

 eBook Gratuit

# APPRENEZ

---

# Git

eBook gratuit non affilié créé à partir des  
**contributeurs de Stack Overflow.**

#git

# Table des matières

|   |           |
|---|-----------|
| À propos.....   | 1         |
| <b>Chapitre 1: Démarrer avec Git.....</b>   | <b>2</b>  |
| Remarques.....  | 2         |
| Versions.....   | 2         |
| Exemples.....   | 4         |
| Créez votre premier référentiel, puis ajoutez et validez des fichiers.....            | 4         |
| Cloner un référentiel.....  | 5         |
| Configuration de la télécommande en amont.....  | 6         |
| Code de partage.....  | 7         |
| Définition de votre nom d'utilisateur et de votre adresse e-mail.....                 | 7         |
| Apprendre une commande.....   | 8         |
| Mettre en place SSH pour Git.....   | 9         |
| Installation de Git.....  | 10        |
| <b>Chapitre 2: Afficher l'historique des validations graphiquement avec Gitk.....</b> | <b>13</b> |
| Exemples.....   | 13        |
| Afficher l'historique des validations pour un fichier.....                            | 13        |
| Afficher tous les commits entre deux commits.....                                     | 13        |
| Affichage validé depuis la balise de version.....                                     | 13        |
| <b>Chapitre 3: Alias.....</b>   | <b>14</b> |
| Exemples.....   | 14        |
| Alias simples.....  | 14        |
| List / search aliases existants.....  | 14        |
| Recherche d'alias.....  | 14        |
| Alias avancés.....  | 15        |
| Ignorer temporairement les fichiers suivis.....                                       | 15        |
| Afficher un joli journal avec un graphique de branche.....                            | 16        |
| Mise à jour du code tout en conservant un historique linéaire.....                    | 17        |
| Voir quels fichiers sont ignorés par votre configuration .gitignore.....              | 17        |
| Dégager les fichiers mis en scène.....  | 17        |
| <b>Chapitre 4: Analyse des types de workflows.....</b>                                | <b>19</b> |

|   |           |
|---|-----------|
| Remarques.....  | 19        |
| Exemples.....   | 19        |
| Workflow Gitflow.....   | 19        |
| Flux de travail.....  | 21        |
| Workflow centralisé.....  | 21        |
| Workflow de la fonction Feature.....  | 23        |
| GitHub Flow.....  | 23        |
| <b>Chapitre 5: arbre diff.....</b>  | <b>25</b> |
| Introduction.....   | 25        |
| Exemples.....   | 25        |
| Voir les fichiers modifiés dans un commit spécifique.....                                       | 25        |
| Usage.....  | 25        |
| Options diff communes.....  | 25        |
| <b>Chapitre 6: Archiver.....</b>  | <b>27</b> |
| Syntaxe.....  | 27        |
| Paramètres.....   | 27        |
| Exemples.....   | 28        |
| Créer une archive du dépôt git avec le préfixe de répertoire.....                               | 28        |
| Créer une archive du dépôt git en fonction d'une branche, d'une révision, d'un tag ou d'un..... | 28        |
| Créer une archive du dépôt git.....   | 28        |
| <b>Chapitre 7: Bisecting / Finding défectueux commits.....</b>                                  | <b>30</b> |
| Syntaxe.....  | 30        |
| Exemples.....   | 30        |
| Recherche binaire (git bisect).....   | 30        |
| Trouver automatiquement un commit défectueux.....   | 31        |
| <b>Chapitre 8: Blâmer.....</b>  | <b>33</b> |
| Syntaxe.....  | 33        |
| Paramètres.....   | 33        |
| Remarques.....  | 34        |
| Exemples.....   | 34        |
| Afficher le commit qui a modifié en dernier une ligne.....                                      | 34        |
| Ignorer les modifications des espaces uniquement.....   | 34        |

|  |           |
|--|-----------|
| Afficher uniquement certaines lignes .....           | 34        |
| Pour savoir qui a changé un fichier .....            | 34        |
| <b>Chapitre 9: Changer le nom du dépôt git .....</b> | <b>36</b> |
| Introduction .....                                   | 36        |
| Exemples .....                                       | 36        |
| Changer le paramètre local .....                     | 36        |
| <b>Chapitre 10: Configuration .....</b>              | <b>37</b> |
| Syntaxe .....  | 37        |
| Paramètres .....                                     | 37        |
| Exemples .....                                       | 37        |
| Nom d'utilisateur et adresse e-mail .....            | 37        |
| Plusieurs configurations git .....                   | 37        |
| Définition de l'éditeur à utiliser .....             | 38        |
| Configuration des fins de ligne .....                | 39        |
| La description .....                                 | 39        |
| Microsoft Windows .....                              | 39        |
| Basé sur Unix (Linux / OSX) .....                    | 39        |
| configuration pour une seule commande .....          | 39        |
| Configurer un proxy .....                            | 40        |
| Correction automatique des fautes de frappe .....    | 40        |
| Lister et éditer la configuration actuelle .....     | 40        |
| Nom d'utilisateur et adresse email multiples .....   | 41        |
| Exemple pour Windows: .....                          | 41        |
| .gitconfig .....                                     | 41        |
| .gitconfig-travail.config .....                      | 41        |
| .gitconfig-opensource.config .....                   | 41        |
| Exemple pour Linux .....                             | 42        |
| <b>Chapitre 11: Crochets .....</b>                   | <b>43</b> |
| Syntaxe .....  | 43        |
| Remarques .....                                      | 43        |
| Exemples .....                                       | 43        |
| Commit-msg .....                                     | 43        |

|  |           |
|--|-----------|
| Crochets locaux.....   | 43        |
| Post-caisse.....   | 44        |
| Post-commit.....   | 44        |
| Post-recevoir.....   | 44        |
| Pré-engagement.....  | 45        |
| Prepare-commit-msg.....  | 45        |
| Pré-rebase.....  | 45        |
| Pré-recevoir.....  | 45        |
| Mettre à jour.....   | 46        |
| Pré-pousser.....   | 46        |
| Vérifier la construction Maven (ou un autre système de construction) avant de valider..... | 48        |
| Transférer automatiquement certaines poussées vers d'autres référentiels.....              | 48        |
| <b>Chapitre 12: Crochets côté client Git.....</b>  | <b>49</b> |
| Introduction.....  | 49        |
| Exemples.....  | 49        |
| Installation d'un crochet.....   | 49        |
| Git pre-push hook.....   | 49        |
| <b>Chapitre 13: Cueillette De Cerises.....</b>   | <b>51</b> |
| Introduction.....  | 51        |
| Syntaxe.....   | 51        |
| Paramètres.....  | 51        |
| Exemples.....  | 51        |
| Copier un commit d'une branche à une autre.....  | 51        |
| Copier une gamme d'engagement d'une branche à l'autre.....                                 | 52        |
| Vérifier si un choix de cerises est requis.....  | 52        |
| Trouver des commits à appliquer à l'amont.....   | 52        |
| <b>Chapitre 14: Des sous-modules.....</b>  | <b>54</b> |
| Exemples.....  | 54        |
| Ajouter un sous-module.....  | 54        |
| Cloner un dépôt Git avec des sous-modules.....   | 54        |
| Mise à jour d'un sous-module.....  | 54        |
| Définir un sous-module pour suivre une branche.....  | 55        |

|   |           |
|---|-----------|
| Retrait d'un sous-module.....   | 55        |
| Déplacement d'un sous-module.....   | 56        |
| <b>Chapitre 15: Écrasement.....</b>                                       | <b>58</b> |
| Remarques.....  | 58        |
| Qu'est-ce qui écrase?.....  | 58        |
| Branches de squash et à distance.....                                     | 58        |
| Exemples.....   | 58        |
| Squash Recent Commits sans relancer.....                                  | 58        |
| Squashing Commits pendant une rebase.....                                 | 58        |
| Autosquash: Valider le code que vous voulez écraser lors d'un rebase..... | 59        |
| Squashing Commit Au cours de la fusion.....                               | 60        |
| Autosquashing et corrections.....   | 60        |
| <b>Chapitre 16: En poussant.....</b>                                      | <b>62</b> |
| Introduction.....   | 62        |
| Syntaxe.....  | 62        |
| Paramètres.....   | 62        |
| Remarques.....  | 62        |
| Amont Aval.....   | 62        |
| Exemples.....   | 62        |
| Pousser.....  | 62        |
| <b>Spécifiez le référentiel distant.....</b>                              | <b>63</b> |
| <b>Spécifier la branche.....</b>  | <b>63</b> |
| <b>Définir la branche de suivi à distance.....</b>                        | <b>63</b> |
| <b>Pousser vers un nouveau référentiel.....</b>                           | <b>63</b> |
| <b>Explication.....</b>   | <b>63</b> |
| Force de poussée.....   | 64        |
| <b>Notes IMPORTANTES.....</b>   | <b>64</b> |
| Poussez un objet spécifique vers une branche distante.....                | 64        |
| <b>Syntaxe générale.....</b>  | <b>64</b> |
| Exemple.....  | 64        |
| <b>Supprimer une branche distante.....</b>                                | <b>64</b> |

|  |           |
|--|-----------|
| Exemple.....   | 64        |
| Exemple.....   | 65        |
| <b>Poussez un seul engagement.....</b>   | <b>65</b> |
| Exemple.....   | 65        |
| Modification du comportement de push par défaut.....   | 65        |
| Balises Push.....  | 66        |
| <b>Chapitre 17: Fichier .mailmap: Associant contributeur et alias de messagerie.....</b>       | <b>67</b> |
| Syntaxe.....   | 67        |
| Remarques.....   | 67        |
| Exemples.....  | 67        |
| Fusionner les contributeurs par alias pour afficher le nombre de validations dans le jour..... | 67        |
| <b>Chapitre 18: Fusion.....</b>  | <b>69</b> |
| Syntaxe.....   | 69        |
| Paramètres.....  | 69        |
| Exemples.....  | 69        |
| Fusionner une branche dans une autre.....  | 69        |
| Fusion automatique.....  | 69        |
| Abandonner une fusion.....   | 70        |
| Garder les changements d'un seul côté d'une fusion.....  | 70        |
| Fusionner avec un commit.....  | 70        |
| Recherche de toutes les branches sans modifications fusionnées.....                            | 70        |
| <b>Chapitre 19: Fusion externe et difftools.....</b>   | <b>71</b> |
| Exemples.....  | 71        |
| Mise en place au-delà de la comparaison.....   | 71        |
| Configurer KDiff3 comme outil de fusion.....   | 71        |
| Configurer KDiff3 comme outil de diff.....   | 71        |
| Configuration d'un IDE IntelliJ en tant qu'outil de fusion (Windows).....                      | 71        |
| Configurer un IDE IntelliJ en tant qu'outil de diff (Windows).....                             | 72        |
| <b>Chapitre 20: GFS Large File Storage (LFS).....</b>  | <b>73</b> |
| Remarques.....   | 73        |
| Exemples.....  | 73        |
| Installer LFS.....   | 73        |

|  |           |
|--|-----------|
| Déclarez certains types de fichiers à stocker en externe .....                       | 73        |
| Définir la configuration de LFS pour tous les clones .....                           | 73        |
| <b>Chapitre 21: Git Clean .....</b>  | <b>75</b> |
| Syntaxe .....  | 75        |
| Paramètres .....   | 75        |
| Exemples .....   | 75        |
| Nettoyer les fichiers ignorés .....  | 75        |
| Nettoyer tous les répertoires non suivis .....                                       | 75        |
| Supprimer avec force les fichiers non suivis .....                                   | 76        |
| Nettoyer interactivement .....   | 76        |
| <b>Chapitre 22: Git Diff .....</b>   | <b>77</b> |
| Syntaxe .....  | 77        |
| Paramètres .....   | 77        |
| Exemples .....   | 78        |
| Afficher les différences dans la branche de travail .....                            | 78        |
| Afficher les différences pour les fichiers mis en scène .....                        | 78        |
| Afficher à la fois les changements mis en scène et non modifiés .....                | 78        |
| Afficher les changements entre deux commits .....                                    | 78        |
| Utiliser meld pour voir toutes les modifications dans le répertoire de travail ..... | 79        |
| Afficher les différences pour un fichier ou un répertoire spécifique .....           | 79        |
| Affichage d'un mot-diff pour les longues lignes .....                                | 79        |
| Affichage d'une fusion à trois voies, y compris l'ancêtre commun .....               | 80        |
| Afficher les différences entre la version actuelle et la dernière version .....      | 81        |
| Fichiers texte codés UTF-16 Diff et fichiers plistes binaires .....                  | 81        |
| Comparer les branches .....  | 81        |
| Afficher les changements entre deux branches .....                                   | 82        |
| Produire un diff compatible avec les patches .....                                   | 82        |
| différence entre deux commit ou branch .....   | 82        |
| <b>Chapitre 23: git envoyer-email .....</b>  | <b>83</b> |
| Syntaxe .....  | 83        |
| Remarques .....  | 83        |
| Exemples .....   | 83        |



|  |           |
|--|-----------|
| Utiliser git send-email avec Gmail.....                      | 83        |
| Composition.....   | 83        |
| Envoi de patches par mail.....                               | 83        |
| <b>Chapitre 24: Git GUI Clients.....</b>                     | <b>85</b> |
| Exemples.....  | 85        |
| GitHub Desktop.....  | 85        |
| Git Kraken.....  | 85        |
| SourceTree.....  | 85        |
| gitk et git-gui.....   | 85        |
| SmartGit.....  | 87        |
| Extensions Git.....  | 88        |
| <b>Chapitre 25: Git Remote.....</b>                          | <b>89</b> |
| Syntaxe.....   | 89        |
| Paramètres.....  | 89        |
| Exemples.....  | 90        |
| Ajouter un référentiel distant.....                          | 90        |
| Renommer un référentiel distant.....                         | 90        |
| Supprimer un référentiel distant.....                        | 90        |
| Afficher les dépôts distants.....                            | 90        |
| Changer l'URL distante de votre dépôt Git.....               | 91        |
| Afficher plus d'informations sur le référentiel distant..... | 91        |
| <b>Chapitre 26: Git rerere.....</b>                          | <b>93</b> |
| Introduction.....  | 93        |
| Exemples.....  | 93        |
| Activation de la rereere.....                                | 93        |
| <b>Chapitre 27: git-svn.....</b>                             | <b>94</b> |
| Remarques.....   | 94        |
| <b>Dépannage.....</b>  | <b>94</b> |
| Exemples.....  | 95        |
| Cloner le dépôt SVN.....                                     | 95        |
| Obtenir les dernières modifications de SVN.....              | 95        |
| Modification locale de SVN.....                              | 95        |

|   |            |
|---|------------|
| Travailler localement.....  | 95         |
| Gestion des dossiers vides.....   | 96         |
| <b>Chapitre 28: git-tfs.....</b>  | <b>98</b>  |
| Remarques.....  | 98         |
| Exemples.....   | 98         |
| git-tfs clone.....  | 98         |
| git-tfs clone du dépôt git.....   | 98         |
| git-tfs installer via Chocolatey.....   | 98         |
| git-tfs Check In.....   | 98         |
| git-tfs pousser.....  | 99         |
| <b>Chapitre 29: Historique de réécriture avec filtre-branche.....</b>                 | <b>100</b> |
| Exemples.....   | 100        |
| Changer l'auteur des commits.....   | 100        |
| Réglage de git committer égal à commit author.....                                    | 100        |
| <b>Chapitre 30: Ignorer les fichiers et les dossiers.....</b>                         | <b>101</b> |
| Introduction.....   | 101        |
| Exemples.....   | 101        |
| Ignorer les fichiers et les répertoires avec un fichier .gitignore.....               | 101        |
| <b>Exemples.....</b>  | <b>101</b> |
| <b>Autres formes de .gitignore.....</b>   | <b>103</b> |
| <b>Nettoyage des fichiers ignorés.....</b>  | <b>103</b> |
| Exceptions dans un fichier .gitignore.....  | 103        |
| Un fichier global .gitignore.....   | 104        |
| Ignorer les fichiers qui ont déjà été validés dans un référentiel Git.....            | 104        |
| Vérifier si un fichier est ignoré.....  | 105        |
| Ignorer les fichiers dans les sous-dossiers (fichiers gitignore multiples).....       | 106        |
| Ignorer un fichier dans n'importe quel répertoire.....                                | 106        |
| Ignorer les fichiers localement sans valider les règles ignore.....                   | 106        |
| Modèles .gitignore pré-remplis.....   | 107        |
| Ignorer les modifications ultérieures apportées à un fichier (sans le supprimer)..... | 107        |
| Ignorer seulement une partie d'un fichier [stub].....                                 | 108        |

|   |            |
|---|------------|
| Ignorer les modifications dans les fichiers suivis. [bout].....           | 109        |
| Effacer les fichiers déjà validés, mais inclus dans .gitignore.....       | 109        |
| Créer un dossier vide.....  | 110        |
| Recherche de fichiers ignorés par .gitignore.....                         | 110        |
| <b>Chapitre 31: Internes.....</b>   | <b>112</b> |
| Exemples.....   | 112        |
| Repo.....   | 112        |
| Objets.....   | 112        |
| HEAD ref.....   | 112        |
| Refs.....   | 112        |
| Objet de validation.....  | 113        |
| <b>Arbre.....</b>   | <b>113</b> |
| <b>Parent.....</b>  | <b>113</b> |
| Objet d'arbre.....  | 114        |
| Objet blob.....   | 114        |
| Créer de nouveaux commits.....  | 115        |
| Déplacement de la tête.....   | 115        |
| Déplacement des références.....   | 115        |
| Créer de nouvelles références.....  | 115        |
| <b>Chapitre 32: Liasses.....</b>  | <b>116</b> |
| Remarques.....  | 116        |
| Exemples.....   | 116        |
| Créer un bundle git sur la machine locale et l'utiliser sur un autre..... | 116        |
| <b>Chapitre 33: Liste de révocation.....</b>                              | <b>117</b> |
| Syntaxe.....  | 117        |
| Paramètres.....   | 117        |
| Exemples.....   | 117        |
| Liste Commit en maître mais pas en origine / maître.....                  | 117        |
| <b>Chapitre 34: Marquage Git.....</b>                                     | <b>118</b> |
| Introduction.....   | 118        |
| Syntaxe.....  | 118        |

|   |            |
|---|------------|
| Exemples.....   | 118        |
| Liste de tous les tags disponibles.....                                       | 118        |
| Créer et envoyer des tags dans GIT.....                                       | 118        |
| <b>Chapitre 35: Mettre à jour le nom de l'objet dans la référence.....</b>    | <b>120</b> |
| Exemples.....   | 120        |
| Mettre à jour le nom de l'objet dans la référence.....                        | 120        |
| Utilisation.....  | 120        |
| SYNOPSIS.....   | 120        |
| Syntaxe générale.....   | 120        |
| <b>Chapitre 36: Migration vers Git.....</b>                                   | <b>122</b> |
| Exemples.....   | 122        |
| Migrer de SVN vers Git en utilisant l'utilitaire de conversion Atlassian..... | 122        |
| SubGit.....   | 123        |
| Migrer de SVN à Git en utilisant svn2git.....                                 | 123        |
| Migration de Team Foundation Version Control (TFVC) vers Git.....             | 123        |
| Migration de Mercurial à Git.....   | 124        |
| <b>Chapitre 37: Mise en scène.....</b>  | <b>125</b> |
| Remarques.....  | 125        |
| Exemples.....   | 125        |
| Mise en scène d'un seul fichier.....  | 125        |
| Mise en place de toutes les modifications apportées aux fichiers.....         | 125        |
| Stade fichiers supprimés.....   | 125        |
| Dégager un fichier contenant des modifications.....                           | 126        |
| Ajout interactif.....   | 126        |
| Ajouter des modifications par morceau.....                                    | 127        |
| Afficher les modifications par étapes.....                                    | 127        |
| <b>Chapitre 38: Montrer.....</b>  | <b>128</b> |
| Syntaxe.....  | 128        |
| Remarques.....  | 128        |
| Exemples.....   | 128        |
| Vue d'ensemble.....   | 128        |
| Pour les commits:.....  | 128        |

|   |            |
|---|------------|
| Pour les arbres et les blobs:.....  | 128        |
| Pour les tags:.....   | 128        |
| <b>Chapitre 39: Nom de la branche Git sur Bash Ubuntu.....</b>                | <b>129</b> |
| Introduction.....   | 129        |
| Exemples.....   | 129        |
| Nom de la succursale dans le terminal.....                                    | 129        |
| <b>Chapitre 40: Parcourir l'historique.....</b>                               | <b>130</b> |
| Syntaxe.....  | 130        |
| Paramètres.....   | 130        |
| Remarques.....  | 130        |
| Exemples.....   | 130        |
| "Git Log" régulier.....   | 130        |
| Journal en ligne.....   | 131        |
| Journal plus joli.....  | 131        |
| Connectez-vous avec les modifications en ligne.....                           | 132        |
| Recherche de journal.....   | 133        |
| Liste toutes les contributions regroupées par nom d'auteur.....               | 133        |
| Filtrer les journaux.....   | 134        |
| Journal d'une plage de lignes dans un fichier.....                            | 134        |
| Coloriser les journaux.....   | 135        |
| Une ligne indiquant le nom du commetteur et l'heure depuis la validation..... | 135        |
| Git Log Entre Deux Branches.....  | 136        |
| Journal affichant les fichiers validés.....                                   | 136        |
| Afficher le contenu d'un seul commit.....                                     | 136        |
| Recherche d'une chaîne de validation dans le journal git.....                 | 137        |
| <b>Chapitre 41: Patch Git.....</b>  | <b>138</b> |
| Syntaxe.....  | 138        |
| Paramètres.....   | 138        |
| Exemples.....   | 139        |
| Créer un patch.....   | 139        |
| Appliquer des patches.....  | 140        |
| <b>Chapitre 42: Perte.....</b>  | <b>141</b> |

|  |            |
|--|------------|
| Exemples.....  | 141        |
| Annuler les fusions.....   | 141        |
| Utiliser le reflog.....  | 142        |
| Retour à un engagement précédent.....  | 143        |
| Annuler les modifications.....   | 143        |
| Inverser certains commits existants.....   | 144        |
| Annuler / Refaire une série de commits.....  | 145        |
| <b>Chapitre 43: Ramification.....</b>  | <b>147</b> |
| Syntaxe.....   | 147        |
| Paramètres.....  | 147        |
| Remarques.....   | 147        |
| Exemples.....  | 148        |
| Liste des succursales.....   | 148        |
| Créer et vérifier de nouvelles branches.....   | 148        |
| Supprimer une branche localement.....  | 149        |
| Découvrez une nouvelle branche de suivi d'une succursale distante.....                         | 150        |
| Renommer une branche.....  | 150        |
| Ecraser un seul fichier dans le répertoire de travail en cours avec le même fichier d'une..... | 150        |
| Supprimer une branche distante.....  | 151        |
| Créer une branche orpheline (c'est-à-dire une branche sans engagement parent).....             | 151        |
| Poussez la branche à distance.....   | 151        |
| Déplacer la branche actuelle HEAD vers un commit arbitraire.....                               | 152        |
| Passage rapide à la branche précédente.....  | 152        |
| Recherche dans les branches.....   | 152        |
| <b>Chapitre 44: Rangement de votre référentiel local et distant.....</b>                       | <b>153</b> |
| Exemples.....  | 153        |
| Supprimer les branches locales qui ont été supprimées sur la télécommande.....                 | 153        |
| <b>Chapitre 45: Rebasing.....</b>  | <b>154</b> |
| Syntaxe.....   | 154        |
| Paramètres.....  | 154        |
| Remarques.....   | 154        |
| Exemples.....  | 154        |

|  |            |
|--|------------|
| Rebranchement de branche locale .....  | 155        |
| Rebase: les nôtres et les leurs, local et distant .....  | 155        |
| <b>Inversion illustrée .....</b>   | <b>156</b> |
| Sur une fusion: .....  | 156        |
| Sur un rebase: .....   | 156        |
| Rebase interactif .....  | 157        |
| <b>Reformulation des messages de validation .....</b>  | <b>157</b> |
| <b>Changer le contenu d'un commit .....</b>  | <b>157</b> |
| <b>Fractionnement d'un seul engagement en plusieurs .....</b>                                  | <b>157</b> |
| <b>Écraser plusieurs commets en un .....</b>   | <b>158</b> |
| Abandon d'une base de données interactive .....  | 158        |
| Pousser après un rebase .....  | 158        |
| Rebase à la validation initiale .....  | 159        |
| Rebasing avant une revue de code .....   | 159        |
| <b>Résumé .....</b>  | <b>159</b> |
| <b>En supposant: .....</b>   | <b>159</b> |
| <b>Stratégie: .....</b>  | <b>159</b> |
| <b>Exemple: .....</b>  | <b>159</b> |
| <b>résumer .....</b>   | <b>161</b> |
| Configuration de git-pull pour effectuer automatiquement une rebase au lieu d'une fusion ..... | 161        |
| Tester tous les commits pendant le rebase .....  | 162        |
| Configuration de l'autostash .....   | 162        |
| <b>Chapitre 46: Récupérer .....</b>  | <b>163</b> |
| Exemples .....   | 163        |
| Récupérer d'un engagement perdu .....  | 163        |
| Restaurer un fichier supprimé après une validation .....                                       | 163        |
| Restaurer le fichier vers une version précédente .....   | 163        |
| Récupérer une branche supprimée .....  | 163        |
| Récupération d'une réinitialisation .....  | 164        |
| Avec Git, vous pouvez (presque) toujours revenir en arrière .....                              | 164        |
| Récupérer de Git Stash .....   | 164        |

|  |            |
|--|------------|
| <b>Chapitre 47: Référentiels de clonage</b> .....  | <b>166</b> |
| Syntaxe.....   | 166        |
| Exemples.....  | 166        |
| Clone peu profond.....   | 166        |
| Clone Régulier.....  | 166        |
| Cloner une branche spécifique.....   | 167        |
| Cloner récursivement.....  | 167        |
| Cloner en utilisant un proxy.....  | 167        |
| <b>Chapitre 48: Reflog - Restauration des commits non affichés dans le journal git</b> ..... | <b>168</b> |
| Remarques.....   | 168        |
| Exemples.....  | 168        |
| Se remettre d'une mauvaise rebase.....   | 168        |
| <b>Chapitre 49: Renommer</b> .....   | <b>169</b> |
| Syntaxe.....   | 169        |
| Paramètres.....  | 169        |
| Exemples.....  | 169        |
| Renommer les dossiers.....   | 169        |
| Renommer une succursale locale.....  | 169        |
| renommer une branche locale et distante.....   | 169        |
| <b>Chapitre 50: Répertoires vides dans Git</b> .....   | <b>170</b> |
| Exemples.....  | 170        |
| Git ne suit pas les répertoires.....   | 170        |
| <b>Chapitre 51: Résoudre les conflits de fusion</b> .....                                    | <b>171</b> |
| Exemples.....  | 171        |
| Résolution manuelle.....   | 171        |
| <b>Chapitre 52: S'engager</b> .....  | <b>172</b> |
| Introduction.....  | 172        |
| Syntaxe.....   | 172        |
| Paramètres.....  | 172        |
| Exemples.....  | 172        |
| S'engager sans ouvrir un éditeur.....  | 173        |



|   |            |
|---|------------|
| Modifier un engagement.....   | 173        |
| Commettre des changements directement.....                          | 174        |
| Créer un commit vide.....   | 174        |
| Stage et validation des modifications.....                          | 174        |
| <b>Les bases.....</b>   | <b>174</b> |
| <b>Raccourcis.....</b>  | <b>175</b> |
| <b>Données sensibles.....</b>                                       | <b>175</b> |
| S'engager au nom de quelqu'un d'autre.....                          | 176        |
| Commettre des modifications dans des fichiers spécifiques.....      | 176        |
| Bon commettre des messages.....                                     | 176        |
| <b>Les sept règles d'un grand message de validation de git.....</b> | <b>177</b> |
| S'engager à une date précise.....                                   | 177        |
| Sélection des lignes à mettre en scène pour la validation.....      | 177        |
| Modifier le temps d'un engagement.....                              | 178        |
| Modifier l'auteur d'un commit.....                                  | 178        |
| La signature GPG s'engage.....                                      | 179        |
| <b>Chapitre 53: Se cacher.....</b>                                  | <b>180</b> |
| Syntaxe.....  | 180        |
| Paramètres.....   | 180        |
| Remarques.....  | 181        |
| Exemples.....   | 181        |
| Qu'est-ce que Stashing?.....  | 181        |
| Créer une cachette.....   | 182        |
| Liste des caches enregistrés.....                                   | 183        |
| Montrer la cachette.....  | 183        |
| Supprimer la cachette.....  | 183        |
| Appliquez et enlevez la cachette.....                               | 183        |
| Appliquer stash sans le retirer.....                                | 183        |
| Récupération des modifications antérieures de la réserve.....       | 184        |
| Cachette partielle.....   | 184        |
| Appliquer une partie d'une cachette avec la caisse.....             | 184        |
| Stashing interactif.....  | 184        |

