge made by recursive.

README | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)



L’historique de développement a divergé à un certain point. 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 réalise une simple fusion à  trois sources, en utilisant les deux instantanés pointés par les sommets des branches et l’ancêtre commun des deux.



Au lieu de simplement d’avancer le pointeur de branche, Git crée un nouvel instantané qui résulte de la fusion à  trois branches et crée automatiquement un nouveau commit qui pointe dessus. On appelle ceci un commit de fusion, qui est spécial en ce qu’il comporte plus d’un parent.

À présent que votre travail a été fusionné, vous n’avez plus besoin de la branche prob53. Vous pouvez l’effacer et fermer manuellement le ticket à l’aide de la commande git branch -d prob5*.*

## Conflits de fusion

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. Git n’a pas automatiquement créé le commit de fusion. Il a arrêté le processus le temps que vous résolviez le conflit.  Git ajoute des marques de conflit standard 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>

>>>>>>> prob53: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 la branche prob53 se trouve en dessous. Pour résoudre le conflit, vous devez choisir une partie ou l’autre ou bien fusionner leurs contenus par vous-même. Après avoir résolu chacune de ces sections dans chaque fichier comportant un conflit, lancez git add sur chaque fichier pour le marquer comme résolu.

Remarque : Si vous souhaitez utiliser un outil graphique pour résoudre ces problèmes, vous pouvez lancer *git mergetool* qui démarre l’outil graphique de fusion approprié et vous permet de naviguer dans les conflits. 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 indexe le fichier pour le marquer comme résolu.

Cela vous convient et vous avez vérifié que tout ce qui comportait un conflit a été indexé, vous pouvez taper la commande *git commit* pour finaliser le commit de fusion.

## Gestion de branches

La commande git branch fait plus que créer et effacer des branches. Si vous la lancez sans argument, vous obtenez la liste des branches courantes :

$ git branch

prob53

\* master

testing

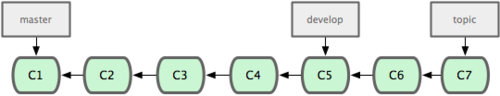
Notez le caractère \* qui préfixe la branche master. Ce caractère indique la branche qui a été extraite. Ceci signifie que si vous validez des modifications, la branche master avancera avec votre travail. Pour visualiser les dernières validations sur chaque branche, vous pouvez lancer le commande git branch –v.

branch --merged pour voir quelles branches ont déjà été fusionnées dans votre branche actuelle.

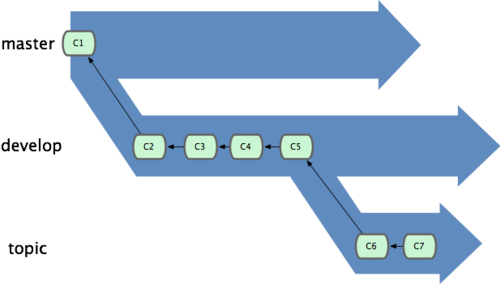
branch --no-merged pour visualiser les branches qui contiennent des travaux qui n’ont pas encore été fusionnés.

1. **TRAVAILLER AVEC LES BRANCHES**
2. La méthode « Branches au long cours »

Comme Git utilise une **fusion à 3 branches**, fusionner une branche dans une autre plusieurs fois sur une longue période est généralement facile. Cela signifie que vous pouvez travailler sur plusieurs branches ouvertes en permanence pendant plusieurs étapes de votre cycle de développement.

De nombreux développeurs utilisent Git avec une méthode qui utilise cette approche, telle que n’**avoir que du code entièrement stable et testé dans la branche master**.

Représentation de la méthode « Branches au long cours »

Ils ont **une autre branche en parallèle appelée develop qui, lorsqu’elle devient stable, peut être fusionnée dans master.** Cette branche est utilisée pour tirer des branches spécifiques (**branches avec une faible durée de vie**, utilisées par exemple pour résoudre des problèmes identifiés).