|  |            |
|--|------------|
| Déplacez votre travail en cours vers une autre branche.....                      | 185        |
| Récupérer une cachette tombée.....   | 185        |
| <b>Chapitre 54: Sous-arbres.....</b>   | <b>187</b> |
| Syntaxe.....   | 187        |
| Remarques.....   | 187        |
| Exemples.....  | 187        |
| Créer, extraire et sous-déplacer.....  | 187        |
| <b>Créer un sous-arbre.....</b>  | <b>187</b> |
| <b>Pull Subtree Updates.....</b>   | <b>187</b> |
| <b>Mises à jour de sous-arborescence Backport.....</b>                           | <b>187</b> |
| <b>Chapitre 55: Statistiques Git.....</b>  | <b>189</b> |
| Syntaxe.....   | 189        |
| Paramètres.....  | 189        |
| Exemples.....  | 189        |
| Commit par développeur.....  | 189        |
| Commit par date.....   | 190        |
| Nombre total de commits dans une succursale.....                                 | 190        |
| Liste de chaque branche et la date de sa dernière révision.....                  | 190        |
| Lignes de code par développeur.....  | 190        |
| Liste tous les commits dans un joli format.....                                  | 190        |
| Trouver tous les dépôts de Git locaux sur ordinateur.....                        | 190        |
| Affiche le nombre total de commits par auteur.....                               | 191        |
| <b>Chapitre 56: Syntaxe des révisions Git.....</b>                               | <b>192</b> |
| Remarques.....   | 192        |
| Exemples.....  | 192        |
| Spécification de la révision par nom d'objet.....                                | 192        |
| Noms de référence symboliques: branches, tags, branches de suivi à distance..... | 192        |
| La révision par défaut: HEAD.....  | 193        |
| Références de reflog: @ { }.....   | 193        |
| Références de reflog: @ { }.....   | 193        |
| Branche suivie / en amont: @{en amont}.....                                      | 193        |
| Commit la chaîne d'ascendance: ^, ~ , etc.....                                   | 194        |

|  |            |
|--|------------|
| Déréférencement des branches et des tags: ^ 0, ^ { }.....  | 194        |
| Plus jeune engagement correspondant: ^ {/ };: /.....       | 195        |
| <b>Chapitre 57: Tirant.....</b>                            | <b>196</b> |
| Introduction.....  | 196        |
| Syntaxe.....   | 196        |
| Paramètres.....  | 196        |
| Remarques.....   | 196        |
| Exemples.....  | 196        |
| Mise à jour avec les modifications locales.....            | 196        |
| Tirez le code de la télécommande.....                      | 197        |
| Tirez, écrasez local.....                                  | 197        |
| Garder une histoire linéaire en tirant.....                | 197        |
| <b>Rebasing en tirant.....</b>                             | <b>197</b> |
| <b>En faisant le comportement par défaut.....</b>          | <b>197</b> |
| <b>Vérifiez si fast-forwardable.....</b>                   | <b>198</b> |
| Pull, "permission refusée".....                            | 198        |
| Extraction de modifications dans un référentiel local..... | 198        |
| <b>Traction simple.....</b>                                | <b>198</b> |
| <b>Tirez d'une autre télécommande ou branche.....</b>      | <b>198</b> |
| <b>Traction manuelle.....</b>                              | <b>198</b> |
| <b>Chapitre 58: TortoiseGit.....</b>                       | <b>200</b> |
| Exemples.....  | 200        |
| Ignorer les fichiers et les dossiers.....                  | 200        |
| Ramification.....  | 200        |
| Assumer inchangé.....                                      | 202        |
| <b>Revenir "Assumer inchangé".....</b>                     | <b>203</b> |
| Squash commit.....   | 204        |
| <b>Le moyen facile.....</b>                                | <b>204</b> |
| <b>Le moyen avancé.....</b>                                | <b>205</b> |
| <b>Chapitre 59: Travailler avec des télécommandes.....</b> | <b>207</b> |
| Syntaxe.....   | 207        |

|   |            |
|---|------------|
| Exemples.....   | 207        |
| Ajout d'un nouveau référentiel distant.....                           | 207        |
| Mise à jour à partir du référentiel en amont.....                     | 207        |
| ls-remote.....  | 207        |
| Suppression d'une succursale distante.....                            | 208        |
| Suppression des copies locales des branches distantes supprimées..... | 208        |
| Afficher des informations sur une télécommande spécifique.....        | 208        |
| Liste des télécommandes existantes.....                               | 208        |
| Commencer.....  | 209        |
| Syntaxe pour pousser vers une branche distante.....                   | 209        |
| Exemple.....  | 209        |
| Définir en amont sur une nouvelle succursale.....                     | 209        |
| Changer un référentiel distant.....                                   | 209        |
| Changer l'URL de Git Remote.....                                      | 209        |
| Renommer une télécommande.....  | 210        |
| Définir l'URL d'une télécommande spécifique.....                      | 211        |
| Obtenir l'URL d'une télécommande spécifique.....                      | 211        |
| <b>Chapitre 60: Utiliser un fichier .gitattributes.....</b>           | <b>212</b> |
| Exemples.....   | 212        |
| Désactiver la normalisation de fin de ligne.....                      | 212        |
| Normalisation automatique de fin de ligne.....                        | 212        |
| Identifier les fichiers binaires.....                                 | 212        |
| Modèles .gitattribute pré-remplis.....                                | 212        |
| <b>Chapitre 61: Worktrees.....</b>                                    | <b>213</b> |
| Syntaxe.....  | 213        |
| Paramètres.....   | 213        |
| Remarques.....  | 213        |
| Exemples.....   | 213        |
| Utiliser un worktree.....   | 213        |
| Déplacement d'un worktree.....  | 214        |
| <b>Crédits.....</b>   | <b>216</b> |

---

# À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [git](#)

It is an unofficial and free Git ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Git.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to [info@zzzprojects.com](mailto:info@zzzprojects.com)

---

# Chapitre 1: Démarrer avec Git

## Remarques

Git est un système de contrôle de version distribué gratuit qui permet aux programmeurs de suivre les modifications du code, via des "instantanés" (commits), dans leur état actuel. L'utilisation de validations permet aux programmeurs de tester, déboguer et créer de nouvelles fonctionnalités en collaboration. Tous les commits sont conservés dans ce que l'on appelle un "référentiel Git" pouvant être hébergé sur votre ordinateur, des serveurs privés ou des sites Web open source, tels que Github.

Git permet également aux utilisateurs de créer de nouvelles "branches" du code, ce qui permet aux différentes versions du code de cohabiter. Cela permet des scénarios où une branche contient la version stable la plus récente, une branche différente contient un ensemble de nouvelles fonctionnalités en cours de développement, et une autre branche contient un ensemble de fonctionnalités différent. Git crée le processus de création de ces branches, puis les fusionne par la suite, presque sans douleur.

Git a 3 "zones" différentes pour votre code:

- **Répertoire de travail** : zone dans laquelle vous allez travailler (création, modification, suppression et organisation des fichiers)
- **Zone de transit** : la zone dans laquelle vous listerez les modifications apportées au répertoire de travail
- **Référentiel** : où Git stocke en permanence les modifications que vous avez apportées sous différentes versions du projet

Git a été créé à l'origine pour gérer les sources du noyau Linux. En les rendant plus faciles, il encourage les petits commits, la création de projets et la fusion entre les fourchettes, ainsi que la présence de nombreuses branches à courte durée de vie.

Le plus grand changement pour les personnes qui sont habituées à CVS ou à Subversion est que chaque extraction contient non seulement l'arborescence des sources, mais aussi l'historique complet du projet. Les opérations courantes comme la diffraction des révisions, la vérification d'anciennes révisions, la validation (de votre historique local), la création d'une branche, l'extraction d'une branche, la fusion de branches ou de fichiers de correctifs peuvent être effectuées localement sans avoir à communiquer avec un serveur central. Ainsi, la plus grande source de latence et de manque de fiabilité est supprimée. La communication avec le référentiel "en amont" est uniquement nécessaire pour obtenir les dernières modifications et pour publier vos modifications locales sur d'autres développeurs. Cela transforme ce qui était auparavant une contrainte technique (quiconque possède le référentiel du projet) en un choix organisationnel (votre "en amont" est celui avec qui vous choisissez de synchroniser).

## Versions

| Version | Date de sortie |
|---------|----------------|
| 2.13    | 2017-05-10     |
| 2.12    | 2017-02-24     |
| 2,11.1  | 2017-02-02     |
| 2.11    | 2016-11-29     |
| 2.10.2  | 2016-10-28     |
| 2.10    | 2016-09-02     |
| 2.9     | 2016-06-13     |
| 2.8     | 2016-03-28     |
| 2.7     | 2015-10-04     |
| 2.6     | 2015-09-28     |
| 2,5     | 2015-07-27     |
| 2.4     | 2015-04-30     |
| 2.3     | 2015-02-05     |
| 2.2     | 2014-11-26     |
| 2.1     | 2014-08-16     |
| 2.0     | 2014-05-28     |
| 1,9     | 2014-02-14     |
| 1.8.3   | 2013-05-24     |
| 1.8     | 2012-10-21     |
| 1.7.10  | 2012-04-06     |
| 1,7     | 2010-02-13     |
| 1.6.5   | 2009-10-10     |
| 1.6.3   | 2009-05-07     |
| 1.6     | 2008-08-17     |
| 1.5.3   | 2007-09-02     |

| Version | Date de sortie |
|---------|----------------|
| 1,5     | 2007-02-14     |
| 1.4     | 2006-06-10     |
| 1.3     | 2006-04-18     |
| 1.2     | 2006-02-12     |
| 1.1     | 2006-01-08     |
| 1.0     | 2005-12-21     |
| 0,99    | 2005-07-11     |

## Exemples

### Créez votre premier référentiel, puis ajoutez et validez des fichiers

Sur la ligne de commande, vérifiez d'abord que vous avez installé Git:

Sur tous les systèmes d'exploitation:

```
git --version
```

Sur les systèmes d'exploitation de type UNIX:

```
which git
```

Si rien n'est renvoyé ou si la commande n'est pas reconnue, vous devrez peut-être installer Git sur votre système en téléchargeant et en exécutant le programme d'installation. Voir la [page d'accueil Git](#) pour des instructions d'installation exceptionnellement claires et faciles.

Après avoir installé Git, [configurez votre nom d'utilisateur et votre adresse e-mail](#) . Faites ceci *avant de* faire un commit.

Une fois Git installé, accédez au répertoire que vous souhaitez placer sous contrôle de version et créez un référentiel Git vide:

```
git init
```

Cela crée un dossier caché, `.git` , qui contient la plomberie nécessaire au fonctionnement de Git.

Ensuite, vérifiez quels fichiers Git ajoutera à votre nouveau référentiel; cette étape mérite une attention particulière:

```
git status
```



Examinez la liste de fichiers obtenue; vous pouvez indiquer à Git quels fichiers placer dans le contrôle de version (évittez d'ajouter des fichiers contenant des informations confidentielles telles que des mots de passe ou des fichiers qui encombrerent le dépôt):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

Si tous les fichiers de la liste doivent être partagés avec tous ceux qui ont accès au référentiel, une seule commande ajoute tout dans votre répertoire actuel et ses sous-répertoires:

```
git add .
```

Cela "mettra en scène" tous les fichiers à ajouter au contrôle de version, en les préparant à être validés lors de votre premier commit.

Pour les fichiers que vous ne souhaitez jamais contrôler par version, créez et nommez `.gitignore` un fichier nommé `.gitignore` avant d'exécuter la commande `add .`

Validez tous les fichiers ajoutés avec un message de validation:

```
git commit -m "Initial commit"
```

Cela crée un nouvel **commit** avec le message donné. Un commit est comme une sauvegarde ou un instantané de l'ensemble de votre projet. Vous pouvez maintenant le transférer ou le télécharger vers un référentiel distant, et plus tard vous pourrez y revenir si nécessaire. Si vous omettez le paramètre `-m`, votre éditeur par défaut s'ouvre et vous pouvez éditer et enregistrer le message de validation.

## Ajouter une télécommande

Pour ajouter une nouvelle télécommande, utilisez la commande `git remote add` sur le terminal, dans le répertoire où se trouve votre référentiel.

La commande `git remote add` prend deux arguments:

1. Un nom distant, par exemple, `origin`
2. Une URL distante, par exemple, `https://<your-git-service-address>/user/repo.git`

```
git remote add origin https://<your-git-service-address>/owner/repository.git
```

**REMARQUE:** Avant d'ajouter la télécommande, vous devez créer le référentiel requis dans votre service git. Vous pourrez pousser / extraire les commits après avoir ajouté votre télécommande.

## Cloner un référentiel

La commande `git clone` est utilisée pour copier un référentiel Git existant d'un serveur vers l'ordinateur local.

Par exemple, pour cloner un projet GitHub:

```
cd <path where you'd like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

Pour cloner un projet BitBucket:

```
cd <path where you'd like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

Cela crée un répertoire appelé `projectname` sur la machine locale, contenant tous les fichiers du référentiel Git distant. Cela inclut les fichiers source du projet, ainsi qu'un sous-répertoire `.git` contenant l'historique complet et la configuration du projet.

Pour spécifier un nom différent du répertoire, par exemple `MyFolder` :

```
git clone https://github.com/username/projectname.git MyFolder
```

Ou pour cloner dans le répertoire courant:

```
git clone https://github.com/username/projectname.git .
```

Remarque:

1. Lors du clonage vers un répertoire spécifié, le répertoire doit être vide ou inexistant.
2. Vous pouvez également utiliser la version `ssh` de la commande:

```
git clone git@github.com:username/projectname.git
```

La version `https` et la version `ssh` sont équivalentes. Cependant, certains services d'hébergement tels que GitHub vous **recommandent** d'utiliser `https` plutôt que `ssh` .

## Configuration de la télécommande en amont

Si vous avez cloné un fork (par exemple un projet open source sur Github), vous ne disposerez peut-être pas d'un accès push au référentiel en amont, vous avez donc besoin des deux, mais vous pourrez récupérer le référentiel en amont.

Vérifiez d'abord les noms distants:

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

Si `upstream` existe déjà (c'est sur *certaines* versions de Git), vous devez définir l'URL (actuellement vide):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

Si l'amont n'est **pas** là, ou si vous souhaitez également ajouter un fork d'un ami / collègue (actuellement, ils n'existent pas):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

## Code de partage

Pour partager votre code, vous créez un référentiel sur un serveur distant sur lequel vous allez copier votre référentiel local.

Pour minimiser l'utilisation de l'espace sur le serveur distant, vous créez un référentiel dénudé: celui qui ne contient que les objets `.git` et ne crée pas de copie de travail dans le système de fichiers. En prime, vous [définissez cette télécommande](#) comme un serveur en amont pour partager facilement des mises à jour avec d'autres programmeurs.

Sur le serveur distant:

```
git init --bare /path/to/repo.git
```

Sur la machine locale:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Notez que `ssh:` n'est qu'un moyen possible d'accéder au référentiel distant.)

Maintenant, copiez votre référentiel local sur la télécommande:

```
git push --set-upstream origin master
```

L'ajout de `--set-upstream` (ou `-u`) a créé une référence en amont (tracking) qui est utilisée par les commandes Git sans argument, par exemple `git pull`.

## Définition de votre nom d'utilisateur et de votre adresse e-mail

```
You need to set who you are *before* creating any commit. That will allow commits to have the right author name and email associated to them.
```

**Cela n'a rien à voir avec l'authentification lors de l'envoi dans un référentiel distant** (par exemple, lors de l'envoi dans un référentiel distant à l'aide de votre compte GitHub, BitBucket ou GitLab).

Pour déclarer cette identité pour *tous les référentiels*, utilisez `git config --global`. Cela stockera le paramètre dans le fichier `.gitconfig` votre utilisateur, par exemple `$HOME/.gitconfig` ou Windows, `%USERPROFILE%\gitconfig`.

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

Pour déclarer une identité pour un seul référentiel, utilisez `git config` dans un dépôt. Cela stockera les paramètres dans le référentiel individuel, dans le fichier `$(GIT_DIR)/config` . par exemple `/path/to/your/repo/.git/config` .

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

Les paramètres stockés dans le fichier de configuration d'un référentiel ont priorité sur la configuration globale lorsque vous utilisez ce référentiel.

Astuces: si vous avez des identités différentes (une pour un projet open-source, une au travail, une pour les repos privés, ...) et que vous ne voulez pas oublier de définir la bonne pour chaque mise en pension sur laquelle vous travaillez :

- **Supprimer une identité globale**

```
git config --global --remove-section user.name
git config --global --remove-section user.email
```

## 2.8

- Pour forcer git à rechercher votre identité uniquement dans les paramètres d'un référentiel, pas dans la configuration globale:

```
git config --global user.useConfigOnly true
```

De cette façon, si vous oubliez de définir votre `user.name` et `user.email` pour un dépôt donné et essayez de vous engager, vous verrez:

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

## Apprendre une commande

Pour obtenir plus d'informations sur une commande git - c.-à-d. Des détails sur la commande, les options disponibles et toute autre documentation - utilisez l'option `--help` ou la commande `help` .

Par exemple, pour obtenir toutes les informations disponibles sur la commande `git diff` , utilisez:

```
git diff --help
git help diff
```

De même, pour obtenir toutes les informations disponibles sur la commande `status` , utilisez:

```
git status --help
git help status
```

Si vous voulez seulement une aide rapide pour vous montrer la signification des indicateurs de ligne de commande les plus utilisés, utilisez `-h` :

```
git checkout -h
```

## Mettre en place SSH pour Git

Si vous utilisez **Windows**, ouvrez [Git Bash](#) . Si vous utilisez **Mac** ou **Linux**, ouvrez votre terminal.

Avant de générer une clé SSH, vous pouvez vérifier si vous avez des clés SSH existantes.

Listez le contenu de votre répertoire `~/.ssh` :

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

Vérifiez la liste des répertoires pour voir si vous avez déjà une clé SSH publique. Par défaut, les noms de fichiers des clés publiques sont l'un des suivants:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

Si vous voyez une paire de clés publique et privée existante que vous souhaitez utiliser sur votre compte Bitbucket, GitHub (ou similaire), vous pouvez copier le contenu du fichier `id_*.pub` .

Sinon, vous pouvez créer une nouvelle paire de clés publique et privée avec la commande suivante:

```
$ ssh-keygen
```

Appuyez sur la touche Entrée ou Retour pour accepter l'emplacement par défaut. Entrez et ressaisissez une phrase secrète lorsque vous y êtes invité ou laissez-la vide.

Assurez-vous que votre clé SSH est ajoutée à l'agent ssh. Démarrez l'agent ssh en arrière-plan s'il n'est pas déjà en cours d'exécution:

```
$ eval "$(ssh-agent -s)"
```

Ajoutez votre clé SSH à l'agent ssh. Notez que vous devrez remplacer `id_rsa` dans la commande par le nom de votre **fichier de clé privée** :

```
$ ssh-add ~/.ssh/id_rsa
```

Si vous souhaitez modifier l'amont d'un référentiel existant de HTTPS à SSH, vous pouvez exécuter la commande suivante:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Pour cloner un nouveau dépôt sur SSH, vous pouvez exécuter la commande suivante:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

## Installation de Git

Entrons dans l'utilisation de Git. Tout d'abord, vous devez l'installer. Vous pouvez l'obtenir de plusieurs façons. Les deux principaux sont de l'installer à partir de la source ou d'installer un paquet existant pour votre plate-forme.

### Installation à partir de la source

Si vous le pouvez, il est généralement utile d'installer Git depuis la source, car vous obtiendrez la version la plus récente. Chaque version de Git a tendance à inclure des améliorations utiles de l'interface utilisateur, donc obtenir la dernière version est souvent le meilleur itinéraire si vous vous sentez à l'aise pour compiler des logiciels à partir des sources. Il est également vrai que de nombreuses distributions Linux contiennent de très vieux paquets. Donc, à moins que vous ne soyez sur une distribution très à jour ou que vous utilisiez des backports, l'installation à partir de la source peut être la meilleure solution.

Pour installer Git, vous devez disposer des bibliothèques suivantes dont Git dépend: curl, zlib, openssl, expat et libiconv. Par exemple, si vous êtes sur un système doté de yum (tel que Fedora) ou apt-get (tel qu'un système basé sur Debian), vous pouvez utiliser l'une de ces commandes pour installer toutes les dépendances:

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

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

Lorsque vous avez toutes les dépendances nécessaires, vous pouvez aller chercher le dernier instantané sur le site Web de Git:

<http://git-scm.com/download> Ensuite, compilez et installez:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Après cela, vous pouvez également obtenir Git via Git lui-même pour les mises à jour:

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

## Installation sous Linux

Si vous souhaitez installer Git sur Linux via un programme d'installation binaire, vous pouvez généralement le faire via l'outil de gestion de paquetage de base fourni avec votre distribution. Si vous êtes sur Fedora, vous pouvez utiliser yum:

```
$ yum install git
```

Ou si vous utilisez une distribution basée sur Debian comme Ubuntu, essayez apt-get:

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

## Installation sur Mac

Il existe trois moyens simples d'installer Git sur un Mac. Le plus simple est d'utiliser l'installateur graphique Git, que vous pouvez télécharger depuis la page SourceForge.

<http://sourceforge.net/projects/git-osx-installer/>

Figure 1-7. Installateur Git OS X. L'autre méthode majeure consiste à installer Git via MacPorts ( <http://www.macports.org> ) . Si vous avez installé MacPorts, installez Git via

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

Vous n'avez pas à ajouter tous les extras, mais vous voudrez probablement inclure + svn au cas où vous devriez utiliser Git avec les dépôts Subversion (voir Chapitre 8).

Homebrew ( <http://brew.sh/> ) est une autre alternative pour installer Git. Si vous avez installé Homebrew, installez Git via

```
$ brew install git
```

## Installation sous Windows

L'installation de Git sur Windows est très simple. Le projet msysGit a l'une des procédures d'installation les plus faciles. Téléchargez simplement le fichier d'installation exe depuis la page GitHub et exécutez-le:

```
http://msysgit.github.io
```

Une fois installé, vous disposez à la fois d'une version en ligne de commande (y compris un client SSH utile ultérieurement) et de l'interface graphique standard.

*Remarque sur l'utilisation de Windows:* vous devez utiliser Git avec le shell msysGit fourni (style Unix), cela permet d'utiliser les lignes de commandes complexes données dans ce livre. Si vous avez besoin, pour une raison quelconque, d'utiliser la console native de shell / ligne de commande Windows, vous devez utiliser des guillemets plutôt que des guillemets simples (pour les paramètres contenant des espaces) et vous devez citer les paramètres se terminant par l'accent circumflex (^) s'ils sont en dernier sur la ligne, car il s'agit d'un symbole de continuation dans Windows.

Lire Démarrer avec Git en ligne: <https://riptutorial.com/fr/git/topic/218/demarrer-avec-git>



---

# Chapitre 2: Afficher l'historique des validations graphiquement avec Gitk

## Exemples

### Afficher l'historique des validations pour un fichier

```
gitk path/to/myfile
```

### Afficher tous les commits entre deux commits

Disons que vous avez deux commits `d9e1db9` et `5651067` et que vous voulez voir ce qui s'est passé entre eux. `d9e1db9` est l'ancêtre le plus ancien et `5651067` est le descendant final de la chaîne de commits.

```
gitk --ancestry-path d9e1db9 5651067
```

### Affichage validé depuis la balise de version

Si vous avez la version `v2.3` vous pouvez afficher tous les commits depuis cette balise.

```
gitk v2.3..
```

Lire [Afficher l'historique des validations graphiquement avec Gitk en ligne](https://riptutorial.com/fr/git/topic/3637/afficher-l-historique-des-validations-graphiquement-avec-gitk):

<https://riptutorial.com/fr/git/topic/3637/afficher-l-historique-des-validations-graphiquement-avec-gitk>

# Chapitre 3: Alias

## Exemples

### Alias simples

Il existe deux manières de créer des alias dans Git:

- avec le fichier `~/.gitconfig` :

```
[alias]
  ci = commit
  st = status
  co = checkout
```

- avec la ligne de commande:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

Une fois l'alias créé, tapez:

- `git ci` au lieu de `git commit` ,
- `git st` au lieu de `git status` ,
- `git co` au lieu de `git checkout` .

Comme pour les commandes git standard, les alias peuvent être utilisés à côté des arguments. Par exemple:

```
git ci -m "Commit message..."
git co -b feature-42
```

### List / search aliases existants

Vous pouvez [lister les alias git existants](#) en utilisant `--get-regexp` :

```
$ git config --get-regexp '^alias\.'
```

## Recherche d'alias

Pour [rechercher des alias](#) , ajoutez ce qui suit à votre `.gitconfig` sous `[alias]` :

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"\$1\" \"#\"
```

Ensuite vous pouvez:

- `git aliases` - affiche TOUS les alias
- `git aliases commit` - uniquement les alias contenant "commit"

## Alias avancés

Git vous permet d'utiliser des commandes non-git et une syntaxe `sh` shell complète dans vos alias si vous les préfixez avec `!`.

Dans votre fichier `~/.gitconfig` :

```
[alias]
temp = !git add -A && git commit -m "Temp"
```

Le fait que la syntaxe complète du shell soit disponible dans ces alias préfixés signifie également que vous pouvez utiliser des fonctions shell pour construire des alias plus complexes, tels que ceux qui utilisent des arguments de ligne de commande:

```
[alias]
ignore = "!f() { echo $1 >> .gitignore; }; f"
```

L'alias ci-dessus définit la fonction `f`, puis l'exécute avec tous les arguments que vous transmettez à l'alias. Donc, lancer `git ignore .tmp/` ajouterait `.tmp/` à votre fichier `.gitignore`.

En fait, ce modèle est si utile que Git définit pour vous les variables `$1`, `$2`, etc., vous n'avez donc même pas besoin de définir une fonction spéciale. (Mais gardez à l'esprit que Git ajoutera également les arguments de toute façon, même si vous y accédez via ces variables, vous pouvez donc ajouter une commande factice à la fin.)

Notez que les alias avec le préfixe avec `!` De cette façon, ils sont exécutés à partir du répertoire racine de votre commande `git`, même si votre répertoire actuel est plus profond dans l'arborescence. Cela peut être un moyen utile d'exécuter une commande à partir de la racine sans avoir à y `cd` explicitement.

```
[alias]
ignore = "! echo $1 >> .gitignore"
```

## Ignorer temporairement les fichiers suivis

Pour marquer temporairement un fichier comme ignoré (fichier pass comme paramètre pour alias) - tapez:

```
unwatch = update-index --assume-unchanged
```

Pour recommencer le suivi du fichier - tapez:

```
watch = update-index --no-assume-unchanged
```

Pour répertorier tous les fichiers temporairement ignorés - tapez:

```
unwatched = "!git ls-files -v | grep '^[:lower:]'"
```

Pour effacer la liste non surveillée - tapez:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Exemple d'utilisation des alias:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

## Afficher un joli journal avec un graphique de branche

```
[alias]
logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

lg = log --graph --date-order --first-parent \
  --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green) (%ad) %C(bold
cyan)<%an>%Creset'
lgb = log --graph --date-order --branches --first-parent \
  --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green) (%ad) %C(bold
cyan)<%an>%Creset'
lga = log --graph --date-order --all \
  --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green) (%ad) %C(bold
cyan)<%an>%Creset'
```

Voici une explication des options et des espaces réservés utilisés dans le format `--pretty` (une liste exhaustive est disponible avec le `git help log`)

`--graph` - dessine l'arbre de validation

`--date-order` - utilise l'ordre d'horodatage de validation lorsque cela est possible

`--first-parent` - ne suit que le premier parent du noeud de fusion.

`--branches` - affiche toutes les branches locales (par défaut, seule la branche actuelle est affichée)

`--all` - affiche toutes les branches locales et distantes

`% h` - valeur de hachage pour commit (abrégé)

`% ad` - Cachet de date (auteur)

`% an` - Nom d'utilisateur auteur

`% an` - Nom d'utilisateur de validation

`% C (auto)` - pour utiliser les couleurs définies dans la section `[color]`

`% Creset` - pour réinitialiser la couleur

% d - --decorate (noms de branches et de tags)

% s - valide le message

% ad - date de l'auteur (suivra la directive --date) (et non la date du commiter)

% an - nom de l'auteur (peut être %cn pour le nom du commutateur)

## Mise à jour du code tout en conservant un historique linéaire

Parfois, vous devez conserver un historique linéaire (sans branchement) de vos commits de code. Si vous travaillez sur une branche pendant un certain temps, cela peut être difficile si vous devez effectuer une `git pull` régulière de `git pull` car cela enregistrera une fusion avec l'amont.

```
[alias]
up = pull --rebase
```

Cela mettra à jour avec votre source en amont, puis réappliquera tout travail que vous n'avez pas poussé au-dessus de ce que vous avez abattu.

Utiliser:

```
git up
```

## Voir quels fichiers sont ignorés par votre configuration .gitignore

```
[ alias ]

ignored = ! git ls-files --others --ignored --exclude-standard --directory \
&& git ls-files --others -i --exclude-standard
```

Affiche une ligne par fichier, vous pouvez donc grep (uniquement les répertoires):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

Ou compte:

```
~$ git ignored | wc -l
199811          # oops, my home directory is getting crowded
```

## Dégager les fichiers mis en scène

Normalement, pour supprimer les fichiers à transférer à l'aide de la validation de `git reset`, la `reset` a beaucoup de fonctions en fonction des arguments fournis. Pour désinstaller complètement tous les fichiers mis en scène, nous pouvons utiliser des alias git pour créer un nouvel alias qui utilise la `reset` mais nous n'avons plus besoin de nous souvenir de fournir les arguments corrects

à `reset` .

```
git config --global alias.unstage "reset --"
```

Maintenant, chaque fois que vous voulez **désinstaller des fichiers de scène**, tapez `git unstage` et vous êtes `git unstage` à partir.

Lire Alias en ligne: <https://riptutorial.com/fr/git/topic/337/alias>

---

# Chapitre 4: Analyse des types de workflows

## Remarques

Utiliser un logiciel de contrôle de version tel que Git peut être un peu effrayant au début, mais sa conception intuitive, spécialisée dans la création de branches, permet de créer différents types de flux de travail. Choisissez celui qui convient à votre propre équipe de développement.

## Exemples

### Workflow Gitflow

Initialement proposé par [Vincent Driessen](#), Gitflow est un workflow de développement utilisant git et plusieurs branches prédéfinies. Cela peut être considéré comme un cas particulier du [workflow Feature Branch](#).

L'idée de celui-ci est d'avoir des branches séparées réservées à des parties spécifiques en développement:

- `master` branche principale est toujours le code de *production* le plus récent. Le code expérimental n'appartient pas ici.
- `develop` branche contient tous les derniers *développements*. Ces changements de développement peuvent être à peu près n'importe quoi, mais des fonctionnalités plus importantes sont réservées à leurs propres branches. Le code ici est toujours travaillé et fusionné dans la `release` avant publication / déploiement.
- `hotfix` branches de `hotfix` sont destinées aux correctifs mineurs, qui ne peuvent pas attendre la prochaine version. `hotfix` branches de `hotfix` sortent de `master` et sont fusionnées en `master` et en `develop`.
- `release` branches de `release` sont utilisées pour libérer de nouveaux développements de `develop` à `master`. Tous les changements de dernière minute, tels que les numéros de version superflus, sont effectués dans la branche de publication, puis sont fusionnés en `master` et `develop`. Lors du déploiement d'une nouvelle version, `master` doit être associé au numéro de version actuel (par exemple, à l'aide du [contrôle de version sémantique](#)) pour référence ultérieure et restauration rapide.
- `feature` sont réservées aux fonctionnalités plus importantes. Celles-ci sont spécifiquement développées dans des branches désignées et intégrées au `develop` lorsque vous avez terminé. Les `feature` dédiées aident à séparer le développement et à pouvoir déployer des fonctionnalités *effectuées* indépendamment les unes des autres.

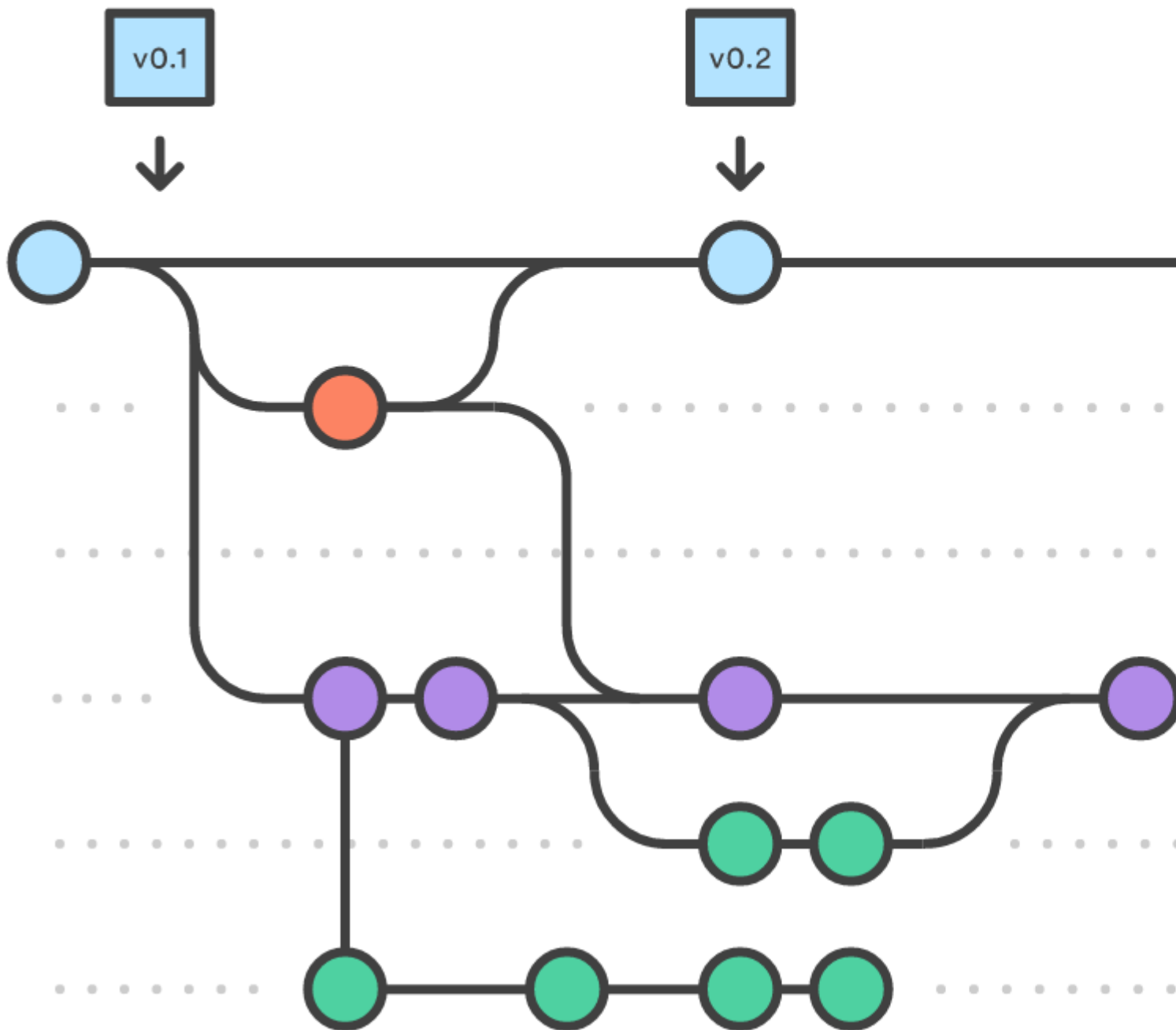
Une représentation visuelle de ce modèle:

Master

Hotfix

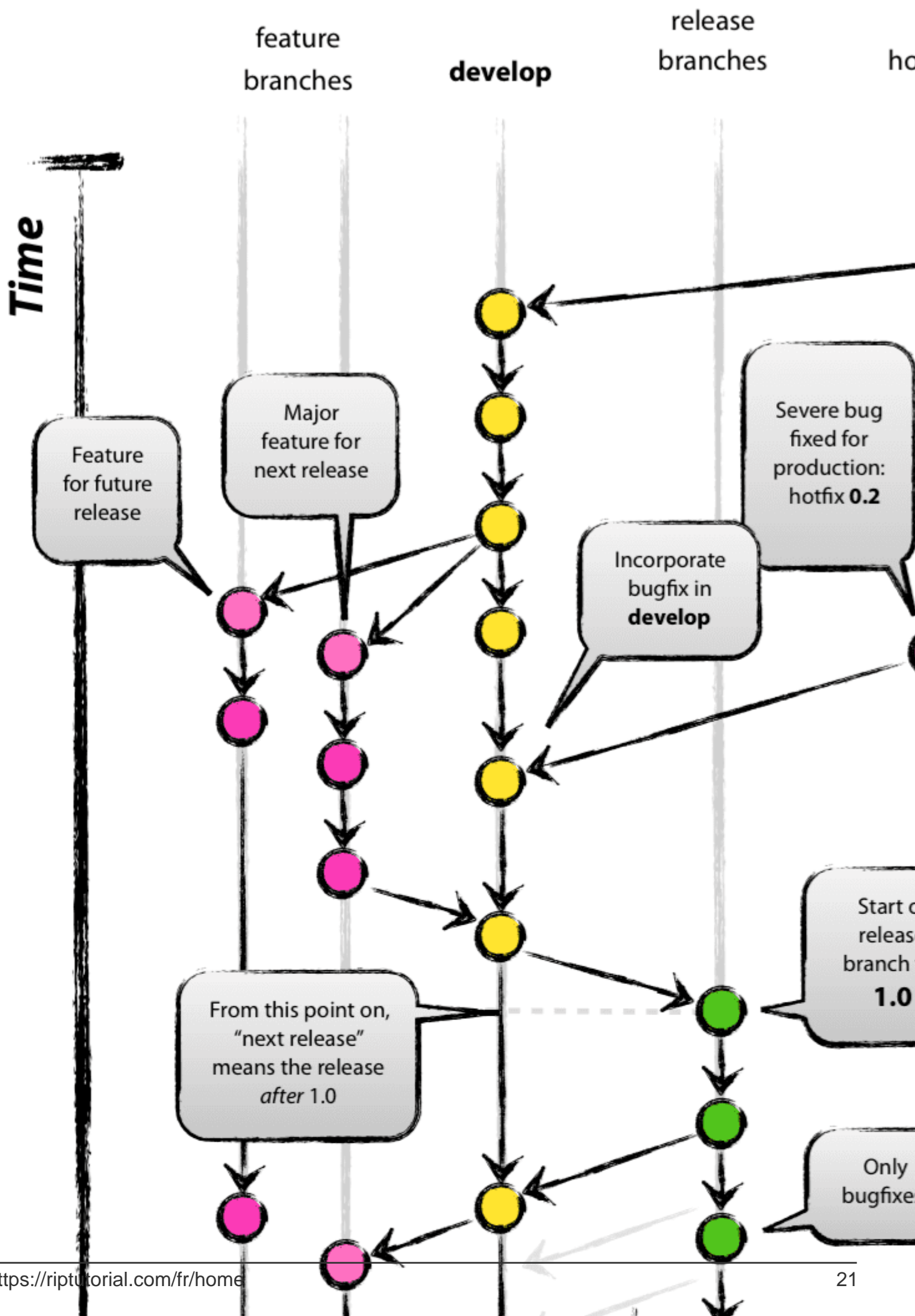
Release

Develop



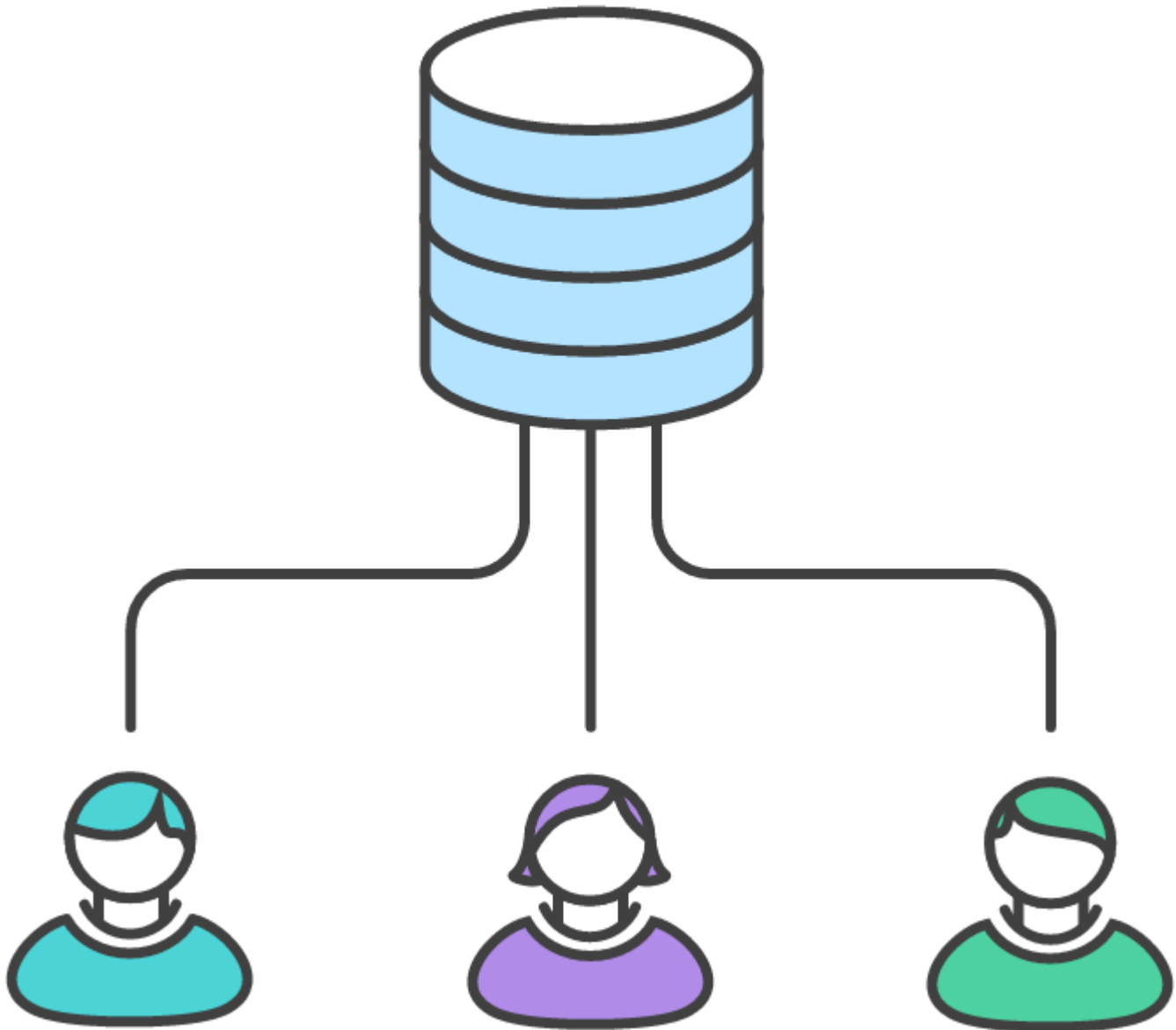
La représentation originale de ce modèle:





branche contient tout développement actif. Les contributeurs devront être particulièrement sûrs de tirer les dernières modifications avant de poursuivre leur développement, car cette branche évoluera rapidement. Tout le monde a accès à ce dépôt et peut engager des modifications directement dans la branche principale.

Représentation visuelle de ce modèle:



C'est le paradigme classique du contrôle de version, sur lequel des systèmes plus anciens tels que Subversion et CVS ont été construits. Les logiciels fonctionnant de cette manière sont appelés systèmes de contrôle de version centralisés ou CVCS. Bien que Git soit capable de fonctionner de cette façon, il existe des inconvénients notables, tels que la nécessité de précéder chaque traction par une fusion. Il est très possible pour une équipe de travailler de cette façon, mais la résolution constante des conflits de fusion peut finir par prendre beaucoup de temps précieux.

C'est pourquoi Linus Torvalds a créé Git non pas en tant que CVCS, mais plutôt en tant que *système de contrôle de version distribué (DVCS)* ou *distribution*, similaire à Mercurial. L'avantage de cette nouvelle façon de faire est la flexibilité démontrée dans les autres exemples

de cette page.

## Workflow de la fonction Feature

L'idée de base derrière la Direction générale Feature Workflow est que tout le développement de fonctionnalités devrait avoir lieu dans une branche dédiée au lieu du `master` branche. Cette encapsulation facilite le travail de plusieurs développeurs sur une fonctionnalité particulière sans perturber la base de code principale. Cela signifie également que la branche `master` ne contiendra jamais de code cassé, ce qui représente un avantage considérable pour les environnements d'intégration continue.

Le développement de fonctionnalités d'encapsulation permet également de tirer parti des requêtes d'extraction, qui permettent d'initier des discussions autour d'une branche. Ils permettent aux autres développeurs d'accepter une fonctionnalité avant qu'elle ne soit intégrée au projet officiel. Ou, si vous êtes coincé au milieu d'une fonctionnalité, vous pouvez ouvrir une demande de tirage en demandant des suggestions à vos collègues. Le fait est que les demandes de tirage facilitent énormément les commentaires de votre équipe sur le travail de chacun.

basé sur des [didacticiels Atlassian](#) .

## GitHub Flow

Populaire dans de nombreux projets open source, mais pas seulement.

**La branche principale** d'un emplacement spécifique (Github, Gitlab, Bitbucket, serveur local) contient la dernière version livrable. Pour chaque nouvelle fonctionnalité / correction de bogue / changement architectural, chaque développeur crée une branche.

Les modifications se produisent sur cette branche et peuvent être discutées dans une demande d'extraction, une révision de code, etc. Une fois acceptées, elles sont fusionnées dans la branche principale.

Flux complet par Scott Chacon:

- Tout ce qui se trouve dans la branche principale est déployable
- Pour travailler sur quelque chose de nouveau, créez une branche nommée nommée à partir de `master` (ex: `new-oauth2-scopes`)
- S'engager localement dans cette branche et pousser régulièrement votre travail sur la même branche nommée sur le serveur
- Lorsque vous avez besoin de commentaires ou d'aide, ou que vous pensez que la succursale est prête à fusionner, ouvrez une demande d'extraction.
- Une fois que quelqu'un a passé en revue et signé la fonctionnalité, vous pouvez la fusionner en `master`
- Une fois fusionné et poussé à «maîtriser», vous pouvez et devez déployer immédiatement

Présenté à l'origine sur [le site Web personnel de Scott Chacon](#) .

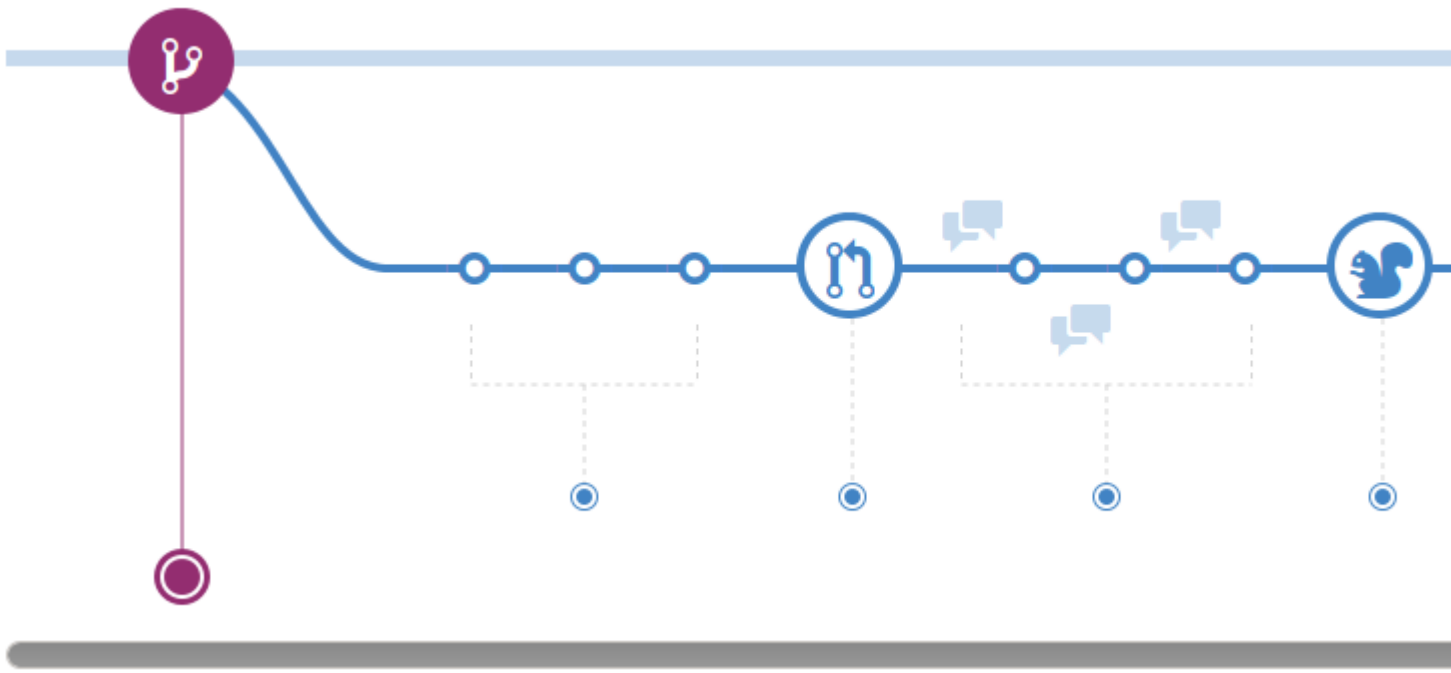


Image reproduite avec l'aimable autorisation de la [référence GitHub Flow](#)

Lire [Analyse des types de workflows en ligne](https://riptutorial.com/fr/git/topic/1276/analyse-des-types-de-workflows): <https://riptutorial.com/fr/git/topic/1276/analyse-des-types-de-workflows>

# Chapitre 5: arbre diff

## Introduction

Compare le contenu et le mode des blobs trouvés via deux objets arborescents.

## Exemples

### Voir les fichiers modifiés dans un commit spécifique

```
git diff-tree --no-commit-id --name-only -r COMMIT_ID
```

## Usage

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

| Option   | Explication  |
|----------|--|
| -r       | diff récursivement   |
| --racine | inclure le commit initial en tant que diff contre / dev / null |

### Options diff communes

| Option            | Explication  |
|-------------------|--|
| -z                | sortie diff-raw avec des lignes terminées par NUL. |
| -p                | format de patch de sortie.                         |
| -u                | synonyme de -p.                                    |
| --patch-with-raw  | sortir à la fois un patch et le format diff-raw.   |
| --stat            | afficher diffstat au lieu de patch.                |
| --numstat         | affiche le diffstat numérique au lieu du patch.    |
| --patch-with-stat | sortir un patch et ajouter son diffstat.           |
| --nom uniquement  | affiche uniquement les noms des fichiers modifiés. |
| --name-status     | afficher les noms et l'état des fichiers modifiés. |

| Option                          | Explication  |
|---------------------------------|--|
| --full-index                    | affiche le nom complet de l'objet sur les lignes d'index.                    |
| --abbrev = <n>                  | abrégier les noms d'objet dans l'en-tête de diff-tree et diff-raw.           |
| -R                              | permuter les paires de fichiers d'entrée.                                    |
| -B                              | détecter les réécritures complètes.  |
| -M                              | détecter les noms  |
| -C                              | détecter des copies.   |
| --find-copies-plus<br>difficile | essayer des fichiers inchangés comme candidat pour la détection de copie.    |
| -l <n>                          | limiter les tentatives de renommage aux chemins d'accès.                     |
| -O                              | réordonner les diffs selon le.   |
| -S                              | trouver filepair dont un seul côté contient la chaîne.                       |
| --pickaxe-all                   | affiche tous les fichiers diff lorsque -S est utilisé et que hit est trouvé. |
| -un texte                       | Traiter tous les fichiers comme du texte.                                    |

Lire arbre diff en ligne: <https://riptutorial.com/fr/git/topic/10937/arbre-diff>

# Chapitre 6: Archiver

## Syntaxe

- `archive git [--format = <fmt>] [--list] [--prefix = <prefix> /] [<extra>] [-o <fichier> | --output = <fichier>] [--worktree-attributes] [--remote = <repo> [--exec = <git-upload-archive>]] <arbre-ish> [<chemin> ...]`

## Paramètres

| Paramètre   | Détails  |
|---|--|
| <code>--format = &lt;fmt&gt;</code>                         | Format de l'archive résultante: <code>tar</code> ou <code>zip</code> . Si cette option n'est pas donnée et que le fichier de sortie est spécifié, le format est inféré du nom de fichier si possible. Sinon, par défaut, <code>tar</code> .  |
| <code>-l, --list</code>                                     | Afficher tous les formats disponibles.   |
| <code>-v, --verbose</code>                                  | Rapporter les progrès à <code>stderr</code> .  |
| <code>--prefix = &lt;préfixe&gt; /</code>                   | Ajoutez le préfixe <code>&lt;/&gt;</code> à chaque nom de fichier dans l'archive.  |
| <code>-o &lt;fichier&gt;, --output = &lt;fichier&gt;</code> | Écrivez l'archive dans <code>&lt;fichier&gt;</code> au lieu de <code>stdout</code> .   |
| <code>--worktree-attributes</code>                          | Recherchez les attributs dans les fichiers <code>.gitattributes</code> dans l'arborescence de travail.   |
| <code>&lt;extra&gt;</code>                                  | Cela peut être n'importe quelle option que le serveur d'archivage comprend. Pour le backend <code>zip</code> , utiliser <code>-0</code> stockera les fichiers sans les dégonfler, tandis que <code>-1</code> à <code>-9</code> pourra être utilisé pour ajuster la vitesse et le rapport de compression. |
| <code>--remote = &lt;repo&gt;</code>                        | Récupérez une archive <code>tar</code> à partir d'un référentiel distant <code>&lt;repo&gt;</code> plutôt que du référentiel local.  |
| <code>--exec = &lt;git-upload-archive&gt;</code>            | Utilisé avec <code>--remote</code> pour spécifier le chemin d'accès à <code>&lt;git-upload-archive&gt;</code> sur la télécommande.   |
| <code>&lt;arbre-ish&gt;</code>                              | L'arbre ou s'engage à produire une archive pour.   |
| <code>&lt;chemin&gt;</code>                                 | Sans paramètre facultatif, tous les fichiers et répertoires du répertoire de travail actuel sont inclus dans l'archive. Si un ou plusieurs chemins sont spécifiés, seuls ceux-ci sont inclus.  |

# Exemples

## Créer une archive du dépôt git avec le préfixe de répertoire

Il est conseillé d'utiliser un préfixe lors de la création d'archives git, de sorte que l'extraction place tous les fichiers dans un répertoire. Pour créer une archive de `HEAD` avec un préfixe de répertoire:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

Une fois extraits, tous les fichiers seront extraits dans un répertoire nommé `src-directory-name` dans le répertoire en cours.

## Créer une archive du dépôt git en fonction d'une branche, d'une révision, d'un tag ou d'un répertoire spécifique

Il est également possible de créer des archives d'autres éléments que `HEAD`, tels que des branches, des commits, des balises et des répertoires.

Pour créer une archive d'un `dev` local:

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

Pour créer une archive d'une `origin/dev` branche distante `origin/dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

Pour créer une archive d'une balise `v.01` :

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Créez une archive de fichiers dans un sous-répertoire spécifique ( `sub-dir` répertoire) de la révision `HEAD` :

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

## Créer une archive du dépôt git

Avec `git archive` il est possible de créer des archives compressées d'un référentiel, par exemple pour distribuer des versions.

Créez une archive tar de la révision `HEAD` cours:

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Créez une archive tar de la révision `HEAD` cours avec la compression gzip:



```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

Cela peut également être fait avec (qui utilisera la gestion tar.gz intégrée):

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Créez une archive zip de la révision `HEAD` cours:

```
git archive --format zip HEAD > archive-HEAD.zip
```

Il est également possible de spécifier un fichier de sortie avec une extension valide et le format et le type de compression en seront déduits:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Lire Archiver en ligne: <https://riptutorial.com/fr/git/topic/2815/archiver>

---

# Chapitre 7: Bisecting / Finding défectueux commits

## Syntaxe

- `git bisect <subcommand> <options>`
- `git bisect start <bad> [<good>...]`
- `git bisect reset`
- `git bisect good`
- `git bisect bad`

## Exemples

### Recherche binaire (git bisect)

`git bisect` vous permet de trouver quel commit a introduit un bogue en utilisant une recherche binaire.

Commencez par diviser une session en fournissant deux références de validation: une bonne validation avant le bogue et une mauvaise validation après le bogue. Généralement, le mauvais engagement est `HEAD`.

```
# start the git bisect session
$ git bisect start

# give a commit where the bug doesn't exist
$ git bisect good 49c747d

# give a commit where the bug exist
$ git bisect bad HEAD
```

`git` lance une recherche binaire: il divise la révision en deux et place le référentiel dans la révision intermédiaire. Inspectez le code pour déterminer si la révision est bonne ou mauvaise:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good

# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

`git` continuera à exécuter la recherche binaire sur chaque sous-ensemble de mauvaises révisions en fonction de vos instructions. `git` présentera une seule révision qui, à moins que vos indicateurs

ne soient pas corrects, représentera exactement la révision où le bogue a été introduit.

Ensuite, souvenez-vous de lancer `git bisect reset` pour terminer la session bisect et retourner à HEAD.

```
$ git bisect reset
```

Si vous avez un script qui peut vérifier le bogue, vous pouvez automatiser le processus avec:

```
$ git bisect run [script] [arguments]
```

Où `[script]` est le chemin d'accès à votre script et `[arguments]` est tout argument devant être transmis à votre script.

En exécutant cette commande, vous exécuterez automatiquement la recherche binaire, en exécutant `git bisect good` ou `git bisect bad` à chaque étape en fonction du code de sortie de votre script. Avec 0 indique la sortie `good`, en sortant avec 1-124, 126 ou 127 indique mauvais. 125 indique que le script ne peut pas tester cette révision (ce qui déclenchera un `git bisect skip`).