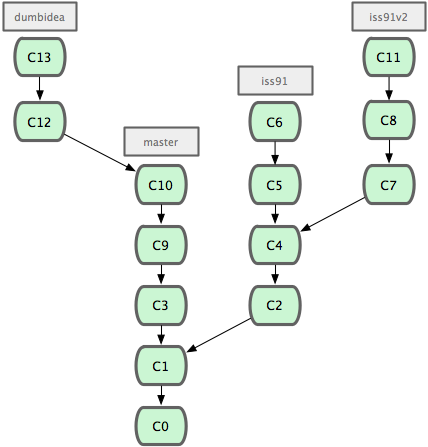
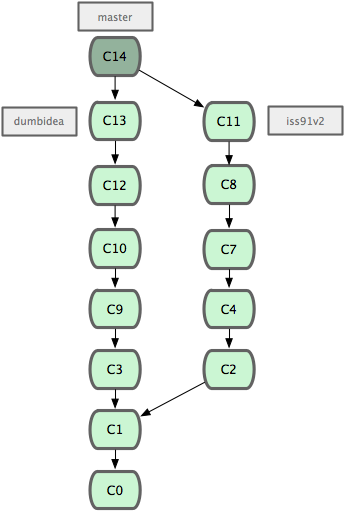
Méthode « Branches au long cours » avec les branches vues comme des silos

En réalité, nous parlons de pointeurs qui se déplacent le long des lignes des commits réalisés. **Les branches stables sont plus en profondeur dans la ligne de l’historique des commits tandis que les branches des derniers développements sont plus en hauteur dans l’historique.**

Vous pouvez reproduire ce schéma sur plusieurs niveaux de stabilité.

1. La méthode « Les branches de sujet »

Les branches de sujet sont tout de même utiles quelle que soit la taille du projet. Une branche de sujet est **une branche de courte durée de vie créée et utilisée pour une fonctionnalité ou une tâche particulière**. Cette technique vous permet de basculer de contexte complètement et immédiatement. Il est beaucoup plus simple de réaliser des revues de code parce que votre travail est isolé dans des silos où toutes les modifications sont liées au sujet. Vous pouvez entreposer vos modifications ici 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 de développées.



Même projet après avoir fusionné la branche dumbidea à la branche master et après avoir la solution v2 au problème 91

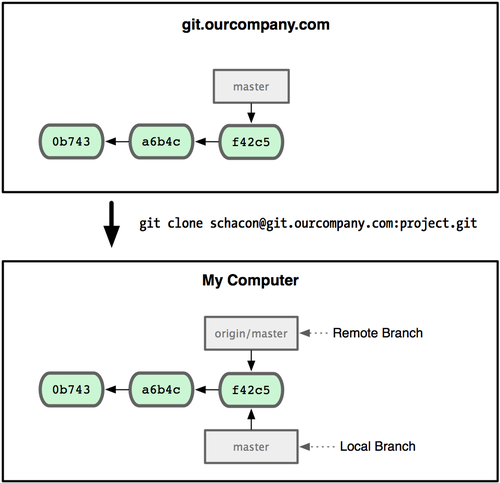
Multitude de branches de sujet.

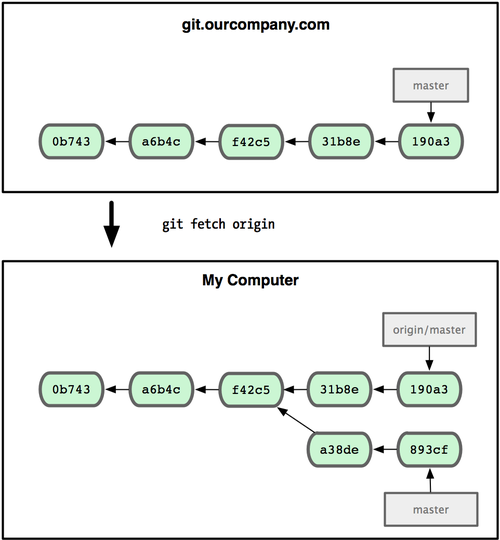
## Les branches distantes

Les branches distantes sont des références à l’état des branches sur votre dépôt distant. Ce sont **des branches locales qu’on ne peut pas modifier**, elles sont modifiées automatiquement lors de communications réseau.