## Trouver automatiquement un commit défectueux

Imaginez que vous êtes sur le `master` branche et quelque chose ne fonctionne pas comme prévu (une régression a été introduite), mais vous ne savez pas où. Tout ce que vous savez, c'est que cela fonctionnait dans la dernière version (qui était par exemple balisé ou que vous connaissiez le hachage de validation, prenons ici le `old-rel`).

Git a de l'aide pour vous, en trouvant le commit défectueux qui a introduit la régression avec un très petit nombre d'étapes (recherche binaire).

Commencez par diviser:

```
git bisect start master old-rel
```

Cela indiquera à git que `master` est une révision cassée (ou la première version cassée) et `old-rel` est la dernière version connue.

Git va maintenant vérifier une tête détachée au milieu des deux commits. Maintenant, vous pouvez faire vos tests. Selon que cela fonctionne ou non

```
git bisect good
```

ou

```
git bisect bad
```

. Si cette validation ne peut pas être testée, vous pouvez facilement `git reset` et tester celle-ci, git will s'en charger.

Après quelques étapes, git affichera le hachage de validation défectueux.

Afin d'abandonner le processus bisect juste émettre

```
git bisect reset
```

et git restaurera l'état précédent.

Lire Bisecting / Finding défectueux commits en ligne:

<https://riptutorial.com/fr/git/topic/3645/bisecting---finding-defectueux-commits>

# Chapitre 8: Blâmer

## Syntaxe

- `git blame [filename]`
- `git blame [-f] [-e] [-w] [nomfichier]`
- `git blame [-L range] [nomfichier]`

## Paramètres

| Paramètre       | Détails   |
|-----------------|---|
| nom de fichier  | Nom du fichier pour lequel les détails doivent être vérifiés  |
| -F              | Afficher le nom du fichier dans le commit d'origine   |
| -e              | Afficher le courrier électronique de l'auteur au lieu du nom de l'auteur  |
| -w              | Ignorer les espaces blancs lors de la comparaison entre la version de l'enfant et celle du parent               |
| -L début, fin   | Afficher uniquement l'intervalle de ligne donné Exemple: <code>git blame -L 1,2 [filename]</code>               |
| --show-stats    | Affiche des statistiques supplémentaires à la fin de la sortie de blâme   |
| -l              | Montrer long rev (Par défaut: off)  |
| -t              | Afficher l'horodatage brut (par défaut: désactivé)  |
| -sens inverse   | Marchez l'histoire au lieu de reculer   |
| -p, --porcelain | Sortie pour consommation machine  |
| -M              | Détecter les lignes déplacées ou copiées dans un fichier  |
| -C              | En plus de -M, détecter les lignes déplacées ou copiées à partir d'autres fichiers modifiés dans le même commit |
| -h              | Afficher le message d'aide  |
| -c              | Utilisez le même mode de sortie que <code>git-annotate</code> (par défaut: off)                                 |
| -n              | Affiche le numéro de ligne dans le commit d'origine (par défaut: désactivé)                                     |

## Remarques

La commande `git blame` est très utile pour savoir qui a apporté des modifications à un fichier sur une base par ligne.

## Exemples

### Afficher le commit qui a modifié en dernier une ligne

```
git blame <file>
```

affichera le fichier avec chaque ligne annotée avec le dernier message modifié.

### Ignorer les modifications des espaces uniquement

Parfois, le repositionnement aura des commits qui ajustent uniquement les espaces, par exemple en fixant une indentation ou en basculant entre les tabulations et les espaces. Cela rend difficile la recherche du commit où le code a été écrit.

```
git blame -w
```

ignorera les modifications des espaces uniquement pour trouver l'origine de la ligne.

### Afficher uniquement certaines lignes

La sortie peut être restreinte en spécifiant des plages de lignes comme

```
git blame -L <start>,<end>
```

Où `<start>` et `<end>` peuvent être:

- numéro de ligne

```
git blame -L 10,30
```

- / regex /

```
git blame -L /void main/ , git blame -L 46,/void foo/
```

- + offset, -offset (uniquement pour `<end>` )

```
git blame -L 108,+30 , git blame -L 215,-15
```

Plusieurs plages de lignes peuvent être spécifiées et des plages de chevauchement sont autorisées.

```
git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40
```

### Pour savoir qui a changé un fichier

```
// Shows the author and commit per line of specified file
git blame test.c

// Shows the author email and commit per line of specified
git blame -e test.c file

// Limits the selection of lines by specified range
git blame -L 1,10 test.c
```

Lire Blâmer en ligne: <https://riptutorial.com/fr/git/topic/3663/blamer>

---

# Chapitre 9: Changer le nom du dépôt git

## Introduction

Si vous modifiez le nom du référentiel du côté distant, tel que votre github ou bitbucket, vous verrez une erreur lorsque vous appuierez votre code existant: Erreur fatale, référentiel introuvable \*\*.

## Exemples

### Changer le paramètre local

Aller au terminal,

```
cd projectFolder
git remote -v (it will show previous git url)
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v (double check, it will show new git url)
git push (do whatever you want.)
```

Lire [Changer le nom du dépôt git en ligne](https://riptutorial.com/fr/git/topic/9291/changer-le-nom-du-depot-git): <https://riptutorial.com/fr/git/topic/9291/changer-le-nom-du-depot-git>



# Chapitre 10: Configuration

## Syntaxe

- `git config [<option-fichier>] nom [valeur] #` l'un des cas d'utilisation les plus courants de `git config`

## Paramètres

| Paramètre             | Détails  |
|-----------------------|--|
| <code>--system</code> | Modifie le fichier de configuration à l'échelle du système, utilisé pour chaque utilisateur (sous Linux, ce fichier se trouve dans <code>\$(prefix)/etc/gitconfig</code> )   |
| <code>--global</code> | Edite le fichier de configuration global, qui est utilisé pour chaque référentiel sur lequel vous travaillez (sous Linux, ce fichier se trouve à <code>~/.gitconfig</code> ) |
| <code>--local</code>  | Modifie le fichier de configuration spécifique au <code>.git/config</code> situé à l' <code>.git/config</code> dans votre référentiel. Ce sont les paramètres par défauts    |

## Exemples

### Nom d'utilisateur et adresse e-mail

Juste après avoir installé Git, la première chose à faire est de définir votre nom d'utilisateur et votre adresse e-mail. À partir d'un shell, tapez:

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- `git config` est la commande pour obtenir ou définir des options
- `--global` signifie que le fichier de configuration spécifique à votre compte utilisateur sera modifié
- `user.name` et `user.email` sont les clés des variables de configuration; `user` est la section du fichier de configuration. `name` et `email` sont les noms des variables.
- `"Mr. Bean"` et `mrbean@example.com` sont les valeurs que vous `mrbean@example.com` dans les deux variables. Notez les guillemets autour de `"Mr. Bean"`, qui sont nécessaires car la valeur que vous stockez contient un espace.

### Plusieurs configurations git

Vous avez jusqu'à 5 sources pour la configuration de git:

- 6 fichiers:

- `%ALLUSERSPROFILE%\Git\Config` (Windows uniquement)
  - (system) `<git>/etc/gitconfig`, `<git>` étant le chemin d'installation de git. (sous Windows, il s'agit de `<git>\mingw64\etc\gitconfig`)
  - (system) `$XDG_CONFIG_HOME/git/config` (Linux / Mac uniquement)
  - (global) `~/.gitconfig` (Windows: `%USERPROFILE%\gitconfig`)
  - (local) `.git/config` (dans un git repo `$GIT_DIR`)
  - un **fichier dédié** (avec `git config -f`), utilisé par exemple pour modifier la configuration des sous-modules: `git config -f .gitmodules ...`
- **la ligne de commande avec** `git -c :git -c core.autocrlf=false fetch` redéfinit *tout* autre `core.autocrlf` sur `false`, *uniquement* pour cette commande `fetch`.

L'ordre est important: tout ensemble de configuration dans une source peut être remplacé par une source listée en dessous.

`git config --system/global/local` est la commande pour lister 3 de ces sources, mais seul `git config -l` listera *toutes les configurations résolues*.

"résolu" signifie qu'il ne liste que la valeur de configuration finale remplacée.

Depuis git 2.8, si vous voulez voir quelle configuration provient de quel fichier, vous tapez:

```
git config --list --show-origin
```

## Définition de l'éditeur à utiliser

Il existe plusieurs manières de définir quel éditeur utiliser pour la validation, le rebasage, etc.

- Modifiez le `core.editor` configuration `core.editor`.

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

- Définissez la variable d'environnement `GIT_EDITOR`.

Pour une commande:

```
$ GIT_EDITOR=nano git commit
```

Ou pour toutes les commandes exécutées dans un terminal. **Remarque:** Cela s'applique uniquement jusqu'à la fermeture du terminal.

```
$ export GIT_EDITOR=nano
```

- Pour modifier l'éditeur de *tous les* programmes de terminal, pas seulement Git, définissez la variable d'environnement `VISUAL` ou `EDITOR`. (Voir [VISUAL VS EDITOR](#).)

```
$ export EDITOR=nano
```

**Remarque:** comme ci-dessus, cela ne concerne que le terminal actuel. Votre shell aura

généralement un fichier de configuration pour vous permettre de le définir de manière permanente. (Sur `bash`, par exemple, ajoutez la ligne ci-dessus à votre `~/.bashrc` ou `~/.bash_profile`.)

Certains éditeurs de texte (principalement ceux de l'interface graphique) n'exécuteront qu'une seule instance à la fois et quitteront généralement si une instance de ceux-ci est déjà ouverte. Si tel est le cas pour votre éditeur de texte, Git imprimera le message `Aborting commit due to empty commit message.` sans vous permettre de modifier le message de validation en premier. Si cela vous arrive, consultez la documentation de votre éditeur de texte pour voir s'il a un drapeau `--wait` (ou similaire) qui le fera suspendre jusqu'à la fermeture du document.

## Configuration des fins de ligne

### La description

Lorsque vous travaillez avec une équipe qui utilise différents systèmes d'exploitation (OS) sur l'ensemble du projet, vous pouvez parfois rencontrer des problèmes lors du traitement des fins de ligne.

### Microsoft Windows

Lorsque vous travaillez sur le système d'exploitation Microsoft Windows, les fins de ligne sont normalement de type retour chariot + retour à la ligne (CR + LF). Ouvrir un fichier qui a été modifié à l'aide d'un ordinateur Unix tel que Linux ou OSX peut causer des problèmes, faisant croire que le texte n'a aucune fin de ligne. Cela est dû au fait que les systèmes Unix appliquent des fins de ligne différentes uniquement pour les sauts de ligne de formulaire.

Pour résoudre ce problème, vous pouvez exécuter les instructions suivantes

```
git config --global core.autocrlf=true
```

À la **caisse**, cette instruction garantit que les fins de ligne sont configurées conformément au système d'exploitation Microsoft Windows (LF -> CR + LF)

### Basé sur Unix (Linux / OSX)

De même, il peut y avoir des problèmes lorsque l'utilisateur du système d'exploitation basé sur Unix essaie de lire des fichiers qui ont été modifiés sous Microsoft Windows. Afin d'éviter des problèmes imprévus

```
git config --global core.autocrlf=input
```

Lors de la **validation**, cela changera les fins de ligne de CR + LF -> + LF

### configuration pour une seule commande

Vous pouvez utiliser `-c <name>=<value>` pour ajouter une configuration à une seule commande.

Pour vous engager en tant qu'autre utilisateur sans avoir à modifier vos paramètres dans `.gitconfig`:

```
git -c user.email = mail@example commit -m "some message"
```

Remarque: pour cet exemple, vous n'avez pas besoin de `user.name` à la fois `user.name` et `user.email`, git complétera les informations manquantes des commits précédents.

## Configurer un proxy

Si vous êtes derrière un proxy, vous devez en parler à git:

```
git config --global http.proxy http://my.proxy.com:portnumber
```

Si vous n'êtes plus derrière un proxy:

```
git config --global --unset http.proxy
```

## Correction automatique des fautes de frappe

```
git config --global help.autocorrect 17
```

Cela permet la correction automatique dans git et vous pardonnera pour vos erreurs mineures (par exemple, les `git stats` de `git status` au lieu du `git status`). Le paramètre que vous fournissez à `help.autocorrect` détermine combien de temps le système doit attendre, en dixièmes de seconde, avant d'appliquer automatiquement la commande corrigée. Dans la commande ci-dessus, 17 signifie que git doit attendre 1,7 seconde avant d'appliquer la commande corrigée.

Cependant, des erreurs plus importantes seront considérées comme des commandes manquantes. `git testingit quelque chose comme git testingit` entraînerait que `testingit is not a git command.`

## Lister et éditer la configuration actuelle

Git config vous permet de personnaliser le fonctionnement de git. Il est couramment utilisé pour définir votre nom et votre adresse e-mail ou votre éditeur favori ou la manière dont les fusions doivent être effectuées.

Pour voir la configuration actuelle

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

Pour éditer la configuration:

```
$ git config <key> <value>
$ git config core.ignorecase true
```

Si vous souhaitez que le changement soit vrai pour tous vos référentiels, utilisez `--global`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config --global core.editor vi
```

Vous pouvez lister à nouveau pour voir vos modifications.

## Nom d'utilisateur et adresse email multiples

Depuis Git 2.13, plusieurs noms d'utilisateur et adresses électroniques peuvent être configurés à l'aide d'un filtre de dossier.

## Exemple pour Windows:

### `.gitconfig`

Editer: `git config --global -e`

Ajouter:

```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
  path = .gitconfig-opensource.config
```

### Remarques

- L'ordre dépend, le dernier qui correspond "gagne".
- le / à la fin est nécessaire - par exemple "gitdir:D:/work" ne fonctionnera pas.
- le préfixe `gitdir:` est requis.

### `.gitconfig-travail.config`

Fichier dans le même répertoire que `.gitconfig`

```
[user]
  name = Money
  email = work@somewhere.com
```

### `.gitconfig-opensource.config`

Fichier dans le même répertoire que `.gitconfig`

```
[user]
  name = Nice
  email = cool@opensource.stuff
```

## Exemple pour Linux

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

Le contenu du fichier et les notes sous la section Windows.

Lire Configuration en ligne: <https://riptutorial.com/fr/git/topic/397/configuration>

---

# Chapitre 11: Crochets

## Syntaxe

- `.git / hooks / applypatch-msg`
- `.git / hooks / commit-msg`
- `.git / hooks / post-update`
- `.git / hooks / pre-applypatch`
- `.git / hooks / pre-commit`
- `.git / hooks / prepare-commit-msg`
- `.git / hooks / pre-push`
- `.git / hooks / pre-rebase`
- `.git / hooks / update`

## Remarques

`--no-verify` ou `-n` pour ignorer tous les hooks locaux de la commande git donnée.

Ex: `git commit -n`

Les informations sur cette page proviennent des documents [officiels Git](#) et d' [Atlassian](#) .

## Exemples

### Commit-msg

Ce hook est similaire au hook `prepare-commit-msg` , mais il est appelé après que l'utilisateur ait entré un message de validation plutôt qu'avant. Ceci est généralement utilisé pour avertir les développeurs si leur message de validation est dans un format incorrect.

Le seul argument transmis à ce hook est le nom du fichier contenant le message. Si vous n'aimez pas le message que l'utilisateur a entré, vous pouvez modifier ce fichier sur place (comme `prepare-commit-msg` ) ou vous pouvez abandonner la validation entièrement en quittant avec un statut différent de zéro.

L'exemple suivant permet de vérifier si le mot ticket suivi d'un numéro est présent dans le message de validation

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
then echo "Commit message KO, $word is missing"; exit 1;
else echo "Commit message OK"; exit 0;
fi
```

### Crochets locaux

Les hooks locaux n'affectent que les référentiels locaux dans lesquels ils résident. Chaque développeur peut modifier ses propres hooks locaux, de sorte qu'ils ne peuvent pas être utilisés de manière fiable pour appliquer une stratégie de validation. Ils sont conçus pour faciliter le respect de certaines directives par les développeurs et éviter des problèmes potentiels ultérieurement.

Il existe six types de hooks locaux: `pre-commit`, `prepare-commit-msg`, `commit-msg`, `post-commit`, `post-checkout` et `pre-rebase`.

Les quatre premiers crochets concernent les commits et vous permettent de contrôler chaque partie du cycle de vie d'un engagement. Les deux derniers vous permettent d'effectuer des actions supplémentaires ou des contrôles de sécurité pour les commandes `git checkout` et `git rebase`.

Tous les "pré-hooks" vous permettent de modifier l'action qui est sur le point de se dérouler, tandis que les "post-hooks" sont principalement utilisés pour les notifications.

## Post-caisse

Ce hook fonctionne de manière similaire au hook `post-commit`, mais il est appelé chaque fois que vous récupérez une référence avec `git checkout`. Cela pourrait être un outil utile pour effacer votre répertoire de travail des fichiers générés automatiquement qui, sinon, causeraient de la confusion.

Ce hook accepte trois paramètres:

1. la référence de la précédente tête,
2. le ref de la nouvelle HEAD, et
3. un indicateur indiquant s'il s'agissait d'une extraction de branche ou d'une extraction de fichier ( `1` ou `0`, respectivement).

Son état de sortie n'a aucun effet sur la `git checkout`.

## Post-commit

Ce hook est appelé immédiatement après le hook `commit-msg`. Il ne peut pas modifier le résultat de l'opération de `git commit`, il est donc principalement utilisé à des fins de notification.

Le script ne prend aucun paramètre et son statut de sortie n'affecte en rien la validation.

## Post-recevoir

Ce hook est appelé après une opération Push réussie. Il est généralement utilisé à des fins de notification.

Le script ne prend aucun paramètre, mais reçoit la même information que la `pre-receive` via l'entrée standard:



```
<old-value> <new-value> <ref-name>
```

## Pré-engagement

Ce hook est exécuté à chaque fois que vous lancez `git commit`, pour vérifier ce qui va être commis. Vous pouvez utiliser ce hook pour inspecter le snapshot sur le point d'être validé.

Ce type de hook est utile pour exécuter des tests automatisés pour vous assurer que le commit entrant ne rompt pas les fonctionnalités existantes de votre projet. Ce type de hook peut également vérifier les erreurs d'espacement ou d'EOL.

Aucun argument n'est transmis au script de pré-validation et quitter avec un statut non nul interrompt la validation entière.

## Prepare-commit-msg

Ce hook est appelé après le hook `pre-commit` pour remplir l'éditeur de texte avec un message de validation. Ceci est généralement utilisé pour modifier les messages de validation générés automatiquement pour les commits écrasés ou fusionnés.

Un à trois arguments sont passés à ce hook:

- Nom d'un fichier temporaire contenant le message.
- Le type de commit, soit
  - message (option `-m` ou `-F`),
  - template (option `-t`),
  - fusionner (si c'est un commit de fusion), ou
  - squash (s'il écrase d'autres commits).
- Le hachage SHA1 du commit concerné. Ceci n'est donné que si l'option `--amend -c`, `-C` ou `--amend` a été donnée.

Semblable à la `pre-commit`, quitter avec un statut non nul interrompt la validation.

## Pré-rebase

Ce hook est appelé avant que `git rebase` commence à modifier la structure du code. Ce crochet est généralement utilisé pour s'assurer qu'une opération de rebase est appropriée.

Ce hook prend 2 paramètres:

1. la branche en amont que la série a été fourchue, et
2. la branche étant rebasée (vide lors du rebasage de la branche en cours).

Vous pouvez abandonner l'opération de rebase en quittant avec un statut différent de zéro.

## Pré-recevoir

Ce hook est exécuté chaque fois que quelqu'un utilise `git push` pour pousser des commits vers le

référentiel. Il réside toujours dans le référentiel distant qui est la destination de la distribution et non dans le référentiel (local) d'origine.

Le hook s'exécute avant toute mise à jour des références. Il est généralement utilisé pour appliquer tout type de politique de développement.

Le script ne prend aucun paramètre, mais chaque ref est envoyé au script sur une ligne distincte sur l'entrée standard au format suivant:

```
<old-value> <new-value> <ref-name>
```

## Mettre à jour

Ce crochet est appelé après la `pre-receive` et fonctionne de la même manière. Il est appelé avant que quelque chose ne soit réellement mis à jour, mais il est appelé séparément pour chaque ref que l'on a poussé plutôt que pour toutes les références à la fois.

Ce hook accepte les 3 arguments suivants:

- nom de la référence en cours de mise à jour,
- ancien nom d'objet stocké dans la référence, et
- nouveau nom d'objet stocké dans la ref.

C'est la même information transmise à la `pre-receive`, mais puisque la `update` est appelée séparément pour chaque ref, vous pouvez rejeter certaines références tout en autorisant d'autres.

## Pré-pousser

Disponible dans [Git 1.8.2](#) et supérieur.

1.8

Les crochets de pré-poussée peuvent être utilisés pour empêcher une poussée. Les raisons pour lesquelles cela est utile sont les suivantes: le blocage des poussées manuelles accidentelles vers des branches spécifiques ou le blocage des push si un contrôle établi échoue (tests unitaires, syntaxe).

Un hook de pré-push est créé en créant simplement un fichier nommé `pre-push` sous `.git/hooks/`, et (**gotcha alert**), en s'assurant que le fichier est exécutable: `chmod +x ./git/hooks/pre-push`.

Voici un exemple d' [Hannah Wolfe](#) qui bloque un effort de maîtrise:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*\/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
then
```

```

    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r <
/dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
    exit 0 # push will execute
fi

```

Voici un exemple de [Volkan Unsal](#) qui fait en sorte que les tests RSpec passent avant d'autoriser la diffusion:

```

#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
  PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
    begin
      stdin.each { |line| print line }
      rescue Errno::EIO
        end
    end
  rescue PTY::ChildExited
    puts "Child process exit!"
  end

  # find out if there were any errors
  html = open(html_path).read
  examples = html.match(/(\d+) examples/)[0].to_i rescue 0
  errors = html.match(/(\d+) errors/)[0].to_i rescue 0
  if errors == 0 then
    errors = html.match(/(\d+) failure/)[0].to_i rescue 0
  end
  pending = html.match(/(\d+) pending/)[0].to_i rescue 0

  if errors.zero?
    puts "0 failed! #{examples} run, #{pending} pending"
    # HTML Output when tests ran successfully:
    # puts "View spec results at #{File.expand_path(html_path)}"
    sleep 1
    exit 0
  else
    puts "\aCOMMIT FAILED!!"
    puts "View your rspec results at #{File.expand_path(html_path)}"
    puts
    puts "#{errors} failed! #{examples} run, #{pending} pending"
    # Open HTML Ooutput when tests failed
    # `open #{html_path}`
    exit 1
  end
end

```

Comme vous pouvez le voir, il y a beaucoup de possibilités, mais l'essentiel est de `exit 0` si de bonnes choses sont arrivées, et de `exit 1` si de mauvaises choses se sont produites. Chaque fois que vous `exit 1` la poussée sera empêchée et votre code sera dans l'état où il était avant de `git push...`

Lorsque vous utilisez des hooks côté client, gardez à l'esprit que les utilisateurs peuvent ignorer tous les hooks côté client en utilisant l'option "--no-verify" sur un push. Si vous comptez sur le crochet pour appliquer le processus, vous pouvez être brûlé.

Documentation: [https://git-scm.com/docs/githooks#\\_pre\\_push](https://git-scm.com/docs/githooks#_pre_push)

Exemple officiel:

<https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample>

## Vérifier la construction Maven (ou un autre système de construction) avant de valider

```
.git/hooks/pre-commit
```

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "Running mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven build failed"
        exit 1
    fi
fi
```

## Transférer automatiquement certaines poussées vers d'autres référentiels

`post-receive` `ancrage` `post-receive` peuvent être utilisés pour transférer automatiquement les envois entrants vers un autre référentiel.

```
$ cat .git/hooks/post-receive

#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

    echo "$remote_ref" | egrep '^refs\*/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
        ref=`echo $remote_ref | sed -e 's/^refs\*/heads\///'`
        echo Forwarding feature branch to other repository: $ref
        git push -q --force other_repos $ref
    }

done
```

Dans cet exemple, l' `egrep` rationnelle `egrep` recherche un format de branche spécifique (ici: JIRA-12345 utilisé pour nommer les problèmes Jira). Vous pouvez laisser cette partie si vous voulez transférer toutes les branches, bien sûr.

Lire Crochets en ligne: <https://riptutorial.com/fr/git/topic/1330/crochets>

# Chapitre 12: Crochets côté client Git

## Introduction

Comme beaucoup d'autres systèmes de contrôle de version, Git permet de lancer des scripts personnalisés lorsque certaines actions importantes se produisent. Il existe deux groupes de ces points d'ancrage: côté client et côté serveur. Les hooks côté client sont déclenchés par des opérations telles que la validation et la fusion, tandis que les hooks côté serveur s'exécutent sur des opérations réseau telles que la réception de commits poussés. Vous pouvez utiliser ces crochets pour toutes sortes de raisons.

## Exemples

### Installation d'un crochet

Les hooks sont tous stockés dans le sous-répertoire `hooks` répertoire Git. Dans la plupart des projets, il s'agit de `.git/hooks`.

Pour activer un script de hook, placez un fichier dans le sous-répertoire `hooks` de votre répertoire `.git` nommé de manière appropriée (sans aucune extension) et exécutable.

### Git pre-push hook

Le script **pre-push** est appelé par `git push` après avoir vérifié l'état de la télécommande, mais avant que rien n'ait été poussé. Si ce script se termine avec un statut non nul, rien ne sera poussé.

Ce hook est appelé avec les paramètres suivants:

```
$1 -- Name of the remote to which the push is being done (Ex: origin)
$2 -- URL to which the push is being done (Ex:
https://<host>:<port>/<username>/<project_name>.git)
```

Les informations sur les commits en cours sont fournies sous forme de lignes à l'entrée standard sous la forme:

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

Valeurs d'échantillon:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

Dans l'exemple ci-dessous, un script pré-push a été extrait de `pre-push.sample` par défaut qui a

été automatiquement créé lors de l'initialisation d'un nouveau référentiel avec `git init`

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # Handle delete
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # New branch, examine all commits
            range="$local_sha"
        else
            # Update to existing branch, examine new commits
            range="$remote_sha..$local_sha"
        fi

        # Check for WIP commit
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "Found WIP commit in $local_ref, not pushing"
            exit 1
        fi
    fi
done

exit 0
```

Lire Crochets côté client Git en ligne: <https://riptutorial.com/fr/git/topic/8654/crochets-cote-client-git>

# Chapitre 13: Cueillette De Cerises

## Introduction

Une sélection de cerises prend le correctif introduit dans un commit et essaie de le réappliquer sur la branche sur laquelle vous vous trouvez.

Source: Livre Git SCM

## Syntaxe

- `git cherry-pick [--edit] [-n] [-m nombre parent] [-s] [-x] [--ff] [-S [id-clé]] commit ...`
- `Git cherry-pick --Continuer`
- `git cherry-pick --quit`
- `git cherry-pick --abort`

## Paramètres

| Paramètres               | Détails  |
|--------------------------|--|
| <code>-e, --edit</code>  | Avec cette option, <code>git cherry-pick</code> vous permettra de modifier le message de validation avant de valider.  |
| <code>-X</code>          | Lors de l'enregistrement de la validation, ajoutez une ligne indiquant "(cherry selected from commit...)" dans le message de validation d'origine afin d'indiquer la validation de cette modification. Ceci est fait uniquement pour les choix de cerises sans conflits. |
| <code>--ff</code>        | Si le HEAD actuel est le même que le parent du commit choisi, alors une avance rapide vers ce commit sera effectuée.   |
| <code>--continuer</code> | Continuez l'opération en cours en utilisant les informations du fichier <code>.git / séquenceur</code> . Peut être utilisé pour continuer après la résolution des conflits dans une sélection ou un retour en échec.   |
| <code>--quitter</code>   | Oubliez l'opération en cours. Peut être utilisé pour effacer l'état du séquenceur après un échec de sélection ou de retour.  |
| <code>--avorter</code>   | Annulez l'opération et revenez à l'état de pré-séquence.   |

## Exemples

### Copier un commit d'une branche à une autre

`git cherry-pick <commit-hash>` appliquera les modifications apportées à un commit existant dans une autre branche, tout en enregistrant un nouveau commit. Essentiellement, vous pouvez copier des commits de branche en branche.

Étant donné l'arbre suivant ([Source](#))

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
      \
      76cada - 62ecb3 - b886a0 [feature]
```

Disons que nous voulons copier `b886a0` sur master (sur `5a6057`).

Nous pouvons courir

```
git checkout master
git cherry-pick b886a0
```

Maintenant, notre arbre ressemblera à quelque chose comme:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
      \
      76cada - 62ecb3 - b886a0 [feature]
```

Où le nouvel commit `a66b23` a le même contenu (diff source, message de validation) que `b886a0` (mais un parent différent). Notez que le ramassage des cerises ne prendra en compte que les modifications sur ce commit (`b886a0` dans ce cas), pas toutes les modifications de la branche des fonctionnalités (pour cela, vous devrez soit utiliser le rebasage ou la fusion).

## Copier une gamme d'engagement d'une branche à l'autre

`git cherry-pick <commit-A>..<commit-B>` placera chaque validation *après* A et jusqu'à et y compris B au-dessus de la branche actuellement extraite.

`git cherry-pick <commit-A>^..<commit-B>` placera le commit A et chaque commit jusqu'à et y compris B au-dessus de la branche actuellement extraite.

## Vérifier si un choix de cerises est requis

Avant de lancer le processus `cherry-pick`, vous pouvez vérifier si la validation que vous souhaitez sélectionner existe déjà dans la branche cible, auquel cas vous n'avez rien à faire.

`git branch --contains <commit>` répertorie les branches locales contenant le commit spécifié.

`git branch -r --contains <commit>` inclut également des branches de suivi à distance dans la liste.

## Trouver des commits à appliquer à l'amont

Command `git cherry` montre les changements qui n'ont pas encore été choisis.



## Exemple:

```
git checkout master
git cherry development
```

... et voir un peu comme ceci:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

Les engagements qui ont été avec + seront ceux qui n'ont pas encore été choisis dans le development .

## Syntaxe:

```
git cherry [-v] [<upstream> [<head> [<limit>]]]
```

## Options:

**-v** Affiche les sujets de validation à côté des SHA1.

**<upstream>** branche en amont pour rechercher des validations équivalentes. Par défaut, la branche en amont de HEAD.

**<head>** branche de travail; par défaut à HEAD.

**<limit>** Ne pas signaler les commits jusqu'à (et y compris) la limite.

Consultez [la documentation de git-cherry](#) pour plus d'informations.

Lire Cueillette De Cerises en ligne: <https://riptutorial.com/fr/git/topic/672/cueillette-de-cerises>

---

# Chapitre 14: Des sous-modules

## Exemples

### Ajouter un sous-module

Vous pouvez inclure un autre référentiel Git en tant que dossier dans votre projet, suivi par Git:

```
$ git submodule add https://github.com/jquery/jquery.git
```

Vous devez ajouter et valider le nouveau fichier `.gitmodules` ; Cela indique à Git quels sous-modules doivent être clonés lorsque la `git submodule update` est exécutée.

### Cloner un dépôt Git avec des sous-modules

Lorsque vous clonez un référentiel utilisant des sous-modules, vous devez les initialiser et les mettre à jour.

```
$ git clone --recursive https://github.com/username/repo.git
```

Cela clone les sous-modules référencés et les place dans les dossiers appropriés (y compris les sous-modules dans les sous-modules). Cela équivaut à exécuter `git submodule update --init --recursive` immédiatement après la fin du clone.

### Mise à jour d'un sous-module

Un sous-module référence une validation spécifique dans un autre référentiel. Pour vérifier l'état exact référencé pour tous les sous-modules, exécutez

```
git submodule update --recursive
```

Parfois, au lieu d'utiliser l'état référencé, vous souhaitez mettre à jour votre extraction locale vers le dernier état de ce sous-module sur une télécommande. Pour extraire tous les sous-modules du dernier état de la télécommande avec une seule commande, vous pouvez utiliser

```
git submodule foreach git pull <remote> <branch>
```

ou utilisez les arguments par défaut de `git pull`

```
git submodule foreach git pull
```

Notez que cela ne fera que mettre à jour votre copie de travail locale. L'exécution de `git status` indique que le répertoire du sous-module est sale s'il a été modifié à cause de cette commande. Pour mettre à jour votre référentiel afin de référencer le nouvel état à la place, vous devez valider les modifications:

```
git add <submodule_directory>
git commit
```

Si vous utilisez `git pull` pour avoir des conflits de fusion, vous pouvez utiliser `git pull --rebase` pour `git pull --rebase` en `git pull --rebase` sur vos modifications, ce qui diminue le plus souvent les risques de conflit. En outre, il tire toutes les branches vers le local.

```
git submodule foreach git pull --rebase
```

Pour vérifier l'état le plus récent d'un sous-module spécifique, vous pouvez utiliser:

```
git submodule update --remote <submodule_directory>
```

## Définir un sous-module pour suivre une branche

Un sous-module est toujours extrait lors d'un commit spécifique SHA1 (le "gitlink", entrée spéciale dans l'index du rapport parent)

Mais on peut demander de mettre à jour ce sous-module avec la dernière validation d'une branche du dépôt à distance du sous-module.

Plutôt que d'aller dans chaque sous-module, en faisant un `git checkout abranch --track origin/abranche`, `git pull`, vous pouvez simplement faire (à partir du dépôt parent) a:

```
git submodule update --remote --recursive
```

Comme le SHA1 du sous-module changerait, vous devriez toujours suivre cela avec:

```
git add .
git commit -m "update submodules"
```

Cela suppose que les sous-modules étaient:

- soit ajouté avec une branche à suivre:

```
git submodule -b abranche -- /url/of/submodule/repo
```

- ou configuré (pour un sous-module existant) pour suivre une branche:

```
cd /path/to/parent/repo
git config -f .gitmodules submodule.asubmodule.branch abranche
```

## Retrait d'un sous-module

### 1.8

Vous pouvez supprimer un sous - module (par exemple `the_submodule`) en appelant:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` **supprime l'entrée** `the_submodule` **s 'de** `.git / config`. Cela **exclut le** `git submodule update git submodule sync` , **de la** `git submodule sync - git submodule foreach` **et du** `git submodule foreach` **appels et supprime son contenu local** ([source](#)) . En outre, cela ne sera pas affiché comme un changement dans votre référentiel parent. `git submodule init` **et la** `git submodule update` **restaureront le sous-module, là encore sans changements dans votre référentiel parent.**
- `git rm the_submodule` **va supprimer le sous-module de l'arborescence. Les fichiers disparaîtront ainsi que l'entrée des sous-modules dans le fichier** `.gitmodules` ([source](#)) . Si **seulement** `git rm the_submodule` (**sans le** `git submodule deinit the_submodule` **antérieur,** `git submodule deinit the_submodule` **est exécuté, cependant, l'entrée du sous-module dans votre fichier** `.git / config` **restera.**

## 1.8

Tiré d' [ici](#) :

1. **Supprimez la section pertinente du fichier** `.gitmodules` .
2. `.gitmodules` **scène les modifications de** `.gitmodules git add .gitmodules`
3. **Supprimez la section correspondante de** `.git/config` .
4. **Exécutez** `git rm --cached path_to_submodule` (**pas de barre oblique**).
5. **Exécutez** `rm -rf .git/modules/path_to_submodule`
6. **Commit** `git commit -m "Removed submodule <name>"`
7. **Supprimer les fichiers de sous-modules désormais non suivis**
8. `rm -rf path_to_submodule`

## Déplacement d'un sous-module

### 1.8

Courir:

```
$ git mv old/path/to/module new/path/to/module
```

### 1.8

1. Modifiez les `.gitmodules` et modifiez le chemin du sous-module de manière appropriée, puis placez-le dans l'index avec `git add .gitmodules` .
2. Si nécessaire, créez le répertoire parent du nouvel emplacement du sous-module ( `mkdir -p new/path/to` ).
3. Déplacer tout le contenu de l'ancien vers le nouveau répertoire ( `mv -vi old/path/to/module new/path/to/submodule` ).
4. Assurez-vous que Git suit ce répertoire ( `git add new/path /to` ).
5. Supprimez l'ancien répertoire avec `git rm --cached old/path/to/module` .
6. Déplacez le répertoire `.git/modules/ old/path/to/module` avec tout son contenu vers `.git/modules/ new/path/to/module` .

7. Editez le `.git/modules/ new/path/to /config` , assurez-vous que l'élément `worktree` pointe vers les nouveaux emplacements. Dans cet exemple, cela devrait être `worktree = ../../../../ old/path/to/module` . En règle générale, il devrait y en avoir deux `..` puis des répertoires dans le chemin direct à cet endroit. . Editez le fichier `new/path/to/module/.git` , assurez-vous que le chemin dans ce fichier pointe vers le nouvel emplacement correct dans le dossier principal du projet `.git` , donc dans cet exemple `gitdir: ../../../../.git/modules/ new/path/to/module` .

git status sortie d' git status ressemble à ceci après:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   .gitmodules
#       renamed:    old/path/to/submodule -> new/path/to/submodule
#
```

8. Enfin, validez les modifications.

---

Cet exemple de [Stack Overflow](#) , par [Axel Beckert](#)

Lire Des sous-modules en ligne: <https://riptutorial.com/fr/git/topic/306/des-sous-modules>

### Remarques

#### Qu'est-ce qui écrase?

Squashing est le processus qui consiste à prendre plusieurs commits et à les combiner en un seul engagement, en encapsulant toutes les modifications apportées lors des commits initiaux.

#### Branches de squash et à distance

Faites particulièrement attention lorsque vous écrasez une branche sur une branche distante; Si vous écrasez un commit qui a déjà été envoyé sur une branche distante, les deux branches seront divergentes et vous devrez utiliser `git push -f` pour forcer ces modifications sur la branche distante. **Sachez que cela peut entraîner des problèmes pour les autres utilisateurs effectuant le suivi de cette branche distante** . Il convient donc de faire preuve de prudence lors de la compression forcée de validations écrasées sur des référentiels publics ou partagés.

Si le projet est hébergé sur GitHub, vous pouvez activer la "protection contre la poussée" sur certaines branches, comme master , en l'ajoutant à Settings - Branches - Protected Branches .

### Exemples

#### Squash Recent Commits sans relancer

Si vous voulez écraser les x commits précédents en un seul, vous pouvez utiliser les commandes suivantes:

```
git reset --soft HEAD~x
git commit
```

Remplacer x par le nombre de validations précédentes que vous souhaitez inclure dans le commit écrasé.

Rappelez-vous que cela créera un *nouveau* commit, en oubliant essentiellement les informations sur les x commits précédents, x compris leur auteur, leur message et leur date. Vous voulez sans doute *d'abord* copier-coller un message de validation existant.

#### Squashing Commits pendant une rebase

Les validations peuvent être écrasées lors d'un git rebase . Il est recommandé que vous compreniez le [changement](#) avant d'essayer d'écraser les commits de cette manière.

1. Déterminez la validation à partir de laquelle vous souhaitez effectuer une réinitialisation et notez son hachage de validation.
2. Exécutez `git rebase -i [commit hash] .`

Sinon, vous pouvez taper `HEAD~4` au lieu d'un hachage de validation, pour afficher le dernier commit et 4 autres commits avant le dernier.

3. Dans l'éditeur qui s'ouvre lors de l'exécution de cette commande, déterminez les validations que vous souhaitez écraser. Remplacez `pick` au début de ces lignes par `squash` pour les écraser dans le commit précédent.
4. Après avoir sélectionné les validations que vous souhaitez écraser, vous serez invité à écrire un message de validation.

Journalisation des commits pour déterminer où rebaser

```
> git log --oneline
612f2f7 This commit should not be squashed
d84b05d This commit should be squashed
ac60234 Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit

> git rebase -i 36d15de
```

À ce stade, votre éditeur de choix apparaît où vous pouvez décrire ce que vous voulez faire avec les commits. Git fournit de l'aide dans les commentaires. Si vous le laissez tel quel, rien ne se passera car chaque validation sera conservée et leur ordre sera le même qu'avant la rebase. Dans cet exemple, nous appliquons les commandes suivantes:

```
pick ac60234 Yet another commit
squash d84b05d This commit should be squashed
pick 612f2f7 This commit should not be squashed

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Journal Git après avoir écrit le message de validation

```
> git log --oneline
77393eb This commit should not be squashed
e090a8c Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit
```

#### Autosquash: Valider le code que vous voulez écraser lors d'un rebase

Compte tenu de l'histoire suivante, imaginez que vous apportez une modification que vous voulez écraser dans le commit bbb2222 A second commit :

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Une fois que vous avez effectué vos modifications, vous pouvez les ajouter à l'index comme d'habitude, puis les valider en utilisant l'argument `--fixup` avec une référence à la validation dans laquelle vous voulez écraser :

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! A second commit
```

Cela créera un nouveau commit avec un message de validation que Git pourra reconnaître lors d'un rebase interactif :

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! A second commit
ccc3333 A third commit
bbb2222 A second commit
aa1111 A first commit
9999999 Initial commit
```

Ensuite, faites un rebase interactif avec l'argument `--autosquash` :

```
$ git rebase --autosquash --interactive HEAD~4
```

Git vous proposera d'écraser le commit que vous avez fait avec le commit `--fixup` dans la bonne position :

```
pick aa1111 A first commit
pick bbb2222 A second commit
fixup ddd4444 fixup! A second commit
pick ccc3333 A third commit
```

Pour éviter d'avoir à taper `--autosquash` à chaque rebase, vous pouvez activer cette option par défaut :

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

### Squashing Commit Au cours de la fusion

Vous pouvez utiliser `git merge --squash` pour écraser les modifications introduites par une branche dans un seul commit. Aucun engagement ne sera créé.

```
git merge --squash <branch>
git commit
```

Cela équivaut plus ou moins à utiliser `git reset`, mais est plus pratique lorsque les modifications incorporées ont un nom symbolique. Comparer :

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

### Autosquashing et corrections

Lors de la validation des modifications, il est possible de spécifier que la validation sera à l'avenir écrasée par une autre validation et que cela peut être fait comme ça,



```
git commit --squash=[commit hash of commit to which this commit will be squashed to]
```

On pourrait aussi utiliser `--fixup=[commit hash]` alternativement pour corriger.

Il est également possible d'utiliser des mots du message de validation au lieu du hachage de validation, comme cela,

```
git commit --squash :/things
```

où le plus récent engagement avec le mot «choses» serait utilisé.

Le message de ces commits commencerait par 'fixup!' ou 'squash!' suivi du reste du message de validation auquel ces commits seront écrasés.

Lors du `--autosquash` doit être utilisée pour utiliser la fonctionnalité de récupération automatique / correction.

Lire Écrasement en ligne: <https://riptutorial.com/fr/git/topic/598/ecrasement>

### Introduction

Après avoir modifié, transféré et validé du code avec Git, il est nécessaire de lancer des modifications pour mettre vos modifications à la disposition des autres et transférer vos modifications locales sur le serveur de référentiel. Cette rubrique couvrira comment propulser correctement le code en utilisant Git.

### Syntaxe

- `git push [-f | --force] [-v | --verbose] [<remote> [<refspec> ...]]`

### Paramètres

| Paramètre                           | Détails  |
|-------------------------------------|--|
| <code>--Obliger</code>              | Écrase la télécommande pour correspondre à votre référence locale. <i>Le dépôt distant peut perdre ses commits, donc utilisez-le avec précaution .</i> |
| <code>--verbeux</code>              | Courez verbeusement.   |
| <code>&lt;à distance&gt;</code>     | Le référentiel distant qui est la destination de l'opération Push.   |
| <code>&lt;refspec&gt;</code><br>... | Spécifiez la référence distante à mettre à jour avec quel ref ou objet local.  |

### Remarques

#### Amont Aval

En termes de contrôle de source, vous êtes **"en aval"** lorsque vous copiez (clone, extraction, etc.) à partir d'un référentiel. Les informations vous ont été transmises "en aval".

Lorsque vous apportez des modifications, vous souhaitez généralement les renvoyer **"en amont"** afin de les intégrer dans ce référentiel afin que tous ceux qui utilisent la même source travaillent avec les mêmes modifications. Il s'agit principalement d'une question sociale de savoir comment chacun peut coordonner son travail plutôt qu'une exigence technique de contrôle des sources. Vous souhaitez intégrer vos modifications au projet principal afin de ne pas suivre les lignes de développement divergentes.

Parfois, vous lisez sur les gestionnaires de paquets ou de versions (les personnes et non sur l'outil) qui parlent de soumettre des modifications à "en amont". Cela signifie généralement qu'ils devaient ajuster les sources d'origine afin de pouvoir créer un package pour leur système. Ils ne veulent pas continuer à faire ces changements, donc s'ils les envoient "en amont" à la source d'origine, ils ne devraient pas avoir à traiter le même problème dans la prochaine version.

( [Source](#) )

### Exemples

#### Pousser

```
git push
```

va pousser votre code à votre amont existant. Selon la configuration de push, il faudra soit pousser le code de votre branche actuelle (par défaut dans Git 2.x) soit de toutes les branches (par défaut dans Git 1.x).

---

### Spécifiez le référentiel distant

Lorsque vous travaillez avec git, il peut être utile d'avoir plusieurs référentiels distants. Pour spécifier un référentiel distant vers lequel pousser, ajoutez simplement son nom à la commande.

```
git push origin
```

---

### Spécifier la branche

Pour pousser vers une branche spécifique, dites `feature_x` :

```
git push origin feature_x
```

---

### Définir la branche de suivi à distance

À moins que la branche sur laquelle vous travaillez ne provienne d'un dépôt distant, l'utilisation de `git push` ne fonctionnera pas du premier coup. Vous devez exécuter la commande suivante pour indiquer à git de pousser la branche en cours vers une combinaison distante / branche spécifique

```
git push --set-upstream origin master
```

Ici, `master` est le nom de la branche sur l' `origin` distante. Vous pouvez utiliser `-u` comme raccourci pour `--set-upstream` .

---

### Pousser vers un nouveau référentiel

Pour pousser vers un référentiel que vous n'avez pas encore créé ou est vide:

1. Créez le référentiel sur GitHub (le cas échéant)
2. Copiez l'URL qui vous a été donnée sous la forme `https://github.com/USERNAME/REPO_NAME.git`
3. Accédez à votre référentiel local et exécutez `git remote add origin URL`
  - Pour le vérifier, lancez `git remote -v`
4. Exécuter `git push origin master`

Votre code devrait maintenant être sur GitHub

Pour plus d'informations, voir [Ajout d'un référentiel distant](#)

---

### Explication

Le code `push` signifie que git analysera les différences entre vos commits locaux et distants et les enverra pour être écrits en amont. Lorsque `Push` réussit, votre référentiel local et votre référentiel distant sont synchronisés et les autres utilisateurs peuvent voir vos commits.

Pour plus de détails sur les concepts de "amont" et "aval", voir [Remarques](#) .

### Force de poussée

Parfois, lorsque vous avez des modifications locales incompatibles avec les modifications à distance (par exemple, lorsque vous ne pouvez pas avancer rapidement la branche distante ou que la branche distante n'est pas un ancêtre direct de votre branche locale), la seule solution pour pousser vos modifications .

```
git push -f
```

ou

```
git push --force
```

---

### Notes IMPORTANTES

Cela **écrasera** tout changement à distance et votre télécommande correspondra à votre local.

Avertissement: L'utilisation de cette commande peut entraîner une **perte** de validation pour le référentiel distant. De plus, il est fortement déconseillé de forcer si vous partagez ce référentiel distant avec d'autres utilisateurs, car leur historique conservera tous les validations écrasées, rendant ainsi leur travail non synchronisé avec le référentiel distant.

En règle générale, ne forcez que lorsque:

- Personne sauf vous n'a retiré les modifications que vous essayez d'écraser
- Vous pouvez forcer tout le monde à copier une nouvelle copie après la poussée forcée et faire en sorte que tout le monde applique ses modifications (les gens peuvent vous détester pour cela).

### Poussez un objet spécifique vers une branche distante

---

#### Syntaxe générale

```
git push <remotename> <object>:<remotebranchname>
```

#### Exemple

```
git push origin master:wip-yourname
```

Poussez votre branche principale vers la branche d'origine de wip-yourname (la plupart du temps, le référentiel à partir duquel vous avez cloné).

---

#### Supprimer une branche distante

Supprimer la branche distante équivaut à y insérer un objet vide.

```
git push <remotename> :<remotebranchname>
```

#### Exemple

```
git push origin :wip-yourname
```

Va supprimer la branche distante wip-yourname

Au lieu d'utiliser les deux-points, vous pouvez également utiliser l'indicateur `--delete`, qui est plus lisible dans certains cas.

#### Exemple

```
git push origin --delete wip-yourname
```

---

### Poussez un seul engagement

Si vous avez un seul engagement dans votre branche que vous souhaitez envoyer à une télécommande sans rien appuyer, vous pouvez utiliser les éléments suivants:

```
git push <remotename> <commit SHA>:<remotebranchname>
```

#### Exemple

En supposant une histoire de git comme ça

```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```

pour pousser seulement commettre `347d700` à maître distant utiliser la commande suivante

```
git push origin 347d700:master
```

---

### Modification du comportement de push par défaut

**Current** met à jour la branche du référentiel distant qui partage un nom avec la branche de travail en cours.

```
git config push.default current
```

**Simple** pousse à la branche en amont, mais ne fonctionnera pas si la branche en amont est appelée quelque chose d'autre.

```
git config push.default simple
```

**En amont**, la poussée vers la branche en amont, quel que soit l'appel.

```
git config push.default upstream
```

**L'appariement** pousse toutes les branches qui correspondent sur la configuration locale et distante de `git config push.default` en amont

Après avoir défini le style préféré, utilisez

```
git push
```

mettre à jour le référentiel distant.

#### **Balises Push**

```
git push --tags
```

Pousse toutes les git tags du référentiel local qui ne sont pas dans le référentiel distant.

Lire En poussant en ligne: <https://riptutorial.com/fr/git/topic/2600/en-poussant>

### Syntaxe

- # Ne remplacez que les adresses e-mail  
 <primary@example.org> <alias@example.org>
- # Remplacer le nom par adresse email  
 Contributeur <primary@example.org>
- # Fusionner plusieurs alias sous un nom et un email  
 # Notez que cela n'associera pas 'Other <alias2@example.org>'.  
 Contributor <primary@example.org> <alias1@example.org> Contributeur <alias2@example.org>

### Remarques

Un fichier .mailmap peut être créé dans n'importe quel éditeur de texte et ne constitue qu'un fichier texte contenant des noms de contributeurs facultatifs, des adresses électroniques principales et leurs alias. il doit être placé dans la racine du projet, à côté du répertoire .git .

Gardez à l'esprit que cela ne fait que modifier la sortie visuelle de commandes telles que git shortlog ou git log --use-mailmap . Cela **ne** réécrira **pas l'** historique des validations ni n'empêchera les commits avec des noms et / ou des adresses électroniques variables.

Pour empêcher les commits basés sur des informations telles que les adresses e-mail, vous devez plutôt utiliser les [hooks git](#) .

### Exemples

**Fusionner les contributeurs par alias pour afficher le nombre de validations dans le journal des commandes.**

Lorsque des contributeurs ajoutent à un projet à partir de machines ou de systèmes d'exploitation différents, il peut arriver qu'ils utilisent des adresses de messagerie ou des noms différents, ce qui fragmentera les listes de contributeurs et les statistiques.

L'exécution de git shortlog -sn pour obtenir la liste des contributeurs et le nombre de validations qu'ils ont effectuées peuvent entraîner la sortie suivante:

```
Patrick Rothfuss 871
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

Cette fragmentation / dissociation peut être ajustée en fournissant un fichier texte brut .mailmap , contenant des mappages de courrier électronique.

Tous les noms et adresses électroniques figurant dans une ligne seront respectivement associés à la première entité nommée.

Pour l'exemple ci-dessus, un mappage pourrait ressembler à ceci:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Une fois que ce fichier existe dans la racine du projet, git shortlog -sn nouveau git shortlog -sn produira une liste condensée:

```
Patrick Rothfuss 961
```

Lire Fichier .mailmap: Associant contributeur et alias de messagerie en ligne:  
<https://riptutorial.com/fr/git/topic/1270/fichier--mailmap--associant-contributeur-et-alias-de-messagerie>



### Syntaxe

- `git merge another_branch [options]`
- `git fusion --abort`

### Paramètres

| Paramètre                   | Détails  |
|-----------------------------|--|
| <code>-m</code>             | Message à inclure dans la validation de fusion   |
| <code>-v</code>             | Afficher le résultat détaillé  |
| <code>--abort</code>        | Tenter de rétablir tous les fichiers dans leur état                                    |
| <code>--ff-only</code>      | Abandonne instantanément lorsqu'un engagement de fusion serait requis                  |
| <code>--no-ff</code>        | Force la création d'une validation de fusion, même si ce n'est pas obligatoire         |
| <code>--no-commit</code>    | Prétend la fusion a échoué pour permettre l'inspection et le réglage du résultat       |
| <code>--stat</code>         | Afficher un diffstat après la fin de la fusion   |
| <code>-n / --no-stat</code> | Ne pas montrer le diffstat   |
| <code>--squash</code>       | Permet une validation unique sur la branche en cours avec les modifications fusionnées |

### Exemples

#### Fusionner une branche dans une autre

```
git merge incomingBranch
```

Cela fusionne la branche `incomingBranch` dans la branche dans laquelle vous vous trouvez actuellement. Par exemple, si vous êtes actuellement dans `master`, `incomingBranch` sera fusionné dans `master`.

La fusion peut créer des conflits dans certains cas. Si cela se produit, vous verrez le message `Automatic merge failed; fix conflicts and then commit the result`. Vous devrez modifier manuellement les fichiers en conflit ou annuler votre tentative de fusion, exécutez :

```
git merge --abort
```

#### Fusion automatique

Lorsque les commits sur deux branches ne sont pas en conflit, Git peut les fusionner automatiquement :

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
 file_a | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

### Abandonner une fusion

Après avoir lancé une fusion, vous souhaitez peut-être arrêter la fusion et rétablir le tout dans son état de pré-fusion. Utiliser `--abort` :

```
git merge --abort
```

### Garder les changements d'un seul côté d'une fusion

Lors d'une fusion, vous pouvez passer `--ours` ou `--theirs` à `git checkout` de prendre toutes les modifications d'un fichier d'un côté ou de l'autre d'une fusion.

```
$ git checkout --ours -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

### Fusionner avec un commit

Le comportement par défaut est lorsque la fusion se résout en une accélération rapide, ne met à jour que le pointeur de branche, sans créer de validation de fusion. Utilisez `--no-ff` pour résoudre le problème.

```
git merge <branch_name> --no-ff -m "<commit message>"
```

### Recherche de toutes les branches sans modifications fusionnées

Parfois, il se peut que des branches soient déjà présentes et que leurs modifications aient déjà été fusionnées dans Master. Cela permet de trouver toutes les branches qui ne sont pas master qui n'ont pas de commits uniques par rapport à master. Ceci est très utile pour trouver des branches qui n'ont pas été supprimées après la fusion du PR avec master.

```
for branch in $(git branch -r) ; do
  [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] &&
  echo -e `git show --pretty=format:@"%ci %cr" $branch | head -n 1`\t$branch
done | sort -r
```

Lire Fusion en ligne: <https://riptutorial.com/fr/git/topic/291/fusion>

## Chapitre 19: Fusion externe et difftools

### Exemples

#### Mise en place au-delà de la comparaison

Vous pouvez définir le chemin d'accès à bcomp.exe

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

et configurez bc3 par défaut

```
git config --global diff.tool bc3
```

#### Configurer KDiff3 comme outil de fusion

Les éléments suivants doivent être ajoutés à votre fichier global .gitconfig

```
[merge]
  tool = kdiff3
[mergetool "kdiff3"]
  path = D:/Program Files (x86)/KDiff3/kdiff3.exe
  keepBackup = false
  keepbackup = false
  trustExitCode = false
```

N'oubliez pas de définir la propriété path pour pointer vers le répertoire où vous avez installé KDiff3

#### Configurer KDiff3 comme outil de diff

```
[diff]
  tool = kdiff3
  guitool = kdiff3
[difftool "kdiff3"]
  path = D:/Program Files (x86)/KDiff3/kdiff3.exe
  cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"\$LOCAL\" \"\$REMOTE\"
```

#### Configuration d'un IDE IntelliJ en tant qu'outil de fusion (Windows)

```
[merge]
  tool = intellij
[mergetool "intellij"]
  cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
\"$LOCAL\") && pwd)/$(basename \"$LOCAL\") $(cd $(dirname \"$REMOTE\") && pwd)/$(basename \"$REMOTE\")
$(cd $(dirname \"$BASE\") && pwd)/$(basename \"$BASE\") $(cd $(dirname \"$MERGED\") &&
pwd)/$(basename \"$MERGED\")\"
  keepBackup = false
  keepbackup = false
  trustExitCode = true
```

La seule chose à faire ici est que cette propriété cmd n'accepte aucun caractère bizarre dans le chemin. Si l'emplacement d'installation de votre IDE contient des caractères étranges (par exemple, il est installé dans Program Files (x86) , vous devrez créer un lien symbolique.

## Configurer un IDE IntelliJ en tant qu'outil de diff (Windows)

```
[diff]
  tool = intellij
  guitool = intellij
[difftool "intellij"]
  path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
  cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename
"$REMOTE")\"
```

La seule chose à faire ici est que cette propriété cmd n'accepte aucun caractère bizarre dans le chemin. Si l'emplacement d'installation de votre IDE contient des caractères étranges (par exemple, il est installé dans Program Files (x86) , vous devrez créer un lien symbolique.

Lire Fusion externe et difftools en ligne: <https://riptutorial.com/fr/git/topic/5972/fusion-externe-et-difftools>

---

## Chapitre 20: GFS Large File Storage (LFS)

### Remarques

Le stockage de fichiers volumineux de Git (LFS) vise à éviter une limitation du système de contrôle de version de Git. LFS résout ce problème en stockant le contenu de ces fichiers sur un serveur externe, puis en validant simplement un pointeur de texte sur le chemin de ces actifs dans la base de données des objets git.