**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 suffit de vérifier la branche origin/master.

**Remarque :** Si vous étiez en train de travailler avec un collègue et qu’il a mis à jour la brancheprob53, vous pourriez avoir votre propre branche prob53 ; mais la branche sur le serveur pointerait sur le commit de origin/prob53.

Supposons que vous avez un serveur Git sur le réseau à l’adresse git.notresociete.com. Si vous clonez à partir de ce serveur, Git le nomme automatiquement origin et en tire tout l’historique, crée un pointeur sur l’état actuel de la branche master et l’appelle localement origin/master ; vous ne pouvez pas la modifier. Git vous crée votre propre branche master qui démarre au même commit que la branche master d’origine, pour que vous puissiez commencer à travailler.

Si vous travaillez sur votre branche locale master et que dans le même temps, quelqu’un pousse versgit.notresociete.com et met à jour cette branche, alors vos deux historiques divergent. Tant que vous restez sans contact avec votre serveur distant, votre pointeur origin/master n’avance pas.Lancez la commande git fetch origin pour synchroniser votre travail.

## Pousser vers un serveur

Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur le serveur distant sur lequel vous avez accès en écriture. Lancez git push.

**Important :** lorsque l’on récupère une nouvelle branche depuis un serveur distant, il n’y a pas de création automatique d’une copie locale éditable.  En d’autres termes, il n’y a pas de branche, seulement un pointeur sur la branche qui n’est pas modifiable.

Pour fusionner ce travail dans votre branche actuelle de travail, vous pouvez lancer git merge [serveur distant]/[branche].

## 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. 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, git pull sur une de ces branches récupère toutes les références distantes et les fusionne automatiquement dans la branche distante correspondante.

Vous pouvez créer des branches de suivi si vous le souhaitez en lançant git checkout -b [branche] [nomdistant]/[branche].

[branche] = nom de la branche de suivie.

[nomdistant]/[branche] = branche qui sera suivie.

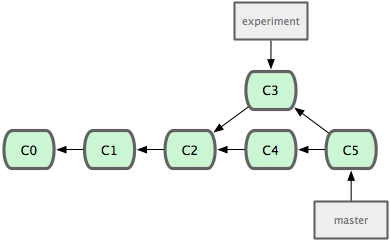
### Effacer des branches distantes

## Vous pouvez effacer une branche distante en utilisant la syntaxe plutôt obtuse git push [nomdistant] :[branch].

## Rebaser

Dans Git, il y a deux façons d’intégrer les modifications d’une branche dans une autre : en fusionnant merge et en rebasant rebase.

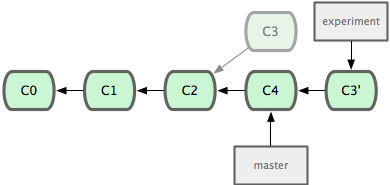
Le moyen le plus simple pour intégrer ensemble des branches est la fusion via la commande merge.



Exemple de fusion avec la commande merge.

Il existe un autre moyen : vous pouvez prendre le patch de la modification introduite en C3 et le réappliquer sur C4. Dans Git, cette action est appelée **rebaser**. Avec la commande rebase, vous prenez toutes les modifications qui ont été validées sur une branche et vous les rejouez sur une autre.

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 entre chaque validation de la branche sur laquelle vous êtes, en les sauvant dans des fichiers temporaires, en basculant sur la branche destination et en réappliquant chaque modification dans le même ordre.



Exemple de rebasage

À présent, l’instantané pointé par C3’ 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. À ce moment, vous pouvez retourner sur la branche master et réaliser une fusion en avance rapide.