Les types de fichiers communs stockés via LFS ont tendance à être compilés en source; des actifs graphiques, tels que les fichiers PSD et JPEG; ou des actifs 3D. De cette manière, les ressources utilisées par les projets peuvent être gérées dans le même référentiel, plutôt que de devoir gérer un système de gestion séparé en externe.

LFS a été initialement développé par GitHub ( <https://github.com/blog/1986-announcing-git-large-file-storage-lfs> ) ; Cependant, Atlassian travaillait sur un projet similaire presque au même moment, appelé `git-lob` . Bientôt, ces efforts ont été fusionnés pour éviter la fragmentation du secteur.

### Exemples

#### Installer LFS

Téléchargez et installez, soit via Homebrew, soit depuis un [site Web](#) .

```
Pour Brew,  
brew install git-lfs  
git lfs install
```

Souvent, vous devrez également effectuer une configuration sur le service qui héberge votre télécommande pour lui permettre de fonctionner avec lfs. Ce sera différent pour chaque hôte, mais il suffira probablement de cocher une case indiquant que vous souhaitez utiliser git lfs.

#### Déclarez certains types de fichiers à stocker en externe

Un flux de travail courant pour l'utilisation de Git LFS consiste à déclarer quels fichiers sont interceptés via un système basé sur des règles, tout comme les fichiers `.gitignore` .

La plupart du temps, les caractères génériques sont utilisés pour sélectionner certains types de fichiers à masquer.

```
Par exemple, git lfs track "*.psd"
```

Lorsqu'un fichier correspondant au modèle ci-dessus est ajouté aux fichiers validés, il est ensuite envoyé séparément à la télécommande, avec un pointeur remplaçant le fichier dans le référentiel distant.

Après qu'un fichier a été suivi avec lfs, votre fichier `.gitattributes` sera mis à jour en conséquence. Github recommande de `.gitattributes` votre fichier local `.gitattributes` , plutôt que de travailler avec un fichier global `.gitattributes` , pour vous assurer que vous ne rencontrez aucun problème lorsque vous travaillez avec différents projets.

#### Définir la configuration de LFS pour tous les clones

Pour définir les options LFS qui s'appliquent à tous les clones, créez et `.lfsconfig` un fichier nommé `.lfsconfig` à la racine du référentiel. Ce fichier peut spécifier les options LFS de la même manière que le permet `.git/config` .

Par exemple, pour exclure un fichier de récupération LFS, créez et `.lfsconfig` avec le contenu suivant:

```
[lfs]
  fetchexclude = ReallyBigFile.wav
```

Lire GFS Large File Storage (LFS) en ligne: <https://riptutorial.com/fr/git/topic/4136/gfs-large-file-storage--lfs->

## Chapitre 21: Git Clean

### Syntaxe

- `git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>`

### Paramètres

| Paramètre        | Détails   |
|------------------|---|
| -ré              | Supprimez les répertoires non suivis en plus des fichiers non suivis. Si un répertoire non suivi est géré par un référentiel Git différent, il n'est pas supprimé par défaut. Utilisez l'option -f deux fois si vous voulez vraiment supprimer un tel répertoire.   |
| -f, --force      | Si la variable de configuration Git est propre. <code>requireForce</code> n'est pas défini sur false, git clean refusera de supprimer des fichiers ou des répertoires à moins de fournir -f, -n ou -i. Git refusera de supprimer des répertoires avec un sous-répertoire ou un fichier .git à moins qu'un deuxième -f soit donné. |
| -i, --interactif | Invite de manière interactive la suppression de chaque fichier.   |
| -n, --dry-run    | Affiche uniquement une liste de fichiers à supprimer sans les supprimer.  |
| -q, - calme      | Afficher uniquement les erreurs, pas la liste des fichiers supprimés avec succès.   |

### Exemples

#### Nettoyer les fichiers ignorés

```
git clean -fX
```

Supprime tous les fichiers **ignorés** du répertoire en cours et de tous les sous-répertoires.

```
git clean -Xn
```

Prévisualisera tous les fichiers qui seront nettoyés.

#### Nettoyer tous les répertoires non suivis

```
git clean -fd
```

Va supprimer tous les répertoires non suivis et les fichiers qu'ils contiennent. Il démarrera dans le répertoire de travail actuel et parcourra tous les sous-répertoires.

```
git clean -dn
```

Prévisualisera tous les répertoires qui seront nettoyés.

### Supprimer avec force les fichiers non suivis

```
git clean -f
```

Va supprimer tous les fichiers non suivis.

### Nettoyer interactivement

```
git clean -i
```

Imprimera les éléments à supprimer et demandera une confirmation via des commandes telles que les suivantes:

```
Would remove the following items:
  folder/file1.py
  folder/file2.py
*** Commands ***
  1: clean      2: filter by pattern    3: select by numbers    4: ask each
  5: quit      6: help
What now>
```

Option interactive i peut être ajouté en même temps que d' autres options comme X , d , etc.

Lire Git Clean en ligne: <https://riptutorial.com/fr/git/topic/1254/git-clean>



### Syntaxe

- `git diff [options] [<commit>] [--] [<path>...]`
  - `git diff [options] --cached [<commit>] [--] [<path>...]`
  - `git diff [options] <commit> <commit> [--] [<path>...]`
  - `git diff [options] <blob> <blob>`
  - `git diff [options] [--no-index] [--] <path> <path>`

### Paramètres

| Paramètre                       | Détails  |
|---------------------------------|--|
| <code>-p, -u, --patch</code>    | Générer un patch   |
| <code>-s, --no-patch</code>     | Supprimer la sortie diff. Utile pour les commandes comme <code>git show</code> qui affichent le patch par défaut ou pour annuler l'effet de <code>--patch</code>   |
| <code>--brut</code>             | Générer le diff au format brut   |
| <code>--diff-algorithm =</code> | Choisissez un algorithme de diff. Les variantes sont les suivantes: <code>myers</code> , <code>minimal</code> , <code>patience</code> , <code>histogram</code>   |
| <code>--résumé</code>           | Générer un résumé condensé des informations d'en-tête étendues telles que les créations, les renommage et les changements de mode  |
| <code>--nom uniquement</code>   | Afficher uniquement les noms des fichiers modifiés   |
| <code>--name-status</code>      | Afficher les noms et les statuts des fichiers modifiés Les statuts les plus courants sont M (modifié), A (ajouté) et D (supprimé)  |
| <code>--vérifier</code>         | Avertir si des modifications introduisent des marqueurs de conflit ou des erreurs d'espacement. Ce qui est considéré comme des espaces blancs est contrôlé par la configuration <code>core.whitespace</code> . Par défaut, les espaces blancs finaux (y compris les lignes composées uniquement d'espaces blancs) et un caractère d'espace immédiatement suivi d'un caractère de tabulation à l'intérieur du retrait initial de la ligne sont considérés comme des espaces. Exit avec un statut non nul si des problèmes sont détectés. Non compatible avec <code>--exit-code</code> |
| <code>--full-index</code>       | Au lieu de la première poignée de caractères, affichez les noms complets des objets blob avant et après l'image sur la ligne "index" lors de la génération d'une sortie au format patch.   |
| <code>--binaire</code>          | En plus de <code>--full-index</code> , <code>--full-index</code> un diff binaire pouvant être appliqué avec <code>git apply</code>   |

| Paramètre | Détails  |
|-----------|--|
| -un texte | Traitez tous les fichiers en tant que texte.   |
| --Couleur | Définir le mode couleur; c'est-à-dire utiliser <code>--color=always</code> si vous souhaitez réduire le diff et garder la couleur de git |

### Exemples

#### Afficher les différences dans la branche de travail

```
git diff
```

Cela montrera les modifications non *planifiées* sur la branche en cours à partir de la validation précédente. Il ne montrera que les changements relatifs à l'index, ce qui signifie qu'il montre ce que vous pouvez ajouter au prochain commit, mais que ce n'est pas le cas. Pour ajouter (mettre en scène) ces modifications, vous pouvez utiliser `git add`.

Si un fichier est mis en scène, mais a été modifié après sa mise en place, `git diff` affiche les différences entre le fichier actuel et la version intermédiaire.

#### Afficher les différences pour les fichiers mis en scène

```
git diff --staged
```

Cela montrera les changements entre le commit précédent et les fichiers actuellement mis en scène.

**Remarque:** vous pouvez également utiliser les commandes suivantes pour accomplir la même chose:

```
git diff --cached
```

Qui est juste un synonyme de `--staged` ou

```
git status -v
```

Ce qui déclenchera les paramètres verbeux de la commande `status`.

#### Afficher à la fois les changements mis en scène et non modifiés

Pour afficher tous les changements par étapes ou non, utilisez:

```
git diff HEAD
```

**REMARQUE:** vous pouvez également utiliser la commande suivante:

```
git status -vv
```

La différence étant que la sortie de ce dernier vous indiquera en fait quelles modifications sont mises en attente pour la validation et lesquelles ne le sont pas.

#### Afficher les changements entre deux commits

```
git diff 1234abc..6789def # old new
```

Ex: Afficher les modifications apportées aux 3 derniers commits:

```
git diff @~3..@ # HEAD -3 HEAD
```

Remarque: les deux points (..) sont facultatifs, mais ajoutent de la clarté.

Cela montrera la différence textuelle entre les commits, indépendamment de l'endroit où ils se trouvent dans l'arbre.

**Utiliser meld pour voir toutes les modifications dans le répertoire de travail**

```
git difftool -t meld --dir-diff
```

affichera les modifications du répertoire de travail. Alternativement,

```
git difftool -t meld --dir-diff [COMMIT_A] [COMMIT_B]
```

montrera les différences entre 2 commits spécifiques.

**Afficher les différences pour un fichier ou un répertoire spécifique**

```
git diff myfile.txt
```

Affiche les modifications entre la validation précédente du fichier spécifié ( myfile.txt ) et la version modifiée localement qui n'a pas encore été mise en place.

Cela fonctionne également pour les répertoires:

```
git diff documentation
```

Ce qui précède montre les changements entre la validation précédente de tous les fichiers du répertoire spécifié ( documentation/ ) et les versions modifiées localement de ces fichiers, qui n'ont pas encore été transférés.

Pour montrer la différence entre une version d'un fichier dans un commit donné et la version HEAD locale, vous pouvez spécifier la validation à comparer:

```
git diff 27fa75e myfile.txt
```

Ou si vous voulez voir la version entre deux commits distincts:

```
git diff 27fa75e ada9b57 myfile.txt
```

Pour afficher la différence entre la version spécifiée par le hachage ada9b57 et la dernière validation de la branche my\_branchname pour le seul répertoire relatif appelé my\_changed\_directory/ vous pouvez le faire:

```
git diff ada9b57 my_branchname my_changed_directory/
```

**Affichage d'un mot-diff pour les longues lignes**

```
git diff [HEAD|--staged...] --word-diff
```

Plutôt que d'afficher les lignes modifiées, cela affichera les différences entre les lignes. Par exemple, plutôt que:

```
-Hello world
+Hello world!
```

Lorsque la ligne entière est marquée comme modifiée, le word-diff modifie le résultat en:

```
Hello [-world-]{+world!+}
```

Vous pouvez omettre les marqueurs [- , -] , {+ , +} en spécifiant --word-diff=color ou --color-words . Cela utilisera uniquement le codage couleur pour marquer la différence:

```
@@ -1 +1 @@
Hello worldworld!
```

**Affichage d'une fusion à trois voies, y compris l'ancêtre commun**

```
git config --global merge.conflictstyle diff3
```

Définit le style diff3 par défaut: au lieu du format habituel dans les sections en conflit, affichant les deux fichiers:

```
<<<<<< HEAD
left
=====
right
>>>>>> master
```

il comprendra une section supplémentaire contenant le texte original (provenant de l'ancêtre commun):

```
<<<<<< HEAD
first
second
|||||
first
=====
last
>>>>>> master
```

Ce format facilite la compréhension des conflits de fusion, c.-à-d. dans ce cas , localement second a été ajoutée, alors que changé à distance d' first à last , à résoudre:

```
last
second
```

La même résolution aurait été beaucoup plus difficile avec la valeur par défaut:

```
<<<<<< HEAD
first
second
=====
```

```
last
>>>>>> master
```

### Afficher les différences entre la version actuelle et la dernière version

```
git diff HEAD^ HEAD
```

Cela montrera les changements entre le commit précédent et le commit en cours.

### Fichiers texte codés UTF-16 Diff et fichiers plistes binaires

Vous pouvez modifier les fichiers encodés en UTF-16 (les fichiers de chaînes de localisation d'OS iOS et MacOS en sont des exemples) en spécifiant comment git doit différer ces fichiers.

Ajoutez ce qui suit à votre fichier ~/.gitconfig .

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

iconv est un programme pour [convertir différents encodages](#) .

Ensuite, modifiez ou créez un fichier .gitattributes à la racine du référentiel où vous souhaitez l'utiliser. Ou modifiez simplement ~/.gitattributes .

```
*.strings diff=utf16
```

Cela convertira tous les fichiers se terminant par .strings avant que git diffs.

Vous pouvez faire des choses similaires pour d'autres fichiers, qui peuvent être convertis en texte.

Pour les fichiers .gitconfig binaires, vous devez modifier .gitconfig

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

et .gitattributes

```
*.plist diff=plist
```

### Comparer les branches

Montrer les changements entre la pointe du **new** et la pointe de l' **original** :

```
git diff original new      # equivalent to original..new
```

Afficher tous les changements sur le **new** depuis qu'il est dérivé de l' **original** :

```
git diff original...new    # equivalent to $(git merge-base original new)..new
```

En utilisant un seul paramètre tel que

```
git diff original
```

est équivalent à

```
git diff original..HEAD
```

#### Afficher les changements entre deux branches

```
git diff branch1..branch2
```

#### Produire un diff compatible avec les patches

Parfois, vous avez juste besoin d'un diff pour appliquer à l'aide de patch. Le `git --diff` ne fonctionne pas. Essayez plutôt ceci:

```
git diff --no-prefix > some_file.patch
```

Ensuite, ailleurs, vous pouvez l'inverser:

```
patch -p0 < some_file.patch
```

#### différence entre deux commit ou branch

Pour voir la différence entre deux branches

```
git diff <branch1>..<branch2>
```

Pour voir la différence entre deux branches

```
git diff <commitId1>..<commitId2>
```

Pour voir diff avec branche actuelle

```
git diff <branch/commitId>
```

Pour afficher le résumé des modifications

```
git diff --stat <branch/commitId>
```

Pour afficher les fichiers modifiés après une certaine validation

```
git diff --name-only <commitId>
```

Pour afficher des fichiers différents d'une branche

```
git diff --name-only <branchName>
```

Pour afficher les fichiers modifiés dans un dossier après une certaine validation

```
git diff --name-only <commitId> <folder_path>
```

Lire Git Diff en ligne: <https://riptutorial.com/fr/git/topic/273/git-diff>

## Chapitre 23: git envoyer-email

### Syntaxe

- `git send-email [options] <fichier | répertoire | options de liste de révocation>...`
- `git send-email --dump-aliases`

### Remarques

<https://git-scm.com/docs/git-send-email>

### Exemples

#### Utiliser git send-email avec Gmail

Contexte: si vous travaillez sur un projet tel que le noyau Linux, plutôt que de faire une demande d'extraction, vous devrez soumettre vos commits à un serveur de liste pour examen. Cette entrée explique comment utiliser le courrier électronique git-send avec Gmail.

Ajoutez ce qui suit à votre fichier `.gitconfig`:

```
[sendemail]
  smtpserver = smtp.googlemail.com
  smtpencryption = tls
  smtpserverport = 587
  smtpuser = name@gmail.com
```

Ensuite, sur le Web: Accédez à Google -> Mon compte -> Applications et sites connectés -> Autoriser les applications moins sécurisées -> Activer

Pour créer un ensemble de patchs:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Envoyez ensuite les correctifs à un serveur de liste:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

Pour créer et envoyer une version mise à jour (version 2 dans cet exemple) du correctif:

```
git format-patch -v 2 HEAD~~~~ .....
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

### Composition

```
--de * Email De: - [no-] à * Email A: - [no-] cc * Email Cc: - [no-] bcc * Email Bcc: --subject
* Email "Subject:" - -in-reply-to * Email "In-Reply-To:" - [no-] xmailer * Ajouter l'en-tête "X-
Mailer:" (par défaut). - [no-] annoter * Passez en revue chaque patch qui sera envoyé dans un
éditeur. --compose * Ouvre un éditeur pour l'introduction. --compose-encoding * Encodage à
prendre en compte pour l'introduction. --8bit-encoding * Encodage pour assumer des mails de 8
bits si non déclaré - transfert-encodage * Encodage de transfert à utiliser (guidé-imprimable, 8
bits, base64)
```

### Envoi de patchs par mail

Supposons que vous ayez beaucoup de validation contre un projet (ici `ulogd2`, la branche

officielle est git-svn) et que vous souhaitiez envoyer votre patch à la liste de diffusion devel@netfilter.org. Pour ce faire, ouvrez simplement un shell à la racine du répertoire git et utilisez:

```
git format-patch --stat -p --raw --signoff --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n
git-svn
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

La première commande créera une série de messages à partir des correctifs dans / tmp / ulogd2 / avec le rapport statistique et ensuite démarrera votre éditeur pour composer un courrier d'introduction au patchset. Pour éviter les séries de mails affreuses, on peut utiliser:

```
git config sendemail.chainreplyto false
```

[la source](#)

Lire [git envoyer-email en ligne](https://riptutorial.com/fr/git/topic/4821/git-envoyer-email): <https://riptutorial.com/fr/git/topic/4821/git-envoyer-email>



### Exemples

#### GitHub Desktop

Site Web: <https://desktop.github.com>

Prix: gratuit

Plateformes: OS X et Windows

Développé par: [GitHub](#)

#### Git Kraken

Site Web: <https://www.gitkraken.com>

Prix: 60 \$ / an (gratuit pour Pour open source, éducation, à but non lucratif, startups ou usage personnel)

Plateformes: Linux, OS X, Windows

Développé par: [Axosoft](#)

#### SourceTree

Site Web: <https://www.sourcetreeapp.com>

Prix: gratuit (compte nécessaire)

Plateformes: OS X et Windows

Développeur: [Atlassian](#)

### gitk et git-gui

Lorsque vous installez Git, vous obtenez également ses outils visuels, gitk et git-gui.

gitk est un visualiseur graphique. Pensez-y comme une puissante interface graphique sur git log et git grep. C'est l'outil à utiliser lorsque vous essayez de trouver quelque chose qui s'est passé par le passé ou de visualiser l'historique de votre projet.

Gitk est le plus facile à appeler depuis la ligne de commande. Juste cd dans un dépôt Git, et tapez:

```
$ gitk [git log options]
```

Gitk accepte de nombreuses options de ligne de commande, dont la plupart sont transmises à l'action git log sous-jacente. Probablement l'un des plus utiles est le drapeau --all , qui dit à gitk de montrer les commits accessibles depuis n'importe quelle ref, pas seulement HEAD. L'interface de Gitk ressemble à ceci:

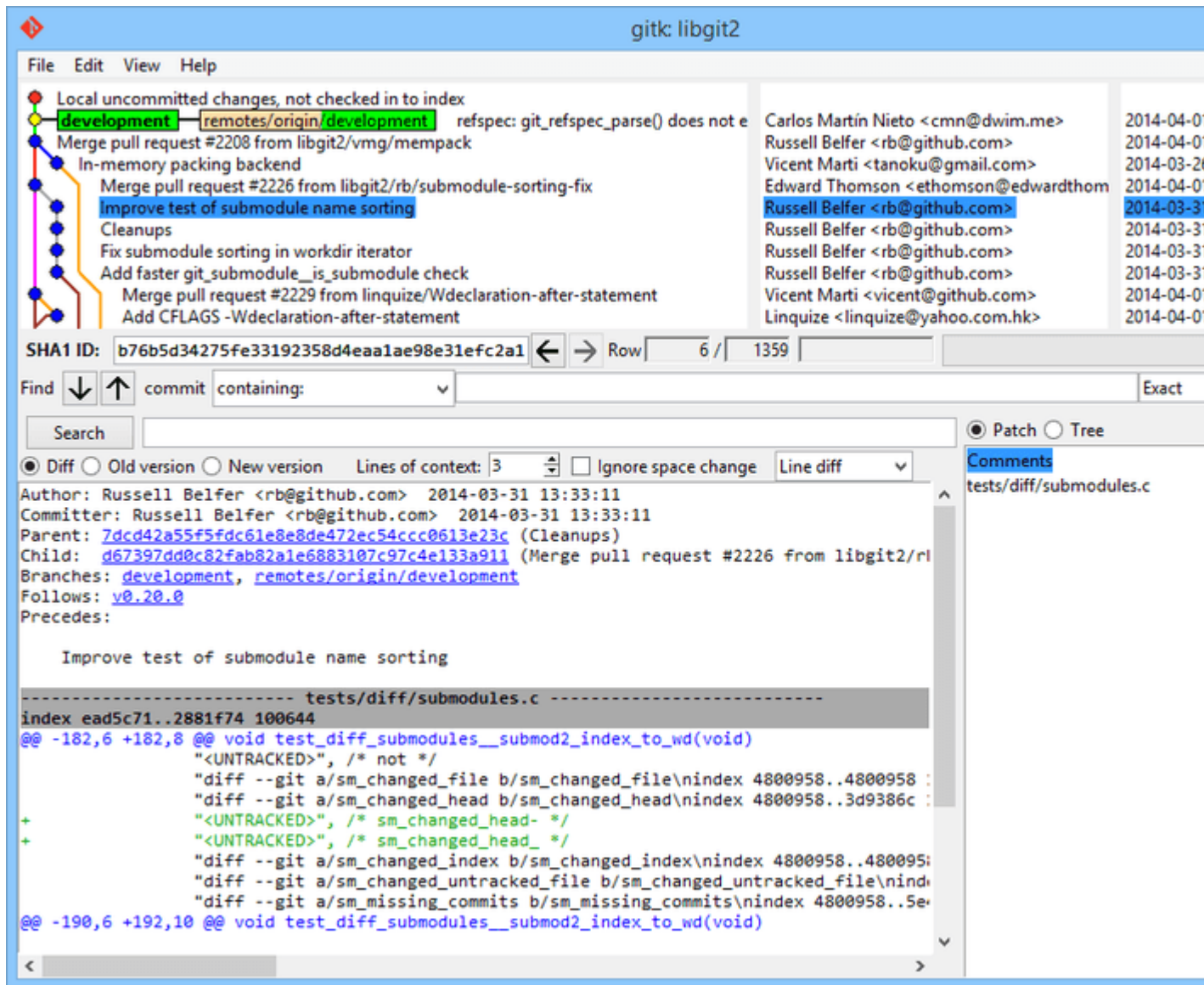


Figure 1-1. Le visualiseur d'historique gitk.

En haut, quelque chose ressemble un peu à la sortie de `git log --graph`; chaque point représente un commit, les lignes représentent des relations parentes et les références sont affichées sous forme de cases colorées. Le point jaune représente HEAD et le point rouge représente les changements qui doivent encore être validés. En bas se trouve une vue du commit sélectionné; les commentaires et le patch à gauche, et une vue récapitulative à droite. Entre les deux est une collection de contrôles utilisés pour rechercher l'historique.

Vous pouvez accéder à de nombreuses fonctions liées à git via un clic droit sur un nom de branche ou un message de validation. Par exemple, il est facile de vérifier une branche ou une sélection de cerises différentes en un clic.

`git-gui`, en revanche, est avant tout un outil de création de commits. Il est également plus facile d'appeler depuis la ligne de commande:

```
$ git gui
```

Et ça ressemble à ceci:

L'outil de validation de `git-gui`.

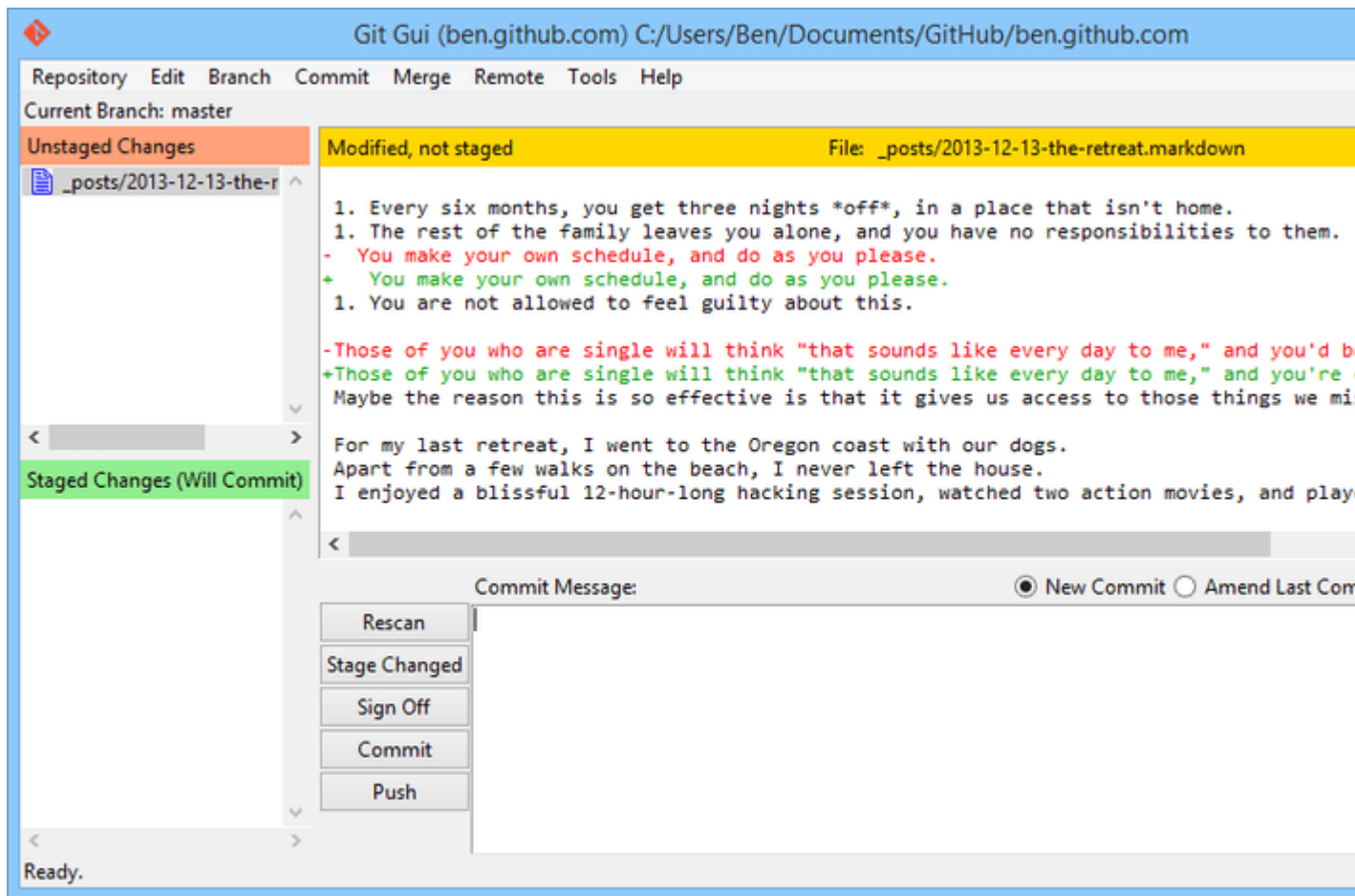


Figure 1-2. L'outil de validation de git-gui.

À gauche est l'index; les changements non planifiés sont en haut, les changements mis en scène en bas. Vous pouvez déplacer des fichiers entiers entre les deux états en cliquant sur leurs icônes, ou vous pouvez sélectionner un fichier à afficher en cliquant sur son nom.

En haut à droite se trouve l'affichage des différences, qui affiche les modifications pour le fichier actuellement sélectionné. Vous pouvez mettre en scène des pièces individuelles (ou des lignes individuelles) en cliquant avec le bouton droit de la souris dans cette zone.

En bas à droite se trouve la zone de message et d'action. Tapez votre message dans la zone de texte et cliquez sur «Commit» pour faire quelque chose de similaire à git commit. Vous pouvez également choisir de modifier le dernier engagement en sélectionnant le bouton radio «Modifier», qui mettra à jour la zone «Modifications par étapes» avec le contenu du dernier engagement. Ensuite, vous pouvez simplement mettre en scène ou décompresser certaines modifications, modifier le message de validation et cliquer à nouveau sur «Valider» pour remplacer l'ancien par un nouveau.

gitk et git-gui sont des exemples d'outils orientés tâche. Chacun d'entre eux est adapté à un objectif spécifique (affichage de l'historique et création de validations, respectivement) et omet les fonctionnalités non nécessaires à cette tâche.

**Source:** <https://git-scm.com/book/en/v2/Git-in-Ather-Environments-Graphical-Interfaces>

#### SmartGit

Site Web: <http://www.syntevo.com/smartgit/>

Prix: Gratuit pour un usage non commercial uniquement. Une licence perpétuelle coûte 99 USD

Plateformes: Linux, OS X, Windows

Développé par: [syntevo](http://www.syntevo.com)

## Extensions Git

Site Web: <https://gitextensions.github.io>

Prix: gratuit

Plate-forme: Windows

Lire Git GUI Clients en ligne: <https://riptutorial.com/fr/git/topic/5148/git-gui-clients>

### Syntaxe

- `git remote [-v | --verbose]`
  - `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>]<name> <url>`
  - `git remote rename <old> <new>`
  - `git remote remove <name>`
  - `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
  - `git remote set-branches [--add] <name> <branch>...`
  - `git remote set-url [--push] <name> <newurl> [<oldurl>]`
  - `git remote set-url --add [--push] <name> <newurl>`
  - `git remote set-url --delete [--push] <name> <url>`
  - `git remote [-v | --verbose] show [-n] <name>...`
  - `git remote prune [-n | --dry-run] <name>...`
  - `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]`
  - `git remote show <name>`

### Paramètres

| Paramètre           | Détails  |
|---------------------|--|
| -v, --verbose       | Courez verbeusement.   |
| -m <maître>         | Définit la tête sur la branche <master> de la télécommande   |
| --mirror = chercher | Les références ne seront pas stockées dans les espaces de noms refs / remotes, mais seront reflétées dans le référentiel local |
| --mirror = pousser  | <code>git push</code> se comportera comme si --mirror était passé  |
| --no-tags           | <code>git fetch &lt;name&gt;</code> n'importe pas les balises du repo distant  |
| -t <branche>        | Spécifie la télécommande à suivre <i>uniquement</i> <branch>   |
| -F                  | <code>git fetch &lt;name&gt;</code> est exécuté immédiatement après la configuration de remote                                 |
| --Mots clés         | <code>git fetch &lt;name&gt;</code> importe chaque balise du repo distant  |
| -a, --auto          | Le HEAD de la référence symbolique est défini sur la même branche que le HEAD de la télécommande.                              |
| -d, --delete        | Toutes les références répertoriées sont supprimées du référentiel distant  |
| --ajouter           | Ajoute <name> à la liste des branches actuellement suivies (set-branches)  |
| --ajouter           | Au lieu de changer une URL, une nouvelle URL est ajoutée (set-url)   |
| --tout              | Poussez toutes les branches.   |

| Paramètre              | Détails   |
|------------------------|---|
| <code>--effacer</code> | Toutes les URL correspondant à <code>&lt;url&gt;</code> sont supprimées. ( <code>set-url</code> )   |
| <code>--pousser</code> | Les URL Push sont manipulées à la place des URL de récupération   |
| <code>-n</code>        | Les têtes distantes ne sont pas interrogées en premier avec <code>git ls-remote &lt;name&gt;</code> , les informations en cache sont utilisées à la place |
| <code>--dry-run</code> | rapporter quelles branches seront élaguées, mais ne les élaguent pas réellement   |
| <code>--prune</code>   | Supprimer les branches distantes qui ne possèdent pas d'homologue local   |

### Exemples

#### Ajouter un référentiel distant

Pour ajouter une télécommande, utilisez `git remote add` dans la racine de votre référentiel local.

Pour ajouter un référentiel Git distant `<url>` comme nom abrégé simple `<name>`, utilisez

```
git remote add <name> <url>
```

La commande `git fetch <name>` peut ensuite être utilisée pour créer et mettre à jour des branches de suivi à distance `<name>/<branch>`.

#### Renommer un référentiel distant

Renommez la télécommande nommée `<old>` en `<new>`. Toutes les branches de suivi à distance et les paramètres de configuration de la télécommande sont mis à jour.

Pour renommer un nom de la branche à distance `dev` à `dev1` :

```
git remote rename dev dev1
```

#### Supprimer un référentiel distant

Supprimez la télécommande nommée `<name>`. Toutes les branches de suivi à distance et les paramètres de configuration de la télécommande sont supprimés.

Pour supprimer un dev référentiel distant :

```
git remote rm dev
```

#### Afficher les dépôts distants

Pour répertorier tous les référentiels distants configurés, utilisez `git remote`.

Il indique le nom abrégé (alias) de chaque handle distant que vous avez configuré.

```
$ git remote
premium
```

```
premiumPro
origin
```

Pour afficher des informations plus détaillées, l' `--verbose` ou `-v` peut être utilisée. La sortie inclura l'URL et le type de la télécommande ( `push` ou `pull` ):

```
$ git remote -v
premiumPro https://github.com/user/CatClickerPro.git (fetch)
premiumPro https://github.com/user/CatClickerPro.git (push)
premium https://github.com/user/CatClicker.git (fetch)
premium https://github.com/user/CatClicker.git (push)
origin https://github.com/ud/starter.git (fetch)
origin https://github.com/ud/starter.git (push)
```

### Changer l'URL distante de votre dépôt Git

Vous pouvez le faire si le référentiel distant est migré. La commande de modification de l'URL distante est la suivante:

```
git remote set-url
```

Il prend 2 arguments: un nom distant existant (origine, amont) et l'URL.

Vérifiez votre URL distante actuelle:

```
git remote -v
origin https://bitbucket.com/develop/myrepo.git (fetch)
origin https://bitbucket.com/develop/myrepo.git (push)
```

Changez votre URL distante:

```
git remote set-url origin https://localhost/develop/myrepo.git
```

Vérifiez à nouveau votre URL distante:

```
git remote -v
origin https://localhost/develop/myrepo.git (fetch)
origin https://localhost/develop/myrepo.git (push)
```

### Afficher plus d'informations sur le référentiel distant

Vous pouvez afficher plus d'informations sur un référentiel distant à l'aide de `git remote show <remote repository alias>`

```
git remote show origin
```

résultat:

```
remote origin
Fetch URL: https://localhost/develop/myrepo.git
Push URL: https://localhost/develop/myrepo.git
HEAD branch: master
Remote branches:
  master      tracked
```

```
Local branches configured for 'git pull':  
  master      merges with remote master  
Local refs configured for 'git push':  
  master      pushes to master      (up to date)
```

Lire Git Remote en ligne: <https://riptutorial.com/fr/git/topic/4071/git-remote>



---

## Chapitre 26: Git rerere

### Introduction

rerere (résolution de réutilisation enregistrée) vous permet de dire à git de vous rappeler comment vous avez résolu un conflit de taille. Cela permet de le résoudre automatiquement la prochaine fois que git rencontre le même conflit.

### Exemples

#### Activation de la rereere

Pour activer rerere exécutez la commande suivante:

```
$ git config --global rerere.enabled true
```

Cela peut être fait dans un référentiel spécifique ainsi que dans le monde entier.

Lire Git rerere en ligne: <https://riptutorial.com/fr/git/topic/9156/git-rerere>

---

## Chapitre 27: git-svn

### Remarques

#### Cloner de très gros référentiels SVN

Si votre historique de dépôt SVN est vraiment très important, cette opération pourrait prendre des heures, car git-svn a besoin de reconstruire l'historique complet du dépôt SVN. Heureusement, il vous suffit de cloner le dépôt SVN une seule fois. Comme avec tout autre dépôt git, vous pouvez simplement copier le dossier repo sur d'autres collaborateurs. La copie du dossier sur plusieurs ordinateurs sera plus rapide que le simple clonage de gros repos SVN.

#### A propos des commits et SHA1

Vos commits git locaux seront *réécrits* lors de l'utilisation de la commande `git svn dcommit`. Cette commande ajoutera un texte au message de commit git faisant référence à la révision SVN créée dans le serveur SVN, ce qui est très utile. Cependant, l'ajout d'un nouveau texte nécessite de modifier le message d'un commit existant, ce qui ne peut pas réellement être fait: git commits est immuable. La solution est de créer un nouveau commit avec le même contenu et le nouveau message, mais techniquement c'est un nouveau commit (c'est-à-dire que le SHA1 du commit git changera)

Comme les commits git créés pour git-svn sont locaux, les identifiants SHA1 pour les commits git sont différents entre chaque dépôt git! Cela signifie que vous ne pouvez pas utiliser un SHA1 pour référencer un commit d'une autre personne car le même commit aura un SHA1 différent dans chaque référentiel git local. Vous devez vous appuyer sur le numéro de révision svn ajouté au message de validation lorsque vous transmettez au serveur SVN si vous souhaitez référencer une validation entre différentes copies du référentiel.

Vous pouvez utiliser le SHA1 pour les opérations locales (`show / diff` un commit spécifique, `cerises-picks` et `resets`, etc.)

---

## Dépannage

#### La commande `git svn rebase` génère une erreur d'incompatibilité de somme de contrôle

La commande `git svn rebase` renvoie une erreur similaire à celle-ci:

```
Checksum mismatch: <path_to_file> <some_kind_of_shal>
expected: <checksum_number_1>
got: <checksum_number_2>
```

La solution à ce problème consiste à réinitialiser svn à la révision lorsque le fichier en question a été modifié pour la dernière fois, et à effectuer une extraction `git svn` pour que l'historique SVN soit restauré. Les commandes pour effectuer la réinitialisation SVN sont:

- `git log -1 - <path_to_file>` (copie le numéro de révision SVN qui apparaît dans le message de validation)
- `git svn reset <revision_number>`
- `git svn fetch`

Vous devriez être capable de pousser / extraire à nouveau les données de SVN

**Le fichier n'a pas été trouvé dans commit** Lorsque vous essayez d'extraire ou d'extraire SVN, vous obtenez une erreur similaire à celle-ci.

```
<file_path> was not found in commit <hash>
```

Cela signifie qu'une révision dans SVN essaie de modifier un fichier qui, pour une raison quelconque, n'existe pas dans votre copie locale. La meilleure façon de se débarrasser de cette erreur est de forcer une extraction en ignorant le chemin de ce fichier et elle sera mise à jour

à son état dans la dernière révision SVN:

- `git svn fetch --ignore-paths <file_path>`

## Exemples

### Cloner le dépôt SVN

Vous devez créer une nouvelle copie locale du référentiel avec la commande

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b BRANCHES_REPO_PATH
```

Si votre référentiel SVN suit la disposition standard (tronc, branches, dossiers de balises), vous pouvez économiser de la saisie:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` extrait chaque révision SVN, une par une, et effectue une validation git dans votre référentiel local afin de recréer l'historique. Si le dépôt SVN contient beaucoup de validations, cela prendra du temps.

Lorsque la commande est terminée, vous disposez d'un dépôt git complet avec une branche locale appelée master qui suit la branche du trunk dans le référentiel SVN.

### Obtenir les dernières modifications de SVN

L'équivalent de `git pull` est la commande

```
git svn rebase
```

Ceci récupère toutes les modifications du référentiel SVN et les applique *sur* vos commits locaux dans votre branche actuelle.

Vous pouvez également utiliser la commande

```
git svn fetch
```

pour récupérer les modifications du référentiel SVN et les apporter à votre machine locale mais sans les appliquer à votre branche locale.

### Modification locale de SVN

La commande

```
git svn dcommit
```

créera une révision SVN pour chacun de vos commits git locaux. Comme avec SVN, votre historique de git local doit être synchronisé avec les dernières modifications du référentiel SVN, donc si la commande échoue, essayez d'abord d'effectuer une `git svn rebase`.

### Travailler localement

Utilisez simplement votre dépôt git local comme un référentiel git normal, avec les commandes git normales:

- `git add FILE` et `git checkout -- FILE` Pour mettre en scène / décompresser un fichier
- `git commit` Pour enregistrer vos modifications. Ces commits seront locaux et ne seront pas "poussés" vers le référentiel SVN, comme dans un dépôt git normal.

- `git stash` et `git stash pop` Permet d'utiliser des stashes
- `git reset HEAD --hard` tous vos changements locaux
- `git log` Accédez à tout l'historique du référentiel
- `git rebase -i` pour que vous puissiez réécrire votre histoire locale librement
- `git branch` et `git checkout` pour créer des branches locales

Comme l'indique la documentation de `git-svn` "Subversion est un système beaucoup moins sophistiqué que Git", vous ne pouvez donc pas utiliser toute la puissance de git sans perturber l'historique du serveur Subversion. Heureusement, les règles sont très simples: **gardez**

### **L'histoire linéaire**

Cela signifie que vous pouvez effectuer presque toutes les opérations git: créer des branches, supprimer / réorganiser / écraser des commits, déplacer l'historique, supprimer des commits, etc. Tout *sauf des fusions* . Si vous avez besoin de réintégrer l'historique des branches locales, utilisez plutôt `git rebase` .

Lorsque vous effectuez une fusion, une validation de fusion est créée. La particularité des validations de fusion est qu'elles ont deux parents, ce qui rend l'historique non linéaire. L'historique non linéaire va confondre SVN dans le cas où vous "poussez" une validation de fusion dans le référentiel.

Cependant, ne vous inquiétez pas: **vous ne casserez rien si vous "poussez" une validation de git merge sur SVN** . Si vous le faites, lorsque la validation git merge est envoyée au serveur svn, elle contiendra toutes les modifications de tous les commits pour cette fusion, vous perdrez ainsi l'historique de ces validations, mais pas les modifications de votre code.

### **Gestion des dossiers vides**

git ne reconnaît pas le concept de dossiers, il ne fait que travailler avec les fichiers et leurs chemins de fichiers. Cela signifie que git ne suit pas les dossiers vides. SVN, cependant. Utiliser `git-svn` signifie que, par défaut, *toute modification apportée à des dossiers vides avec git ne sera pas propagée à SVN* .

L'utilisation de l'indicateur `--rmdir` lors de l'émission d'un commentaire corrige ce problème et supprime un dossier vide dans SVN si vous supprimez localement le dernier fichier qu'il `--rmdir` :

```
git svn dcommit --rmdir
```

Malheureusement, **il ne supprime pas les dossiers vides existants** : vous devez le faire manuellement.

Pour éviter d'ajouter le drapeau à chaque fois que vous faites un `dcommit`, ou pour le jouer en toute sécurité si vous utilisez un outil graphique git (comme SourceTree), vous pouvez définir ce comportement par défaut avec la commande:

```
git config --global svn.rmdir true
```

Cela modifie votre fichier `.gitconfig` et ajoute ces lignes:

```
[svn]
rmdir = true
```

Pour supprimer tous les fichiers et dossiers non suivis devant rester vides pour SVN, utilisez la commande git:

```
git clean -fd
```

Remarque: la commande précédente supprime tous les fichiers non suivis et les dossiers vides, même ceux qui doivent être suivis par SVN! Si vous devez générer des rapports, les dossiers vides suivis par SVN utilisent la commande

```
git svn makedirs
```

En pratique, cela signifie que si vous souhaitez nettoyer votre espace de travail à partir de fichiers et de dossiers non suivis, vous devez toujours utiliser les deux commandes pour recréer les dossiers vides suivis par SVN:

```
git clean -fd && git svn makedirs
```

Lire `git-svn` en ligne: <https://riptutorial.com/fr/git/topic/2766/git-svn>

## Chapitre 28: git-tfs

### Remarques

`Git-tfs` est un outil tiers permettant de connecter un référentiel Git à un référentiel Team Foundation Server («TFS»).

La plupart des instances TFVS distantes demanderont vos informations d'identification à chaque interaction et l'installation de `Git-Credential-Manager-for-Windows` peut ne pas vous aider. Cela peut être surmonté en ajoutant votre *nom* et votre *mot de passe* à votre `.git/config`

```
[tfs-remote "default"]
  url = http://tfs.mycompany.co.uk:8080/tfs/DefaultCollection/
  repository = $/My.Project.Name/
  username = me.name
  password = My733TPwd
```

### Exemples

#### `git-tfs clone`

Cela créera un dossier avec le même nom que le projet, à savoir `/My.Project.Name`

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

#### `git-tfs clone du dépôt git`

Le clonage à partir d'un référentiel git est dix fois plus rapide que le clonage direct à partir de TFVS et fonctionne bien dans un environnement d'équipe. Au moins un membre de l'équipe devra créer le référentiel git en effectuant le clone habituel de `git-tfs`. Ensuite, le nouveau référentiel peut être démarré pour fonctionner avec TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

#### `git-tfs installer via Chocolatey`

Ce qui suit suppose que vous utiliserez `kdiff3` pour la diff

```
C:\> choco install kdiff3
```

Git peut être installé en premier afin que vous puissiez indiquer tous les paramètres que vous souhaitez. Ici, tous les outils Unix sont également installés et `"NoAutoCrlf"` signifie `"checkout"` tel quel, valide tel quel.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"
```

C'est tout ce dont vous avez vraiment besoin pour pouvoir installer `git-tfs` via `chocolatey`.

```
C:\> choco install git-tfs
```

#### `git-tfs Check In`

Lancez la boîte de dialogue Check In pour TFVS.

```
$ git tfs checkintool
```

Cela prendra tous vos commits locaux et créera un enregistrement unique.

### **git-tfs pousser**

Poussez tous les commits locaux vers la télécommande TFVS.

```
$ git tfs rcheckin
```

Remarque: ceci échouera si des notes d'enregistrement sont requises. Celles-ci peuvent être git-tfs-force: rcheckin en ajoutant git-tfs-force: rcheckin au message de validation.

Lire git-tfs en ligne: <https://riptutorial.com/fr/git/topic/2660/git-tfs>

## Chapitre 29: Historique de réécriture avec filtre-branche

### Exemples

#### Changer l'auteur des commits

Vous pouvez utiliser un filtre d'environnement pour modifier l'auteur des validations. Modifiez et exportez `$GIT_AUTHOR_NAME` dans le script pour modifier l'auteur du commit.

Créez un fichier `filter.sh` avec un contenu tel que:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
  export GIT_AUTHOR_NAME="Author to Change To"
  export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Puis lancez `filter-branch` partir de la ligne de commande:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

#### Réglage de `git commit` égal à `commit author`

Cette commande, avec une plage de validation `commit1..commit2`, réécrit l'historique pour que `git commit author` devienne également `git commit`:

```
git filter-branch -f --commit-filter \
'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
git commit-tree $@' \
-- commit1..commit2
```

Lire Historique de réécriture avec filtre-branche en ligne:

<https://riptutorial.com/fr/git/topic/2825/historique-de-reecriture-avec-filtre-branche>



---

## Chapitre 30: Ignorer les fichiers et les dossiers

### Introduction

Cette rubrique explique comment éviter d'ajouter des fichiers indésirables (ou des modifications de fichiers) dans un dépôt Git. Il existe plusieurs manières (globales ou locales `.gitignore`, `.git/exclude`, `git update-index --assume-unchanged` et `git update-index --skip-tree`), mais gardez à l'esprit que Git gère le *contenu*, ce qui signifie: ignorer ignore en fait un *contenu* de dossier (c.-à-d. des fichiers). Un dossier vide serait ignoré par défaut, car il ne peut pas être ajouté de toute façon.

### Exemples

#### Ignorer les fichiers et les répertoires avec un fichier `.gitignore`

Vous pouvez faire en sorte que Git ignore certains fichiers et répertoires - c'est-à-dire les exclure du suivi par Git - en créant un ou plusieurs fichiers `.gitignore` dans votre référentiel.

Dans les projets logiciels, `.gitignore` contient généralement une liste des fichiers et / ou des répertoires générés au cours du processus de génération ou à l'exécution. Les entrées du fichier `.gitignore` peuvent inclure des noms ou des chemins pointant vers:

1. ressources temporaires, par exemple caches, fichiers journaux, code compilé, etc.
2. fichiers de configuration locaux à ne pas partager avec d'autres développeurs
3. les fichiers contenant des informations secrètes, telles que les mots de passe de connexion, les clés et les informations d'identification

Lorsqu'elles sont créées dans le répertoire de niveau supérieur, les règles s'appliquent de manière récursive à tous les fichiers et sous-répertoires de l'ensemble du référentiel. Lorsqu'elles sont créées dans un sous-répertoire, les règles s'appliquent à ce répertoire spécifique et à ses sous-répertoires.

Lorsqu'un fichier ou un répertoire est ignoré, il ne sera pas:

1. suivi par Git
2. signalé par des commandes telles que `git status` ou `git diff`
3. mis en scène avec des commandes telles que `git add -A`

Dans le cas inhabituel où vous devez ignorer les fichiers suivis, vous devez faire particulièrement attention. Voir: [Ignorer les fichiers qui ont déjà été validés dans un référentiel Git](#) .

---

### Exemples

Voici quelques exemples génériques de règles dans un fichier `.gitignore`, basées sur [des modèles de fichiers glob](#) :

```
# Lines starting with `#` are comments.

# Ignore files called 'file.ext'
file.ext

# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment

# Ignoring files with full path.
# This matches files in the root directory and subdirectories too.
```

```

# i.e. otherfile.ext will be ignored anywhere on the tree.
dir/otherdir/file.ext
otherfile.ext

# Ignoring directories
# Both the directory itself and its contents will be ignored.
bin/
gen/

# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

# Ignoring files by extension
# All files with these extensions will be ignored in
# this directory and all its sub-directories.
*.apk
*.class

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
DirectoryA/**/DirectoryB/
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

```

```
# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

La plupart des fichiers `.gitignore` sont standard dans différents langages. Pour commencer, voici un ensemble de [fichiers .gitignore](#) répertoriés par langue à partir desquels cloner ou copier / modifier dans votre projet. Sinon, pour un nouveau projet, vous pouvez envisager de générer automatiquement un fichier de démarrage à l'aide d'un [outil en ligne](#) .

---

### Autres formes de `.gitignore`

`.gitignore` fichiers `.gitignore` sont destinés à être `.gitignore` dans le cadre du référentiel. Si vous souhaitez ignorer certains fichiers sans valider les règles d'ignorance, voici quelques options:

- Editez le `.git/info/exclude` (en utilisant la même syntaxe que `.gitignore` ). Les règles seront globales dans le périmètre du référentiel;
- Configurez un [fichier gitignore global](#) qui appliquera les règles ignore à tous vos référentiels locaux:

De plus, vous pouvez ignorer les modifications locales apportées aux fichiers suivis sans modifier la configuration globale de git avec:

- `git update-index --skip-worktree [<file>...]` : pour les modifications locales mineures
- `git update-index --assume-unchanged [<file>...]` : pour les fichiers prêts à changer en amont de la production

Voir [plus de détails sur les différences entre les derniers indicateurs](#) et la documentation de `git update-index` pour d'autres options.

---

### Nettoyage des fichiers ignorés

Vous pouvez utiliser `git clean -X` pour `git clean -X` les fichiers ignorés:

```
git clean -Xn #display a list of ignored files
git clean -Xf #remove the previously displayed files
```

Remarque: `-X` (majuscules) nettoie *uniquement* les fichiers ignorés. Utilisez `-x` (no caps) pour supprimer également les fichiers non suivis.

Consultez [la documentation](#) de `git clean` pour plus de détails.

---

Voir [le manuel de Git](#) pour plus de détails.

### Exceptions dans un fichier `.gitignore`

Si vous ignorez des fichiers en utilisant un modèle mais que vous avez des exceptions, préfixez un point d'exclamation (!) à l'exception. Par exemple:

```
*.txt
!important.txt
```

L'exemple ci-dessus indique à Git d'ignorer tous les fichiers portant l'extension `.txt` exception des fichiers nommés `important.txt` .

---

Si le fichier est dans un dossier ignoré, vous ne pouvez **PAS le** ré-inclure si facilement:

```
folder/  
!folder/*.txt
```

Dans cet exemple, tous les fichiers .txt du dossier resteraient ignorés.

La bonne manière est de ré-inclure le dossier lui-même sur une ligne distincte, puis d'ignorer tous les fichiers du folder par \* , enfin d'inclure à nouveau le fichier \*.txt dans le folder , comme suit:

```
!folder/  
folder/*  
!folder/*.txt
```

**Remarque** : Pour les noms de fichiers commençant par un point d'exclamation, ajoutez deux points d'exclamation ou un échappement avec le caractère \ :

```
!!includethis  
\!excludethis
```

### Un fichier global .gitignore

Pour que Git ignore certains fichiers sur tous les référentiels, vous pouvez [créer un fichier .gitignore global](#) avec la commande suivante dans votre terminal ou votre invite de commande:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git l'utilisera désormais en plus du fichier .gitignore de chaque dépôt. Les règles sont les suivantes:

- Si le fichier .gitignore local .gitignore explicitement un fichier alors que le fichier .gitignore ignore, le .gitignore local est prioritaire (le fichier sera inclus)
- Si le référentiel est cloné sur plusieurs machines, le .gitignore global doit être chargé sur toutes les machines ou au moins l'inclure, car les fichiers ignorés seront .gitignore alors que le PC avec le .gitignore global .gitignore mettra pas à jour. . Ceci est la raison pour laquelle un particulier repo .gitignore est une meilleure idée que un mondial si le projet est travaillé par une équipe

Ce fichier est un bon endroit pour garder des ignorés spécifiques aux plates-formes, aux machines ou aux utilisateurs, par exemple OSX .DS\_Store , Windows Thumbs.db ou Vim \*.ext~ et \*.ext.swp ignore si vous ne souhaitez pas conserver ceux du référentiel . Ainsi, un membre de l'équipe travaillant sur OS X peut ajouter tous les .DS\_STORE et \_MACOSX (ce qui est en fait inutile), tandis qu'un autre membre de l'équipe sous Windows peut ignorer tous les thumbs.bd

### Ignorer les fichiers qui ont déjà été validés dans un référentiel Git

Si vous avez déjà ajouté un fichier à votre référentiel Git et que vous souhaitez maintenant **arrêter de le suivre** (afin qu'il ne soit plus présent dans les futurs commits), vous pouvez le supprimer de l'index:

```
git rm --cached <file>
```

Cela supprimera le fichier du référentiel et empêchera Git de suivre d'autres modifications. L'option --cached s'assurera que le fichier n'est pas physiquement supprimé.

Notez que le contenu précédemment ajouté du fichier sera toujours visible via l'historique Git.

Gardez à l'esprit que si quelqu'un d'autre tire du référentiel après avoir supprimé le fichier de l'index, **sa copie sera supprimée physiquement** .

---

Vous pouvez faire en sorte que Git prétende que la version du répertoire de travail du fichier est à jour et lit la version de l'index (ignorant ainsi les modifications apportées) avec le bit " `skip worktree` ":

```
git update-index --skip-worktree <file>
```

L'écriture n'est pas affectée par ce bit, la sécurité du contenu reste la première priorité. Vous ne perdrez jamais vos précieux changements ignorés; par contre ce bit est en conflit avec le cache: pour supprimer ce bit, utilisez

```
git update-index --no-skip-worktree <file>
```

---

Il est parfois recommandé à **tort** de mentir à Git et de lui faire supposer que le fichier n'a pas été modifié sans l'examiner. À première vue, il ignore les modifications apportées au fichier sans le supprimer de son index:

```
git update-index --assume-unchanged <file>
```

Cela forcera git à ignorer les modifications apportées au fichier (gardez à l'esprit que si vous apportez des modifications à ce fichier ou si vous le rangez, **vos modifications ignorées seront perdues** )

Si vous voulez que git se soucie de ce fichier, lancez la commande suivante:

```
git update-index --no-assume-unchanged <file>
```

### Vérifier si un fichier est ignoré

La commande `git check-ignore` rapporte les fichiers ignorés par Git.

Vous pouvez passer des noms de fichiers sur la ligne de commande et `git check-ignore` listera les noms de fichiers ignorés. Par exemple:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Ici, seuls les fichiers `*.o` sont définis dans `.gitignore`, `Readme.md` n'est donc pas répertorié dans la sortie de `git check-ignore` .

Si vous voulez voir la ligne dont `.gitignore` est responsable d'ignorer un fichier, ajoutez `-v` à la commande `git check-ignore`:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o      example.o
```

---

A partir de Git 1.7.6, vous pouvez également utiliser `git status --ignored` pour voir les fichiers ignorés. Vous pouvez trouver plus d'informations à ce sujet dans la [documentation officielle](#) ou dans [Recherche de fichiers ignorés par .gitignore](#) .

## Ignorer les fichiers dans les sous-dossiers (fichiers gitignore multiples)

Supposons que vous ayez une structure de référentiel comme celle-ci:

```
examples/  
  output.log  
src/  
  <files not shown>  
  output.log  
README.md
```

output.log dans le répertoire des exemples est valide et nécessaire pour que le projet prenne une compréhension tandis que celui sous src/ est créé lors du débogage et ne doit pas figurer dans l'historique ou une partie du référentiel.

Il existe deux manières d'ignorer ce fichier. Vous pouvez placer un chemin absolu dans le fichier .gitignore à la racine du répertoire de travail:

```
# /.gitignore  
src/output.log
```

Vous pouvez également créer un fichier .gitignore dans le répertoire src/ et ignorer le fichier relatif à ce .gitignore :

```
# /src/.gitignore  
output.log
```

## Ignorer un fichier dans n'importe quel répertoire

Pour ignorer un fichier foo.txt dans **n'importe quel** répertoire, vous devez simplement écrire son nom:

```
foo.txt # matches all files 'foo.txt' in any directory
```

Si vous souhaitez ignorer le fichier uniquement dans une partie de l'arborescence, vous pouvez spécifier les sous-répertoires d'un répertoire spécifique avec \*\* pattern:

```
bar/**/foo.txt # matches all files 'foo.txt' in 'bar' and all subdirectories
```

Ou vous pouvez créer un fichier .gitignore dans le répertoire bar/ . Équivalent à l'exemple précédent serait créer un fichier bar/.gitignore avec ces contenus:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

## Ignorer les fichiers localement sans valider les règles ignore

.gitignore ignore les fichiers localement, mais il est destiné à être .gitignore au référentiel et partagé avec d'autres contributeurs et utilisateurs. Vous pouvez définir un .gitignore global, mais tous vos référentiels partageraient ces paramètres.

Si vous souhaitez ignorer certains fichiers dans un référentiel localement et ne pas inclure le fichier dans un référentiel, éditez le .git/info/exclude dans votre référentiel.

Par exemple:

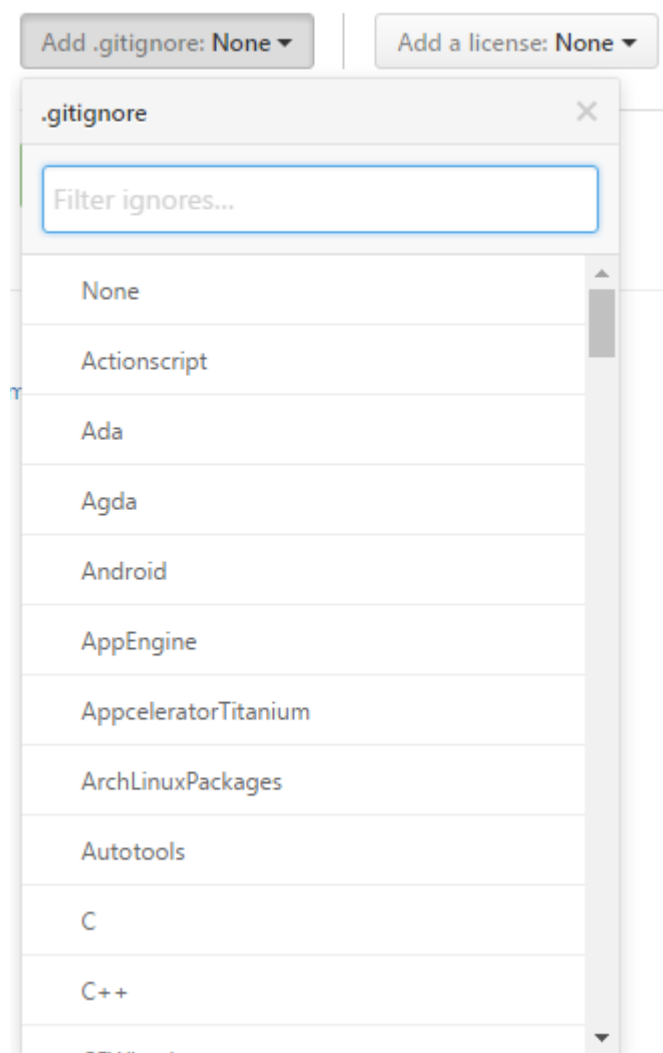
```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
pushReports.py
server/
```

### Modèles .gitignore pré-remplis

Si vous ne savez pas quelles règles .gitignore dans votre fichier .gitignore ou si vous souhaitez simplement ajouter des exceptions généralement acceptées à votre projet, vous pouvez choisir ou générer un fichier .gitignore :

- <https://www.gitignore.io/>
- <https://github.com/github/gitignore>

De nombreux services d'hébergement, tels que GitHub et BitBucket, permettent de générer des fichiers .gitignore en fonction des langages de programmation et des IDE que vous utilisez:



### Ignorer les modifications ultérieures apportées à un fichier (sans le supprimer)

Parfois, vous voulez avoir un fichier dans Git mais ignorez les modifications ultérieures.

Dites à Git d'ignorer les modifications apportées à un fichier ou à un répertoire en utilisant

update-index :

```
git update-index --assume-unchanged my-file.txt
```

La commande ci-dessus indique à Git de supposer que my-file.txt n'a pas été modifié et de ne pas vérifier ou signaler les modifications. Le fichier est toujours présent dans le référentiel.

Cela peut être utile pour fournir des valeurs par défaut et autoriser des substitutions d'environnement local, par exemple:

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git
git add .env
git commit -m "Adding .env template"

# ignore future changes to .env
git update-index --assume-unchanged .env

# update your password
vi .env

# no changes!
git status
```

### Ignorer seulement une partie d'un fichier [stub]

Parfois, vous souhaitez peut-être modifier localement un fichier que vous ne souhaitez ni valider ni publier. Dans l'idéal, les paramètres locaux devraient être concentrés dans un fichier distinct pouvant être placé dans .gitignore , mais parfois, dans le cadre d'une solution à court terme, il peut être utile d'avoir quelque chose de local dans un fichier .gitignore .

Vous pouvez faire Git "unsee" ces lignes en utilisant un filtre propre. Ils ne vont même pas apparaître dans les diffs.

Supposons ici un extrait du fichier file1.c :

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

Vous ne voulez pas publier les lignes NOCOMMIT n'importe où.

Créez le filtre "nocommit" en ajoutant ce fichier au fichier de configuration Git tel que .git/config :

```
[filter "nocommit"]
  clean=grep -v NOCOMMIT
```

Ajoutez (ou créez) ceci à .git/info/attributes ou .gitmodules :



```
file1.c filter=nocommit
```

Et vos lignes NOCOMMIT sont cachées de Git.

Mises en garde:

- L'utilisation d'un filtre propre ralentit le traitement des fichiers, en particulier sous Windows.
- La ligne ignorée peut disparaître du fichier lorsque Git le met à jour. Il peut être contrecarré par un filtre à tâches, mais c'est plus compliqué.
- Non testé sur Windows

**Ignorer les modifications dans les fichiers suivis. [bout]**

`.gitignore` et `.git/info/exclude` ne fonctionnent que pour les fichiers non suivis.

Pour définir l'indicateur ignore sur un fichier suivi, utilisez la commande `update-index` :

```
git update-index --skip-worktree myfile.c
```

Pour revenir en arrière, utilisez:

```
git update-index --no-skip-worktree myfile.c
```

Vous pouvez ajouter cet extrait à votre `configuration git` globale pour disposer de commandes `git hide` , `git unhide` et `git hidden` plus pratiques:

```
[alias]
  hide   = update-index --skip-worktree
  unhide = update-index --no-skip-worktree
  hidden = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

Vous pouvez également utiliser l'option `--assume-unchanged` avec la fonction `update-index`

```
git update-index --assume-unchanged <file>
```

Si vous voulez revoir ce fichier pour les modifications, utilisez

```
git update-index --no-assume-unchanged <file>
```

Lorsque l'option `--assume-unchanged` est spécifiée, l'utilisateur s'engage à ne pas modifier le fichier et permet à Git de supposer que le fichier de l'arborescence de travail correspond à ce qui est enregistré dans l'index. Git échoue s'il doit modifier ce fichier dans l'index par exemple lors de la fusion dans un commit; ainsi, si le fichier supposé non suivi est modifié en amont, vous devrez gérer la situation manuellement. Dans ce cas, l'accent est mis sur les performances.

Alors que l'option `--skip-worktree` est utile lorsque vous indiquez à git de ne pas toucher un fichier spécifique parce que le fichier doit être modifié localement et que vous ne souhaitez pas commettre accidentellement les modifications (fichier de configuration / propriétés configuré pour un fichier particulier) environnement). `Skip-worktree` est prioritaire par rapport à `assume-unchanged` lorsque les deux sont définis.

**Effacer les fichiers déjà validés, mais inclus dans `.gitignore`**

Parfois, il arrive qu'un fichier soit suivi par git, mais à un moment ultérieur, il a été ajouté à `.gitignore`, afin de ne plus le suivre. C'est un scénario très courant d'oublier de nettoyer de

tels fichiers avant son ajout à `.gitignore`. Dans ce cas, l'ancien fichier sera toujours dans le référentiel.

Pour résoudre ce problème, il est possible d'effectuer une suppression "sèche" de tout ce qui se trouve dans le référentiel, puis de rajouter tous les fichiers. Tant que vous n'avez pas de modifications en attente et que le paramètre `--cached` est passé, cette commande est assez sûre pour être exécutée:

```
# Remove everything from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

### Créer un dossier vide

Il n'est pas possible d'ajouter et de valider un dossier vide dans Git car Git gère les *fichiers* et leur associe leur répertoire, ce qui ralentit les validations et améliore la vitesse. Pour contourner cela, il existe deux méthodes:

`.gitkeep` méthode: `.gitkeep`

Pour contourner ce `.gitkeep` utilisez un fichier `.gitkeep` pour enregistrer le dossier de Git. Pour ce faire, créez simplement le répertoire requis et ajoutez un fichier `.gitkeep` au dossier. Ce fichier est vide et ne sert à rien d'autre que d'enregistrer le dossier. Pour ce faire, dans Windows (qui a des conventions de nommage de fichiers peu pratiques), ouvrez simplement git bash dans le répertoire et exécutez la commande:

```
$ touch .gitkeep
```

Cette commande crée simplement un fichier `.gitkeep` vierge dans le répertoire en cours

Méthode deux: `dummy.txt`

Un autre hack pour cela est très similaire à ce qui précède et les mêmes étapes peuvent être suivies, mais au lieu d'un `.gitkeep`, utilisez simplement un `dummy.txt` place. Cela a l'avantage supplémentaire de pouvoir le créer facilement dans Windows en utilisant le menu contextuel. Vous pouvez également y laisser des messages amusants. Vous pouvez également utiliser le fichier `.gitkeep` pour suivre le répertoire vide. `.gitkeep` est normalement un fichier vide qui est ajouté pour suivre le directoy vide.

### Recherche de fichiers ignorés par `.gitignore`

Vous pouvez lister tous les fichiers ignorés par git dans le répertoire courant avec la commande:

```
git status --ignored
```

Donc, si nous avons une structure de référentiel comme celle-ci:

```
.git
.gitignore
./example_1
./dir/example_2
```

```
./example_2
```

... et le fichier .gitignore contenant:

```
example_2
```

... que le résultat de la commande sera:

```
$ git status --ignored
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
.example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/
example_2
```

Si vous souhaitez répertorier les fichiers ignorés récursivement dans les répertoires, vous devez utiliser des paramètres supplémentaires - `--untracked-files=all`

Le résultat ressemblera à ceci:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Lire Ignorer les fichiers et les dossiers en ligne:

<https://riptutorial.com/fr/git/topic/245/ignorer-les-fichiers-et-les-dossiers>

## Chapitre 31: Internes

### Exemples

#### Repo

Un git repository est une structure de données sur disque qui stocke les métadonnées pour un ensemble de fichiers et de répertoires.

Il réside dans le dossier `.git/` votre projet. Chaque fois que vous validez des données sur git, elles sont stockées ici. Inversement, `.git/` contient chaque commit unique.

Sa structure de base est la suivante:

```
.git/  
  objects/  
  refs/
```

#### Objets

git est fondamentalement un magasin clé-valeur. Lorsque vous ajoutez des données à git, il crée un object et utilise le hachage SHA-1 du contenu de l' object comme clé.

Par conséquent, tout contenu de git peut être recherché par son hash:

```
git cat-file -p 4bb6f98
```

Il existe 4 types d' Object :

- blob
- tree
- commit
- tag

#### HEAD ref

HEAD est une ref spéciale Il pointe toujours vers l'objet en cours.

Vous pouvez voir où il pointe actuellement en vérifiant le fichier `.git/HEAD` .

HEAD désigne normalement une autre ref :

```
$cat .git/HEAD  
ref: refs/heads/mainline
```

Mais il peut aussi pointer directement vers un object :

```
$ cat .git/HEAD  
4bb6f98a223abc9345a0cef9200562333
```

C'est ce qu'on appelle une « tête détachée » - parce HEAD est pas attaché à (montrant) tout ref, mais désigne plutôt directement à un object .

#### Refs

Un ref est essentiellement un pointeur. C'est un nom qui pointe vers un object . Par exemple,

```
"master" --> 1a410e...
```

Ils sont stockés dans ``.git / refs / heads /` dans les fichiers texte.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

C'est communément ce qu'on appelle les branches . Cependant, vous remarquerez qu'en git il n'existe pas de branch - seulement une ref .

Maintenant, il est possible de naviguer dans git simplement en sautant sur différents objets directement par leurs hashes. Mais cela serait terriblement gênant. Un ref vous donne un nom pratique pour faire référence aux objets par. Il est beaucoup plus facile de demander à git d'aller à un endroit précis par son nom plutôt que par hash.

### Objet de validation

Un commit est probablement le type d' object plus familier pour les utilisateurs git , car c'est ce qu'ils sont habitués à créer avec les commandes git commit .

Cependant, la commit ne contient pas directement de fichiers ou de données modifiés. Au contraire, il contient principalement des métadonnées et des pointeurs vers d'autres objets qui contiennent le contenu réel du commit .

Un commit contient quelques choses:

- hachage d'un tree
- hash d'un parent commit
- nom de l'auteur / email, nom du commiter / email
- commettre un message

Vous pouvez voir le contenu de tout commit comme ceci:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
committer Neil Peart <npeart@rush.com>

First commit!
```

---

### Arbre

Une note très importante est que l' tree objets stocke TOUS les fichiers dans votre projet et stocke des fichiers entiers non diffs. Cela signifie que chaque commit contient un instantané du projet entier \*.

*\* Techniquement, seuls les fichiers modifiés sont stockés. Mais c'est plus un détail d'implémentation pour l'efficacité. Du point de vue de la conception, un commit doit être considéré comme contenant une copie complète du projet .*

---

### Parent

La ligne parent contient un hachage d'un autre objet commit et peut être considérée comme un "pointeur parent" qui pointe vers le "commit précédent". Cela forme implicitement un graphe de commits appelé **graphe de validation** . Plus précisément, il s'agit d'un [graphe acyclique dirigé](#) (ou DAG).

## Objet d'arbre

Un tree représente essentiellement un dossier dans un système de fichiers traditionnel: des conteneurs imbriqués pour des fichiers ou d'autres dossiers.

Un tree contient:

- 0 ou plusieurs objets blob
- 0 ou plusieurs objets d' tree

Tout comme vous pouvez utiliser `ls` ou `dir` pour répertorier le contenu d'un dossier, vous pouvez répertorier le contenu d'un objet d' tree .

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b  .gitignore
100644 blob cc0956f1  Makefile
040000 tree 92e1ca7e  src
...
```

Vous pouvez rechercher les fichiers dans une commit en trouvant d'abord le hachage de l' tree dans la commit , puis en regardant cette tree :

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
committer ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b  .gitignore
100644 blob cc0956f1  Makefile
040000 tree 92e1ca7e  src
...
```

## Objet blob

Un blob contient un contenu de fichier binaire arbitraire. Généralement, il s'agira d'un texte brut tel qu'un code source ou un article de blog. Mais il pourrait tout aussi bien s'agir des octets d'un fichier PNG ou de toute autre chose.

Si vous avez le hash d'un blob , vous pouvez regarder son contenu.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
  ...
}
...
```

Par exemple, vous pouvez parcourir un tree comme ci-dessus, puis regarder l'un des blobs qu'il blobs .

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b  .gitignore
100644 blob cc0956f1  Makefile
040000 tree 92e1ca7e  src
```

```
100644 blob cae391ff  Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
...
```

### Créer de nouveaux commits

La commande de git commit fait quelques choses:

1. Créez des blobs et des trees pour représenter votre répertoire de projet - stockés dans des `.git/objects`
2. Crée un nouvel objet de commit avec les informations de votre auteur, le message de validation et l' tree racine de l'étape 1 - également stocké dans les `.git/objects`
3. `.git/HEAD` à jour la référence HEAD dans `.git/HEAD` au hachage du commit nouvellement créé

Cela se traduit par un nouvel instantané de votre projet ajouté à git qui est connecté à l'état précédent.

### Déplacement de la tête

Lorsque vous exécutez `git checkout` sur un commit (spécifié par hash ou ref), vous git à git que votre répertoire de travail doit ressembler à ce qu'il a fait lors de la prise de l'instantané.

1. Mettre à jour les fichiers du répertoire de travail pour qu'ils correspondent à l' tree la commit
2. Mettre à jour HEAD pour pointer sur le hachage ou la référence spécifié

### Déplacement des références

`git reset --hard` déplace la référence au hachage / ref spécifié.

Déplacement de MyBranch vers b8dc53 :

```
$ git checkout MyBranch      # moves HEAD to MyBranch
$ git reset --hard b8dc53    # makes MyBranch point to b8dc53
```

### Créer de nouvelles références

`git checkout -b <refname>` va créer une nouvelle ref qui pointe sur le commit .

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

Lire Internes en ligne: <https://riptutorial.com/fr/git/topic/2637/internes>

## Chapitre 32: Liasses

### Remarques

La clé pour faire ce travail est de commencer par cloner un paquet qui commence dès le début de l'historique des pensions:

```
git bundle create initial.bundle master
git tag -f some_previous_tag master # so the whole repo does not have to go each time
```

obtenir ce paquet initial sur la machine distante; et

```
git clone -b master initial.bundle remote_repo_name
```

### Exemples

#### Créer un bundle git sur la machine locale et l'utiliser sur un autre

Parfois, vous pouvez souhaiter conserver des versions d'un référentiel git sur des machines sans connexion réseau. Les bundles vous permettent de regrouper des objets et des références git dans un référentiel sur une machine et de les importer dans un référentiel sur une autre.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

Transférer en quelque sorte le fichier **changes\_between\_tags.bundle** sur la machine distante; par exemple, par clé USB. Une fois que vous l'avez là:

```
git bundle verify changes_between_tags.bundle # make sure bundle arrived intact
git checkout [some branch] # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the previous output]
```

L'inverse est également possible. Une fois les modifications apportées au référentiel distant, vous pouvez regrouper les deltas. mettez les modifications sur, par exemple, une clé USB et fusionnez-les dans le référentiel local pour que les deux puissent rester synchronisés sans nécessiter un accès direct au protocole git , ssh , rsync ou http entre les machines.

Lire Liasses en ligne: <https://riptutorial.com/fr/git/topic/3612/liasses>



## Chapitre 33: Liste de révocation

### Syntaxe

- `git rev-list [options] <commit> ...`

### Paramètres

| Paramètre                | Détails   |
|--------------------------|---|
| <code>--une ligne</code> | L'affichage valide comme une seule ligne avec leur titre. |

### Exemples

Liste Commit en maître mais pas en origine / maître

```
git rev-list --oneline master ^origin/master
```

Git `rev-list` listera les commits dans une branche qui ne sont pas dans une autre branche. C'est un excellent outil lorsque vous essayez de déterminer si le code a été fusionné dans une branche ou non.

- L'utilisation de l'option `--oneline` affichera le titre de chaque validation.
- L'opérateur `^` exclut les commits dans la branche spécifiée de la liste.
- Vous pouvez passer plus de deux branches si vous le souhaitez. Par exemple, `git rev-list foo bar ^baz` liste les commits dans `foo` et `bar`, mais pas `baz`.

Lire Liste de révocation en ligne: <https://riptutorial.com/fr/git/topic/431/liste-de-revocation>

## Chapitre 34: Marquage Git

### Introduction

Comme la plupart des systèmes de contrôle de version (VCS), Git a la capacité de tag des points spécifiques de l'histoire comme étant importants. En règle générale, les utilisateurs utilisent cette fonctionnalité pour marquer les points de publication (version v1.0 , etc.).

### Syntaxe

- `git tag [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <fichier>] <tagname> [<commit> | <objet>]`
- `git tag -d <tagname>`
- balise `git [-n [<num>]] -l [--contient <commit>] [--contains <commit>] [--points-at <objet>] [--column [= <options>] | --no-column] [--create-reflog] [--sort = <clé>] [--format = <format>] [- [no-] fusionné [<commit>]] [<pattern>... ]`
- `git tag -v [--format = <format>] <tagname>...`

### Exemples

#### Liste de tous les tags disponibles

Utiliser la commande `git tag` répertorie tous les tags disponibles:

```
$ git tag
<output follows>
v0.1
v1.3
```

**Remarque :** les tags sont affichées dans un ordre **alphabétique** .

On peut également search les tags disponibles:

```
$ git tag -l "v1.8.5*"
<output follows>
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 et envoyer des tags dans GIT

##### Créez un tag:

- Pour créer un tag sur votre branche actuelle:

```
git tag < tagname >
```

Cela créera une tag locale avec l'état actuel de la branche sur laquelle vous vous trouvez.

- Pour créer une balise avec un commit:

```
git tag tag-name commit-identifiant
```

Cela créera une tag locale avec l'identificateur de validation de la branche sur laquelle vous vous trouvez.

#### **Poussez un commit dans GIT:**

- Appuyez sur une balise individuelle:

```
git push origin tag-name
```

- Poussez toutes les étiquettes à la fois

```
git push origin --tags
```

Lire Marquage Git en ligne: <https://riptutorial.com/fr/git/topic/10098/marquage-git>

## Chapitre 35: Mettre à jour le nom de l'objet dans la référence

### Exemples

#### Mettre à jour le nom de l'objet dans la référence

### Utilisation

Mettre à jour le nom de l'objet qui est stocké dans la référence

### SYNOPSIS

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref> <newvalue> [<oldvalue>] | --stdin [-z])
```

### Syntaxe générale

1. En déréférencant les refs symboliques, mettez à jour la branche en cours sur le nouvel objet.

```
git update-ref HEAD <newvalue>
```

2. Stocke la nouvelle newvalue dans ref , après avoir vérifié que la valeur actuelle de la ref correspond à oldvalue .

```
git update-ref refs/head/master <newvalue> <oldvalue>
```

La syntaxe ci-dessus met à jour la tête de branche principale à newvalue uniquement si sa valeur actuelle est oldvalue .

Utilisez l' -d flag pour supprimer le <ref> nommé après avoir vérifié qu'il contient toujours <oldvalue> .

Utilisez --create-reflog , update-ref créera un renvoi pour chaque ref même si on ne le créerait pas normalement.

Utilisez l' -z pour spécifier dans un format terminé par NUL, qui a des valeurs telles que update, create, delete, verify.

### Mettre à jour

Définissez <ref> sur <newvalue> après avoir vérifié <oldvalue> , si elle est donnée. Indiquez un zéro <newvalue> pour vous assurer que la référence n'existe pas après la mise à jour et / ou un zéro <oldvalue> pour vous assurer que la référence n'existe pas avant la mise à jour.

### Créer

Créez <ref> avec <newvalue> après avoir vérifié qu'il n'existe pas. La <newvalue> peut ne pas être égale à zéro.

### Effacer

Supprimez <ref> après avoir vérifié qu'elle existe avec <oldvalue> , si elle est donnée. Si donné, <oldvalue> peut ne pas être zéro.

### Vérifier

Vérifiez <ref> contre <oldvalue> mais ne le changez pas. Si <oldvalue> zéro ou manquant, la référence ne doit pas exister.

Lire Mettre à jour le nom de l'objet dans la référence en ligne:

<https://riptutorial.com/fr/git/topic/7579/mettre-a-jour-le-nom-de-l-objet-dans-la-reference>

### Exemples

#### Migrer de SVN vers Git en utilisant l'utilitaire de conversion Atlassian

Téléchargez l'utilitaire de conversion Atlassian [ici](#) . Cet utilitaire nécessite Java, donc assurez-vous que [JRE](#) Java Runtime Environment est installé sur la machine sur laquelle vous souhaitez effectuer la conversion.

Utilisez la commande `java -jar svn-migration-scripts.jar verify` pour vérifier si votre `java -jar svn-migration-scripts.jar verify` manque l'un des programmes nécessaires pour terminer la conversion. Plus précisément, cette commande vérifie les utilitaires Git, subversion et git-svn . Il vérifie également que vous effectuez la migration sur un système de fichiers sensible à la casse. La migration vers Git doit être effectuée sur un système de fichiers sensible à la casse pour éviter de corrompre le référentiel.

Ensuite, vous devez générer un fichier auteurs. Subversion suit les modifications par le nom d'utilisateur du committer uniquement. Git utilise cependant deux informations pour distinguer un utilisateur: un vrai nom et une adresse électronique. La commande suivante générera un fichier texte mappant les noms d'utilisateur subversion à leurs équivalents Git:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

où `<svn-repo>` est l'URL du dépôt de subversion que vous souhaitez convertir. Après avoir exécuté cette commande, les informations d'identification des contributeurs seront mappées dans `authors.txt` . Les adresses e-mail seront de la forme `<username>@mycompany.com` . Dans le fichier auteurs, vous devrez modifier manuellement le nom par défaut de chaque personne (qui est devenu son nom d'utilisateur par défaut) en leur donnant le nom réel. Assurez-vous également de vérifier l'exactitude de toutes les adresses électroniques avant de continuer.

La commande suivante clone un svn repo en tant que Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

où `<svn-repo>` est la même URL de référentiel utilisée ci-dessus et `<git-repo-name>` est le nom du dossier dans le répertoire actuel dans lequel cloner le référentiel. Il existe quelques considérations avant d'utiliser cette commande:

- Le `--stdlayout` drapeau de Git ci - dessus indique que vous utilisez une mise en page standard avec le trunk , les branches et tags des dossiers. Les référentiels Subversion avec des mises en page non standard nécessitent que vous spécifiez les emplacements du dossier de la ligne trunk , de tous les dossiers de branch / et de tous les dossiers de tags . Cela peut se faire en suivant l'exemple suivant: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-file=authors.txt <svn-repo> <git-repo-name>` .
- Cette commande peut prendre plusieurs heures, selon la taille de votre dépôt.
- Pour réduire le temps de conversion des référentiels volumineux, la conversion peut être exécutée directement sur le serveur hébergeant le référentiel Subversion afin d'éliminer la surcharge du réseau.

`git svn clone` importe les branches subversion (et le tronc) en tant que branches distantes, y compris les balises de subversion (branches distantes préfixées par des tags/ ). Pour les convertir en branches et balises réelles, exécutez les commandes suivantes sur un ordinateur Linux dans l'ordre dans lequel elles sont fournies. Après leur exécution, `git branch -a` devrait afficher les noms de branche corrects, et `git tag -l` devrait afficher les balises du référentiel.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname;
do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git
```

```
branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname";  
done
```

La conversion de svn à Git est maintenant terminée! Il suffit de push votre repo local vers un serveur et vous pouvez continuer à contribuer en utilisant Git ainsi que d' avoir une histoire de version entièrement préservée de svn.

### SubGit

[SubGit](#) peut être utilisé pour effectuer une importation unique d'un référentiel SVN vers git.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

### Migrer de SVN à Git en utilisant svn2git

[svn2git](#) est un wrapper Ruby autour du support SVN natif de [git](#) via [git-svn](#) .

#### Exemples

Pour migrer un référentiel svn avec la disposition standard (par exemple, les branches, les balises et le tronc au niveau racine du référentiel):

```
$ svn2git http://svn.example.com/path/to/repo
```

Pour migrer un référentiel svn qui n'est pas dans une présentation standard:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches  
branches-dir
```

Si vous ne souhaitez pas migrer (ou ne pas avoir) de branches, de balises ou de tronc, vous pouvez utiliser les options `--notrunk` , `--nobranches` et `--notags` .

Par exemple, `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` ne migrera que l'historique des troncs.

Pour réduire l'espace requis par votre nouveau référentiel, vous souhaitez peut-être exclure les répertoires ou fichiers que vous avez ajoutés une fois que vous n'auriez pas dû (par exemple, répertoire de construction ou archives):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

### Optimisation post-migration

Si vous avez déjà quelques milliers de commits (ou plus) dans votre référentiel git nouvellement créé, vous souhaitez peut-être réduire l'espace utilisé avant de transférer votre référentiel sur une télécommande. Cela peut être fait en utilisant la commande suivante:

```
$ git gc --aggressive
```

**Remarque:** la commande précédente peut durer plusieurs heures sur de grands référentiels (des dizaines de milliers de commits et / ou des centaines de mégaoctets d'historique).

### Migration de Team Foundation Version Control (TFVC) vers Git

Vous pouvez migrer du contrôle de version Team Foundation vers GIT en utilisant un outil open source appelé Git-TF. La migration transférera également votre historique existant en

convertissant les checkins tfs en git commits.

Pour mettre votre solution dans Git en utilisant Git-TF, suivez ces étapes:

### Télécharger Git-TF

Vous pouvez télécharger (et installer) Git-TF à partir de Codeplex: [Git-TF @ Codeplex](#)

### Cloner votre solution TFVC

Lancer powershell (win) et tapez la commande

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection
'$/myproject/mybranch/mysolution' --deep
```

Le paramètre --deep est le mot-clé à noter car il indique à Git-Tf de copier votre historique de vérification. Vous avez maintenant un référentiel git local dans le dossier à partir duquel vous avez appelé votre commande clon.

### Nettoyer

- Ajoutez un fichier .gitignore. Si vous utilisez Visual Studio, l'éditeur peut le faire pour vous, sinon vous pouvez le faire manuellement en téléchargeant un fichier complet à partir de [github / gitignore](#) .
- Liaisons de contrôle de source RemoveTFS à partir de la solution (supprimez tous les fichiers \* .vsscc). Vous pouvez également modifier votre fichier de solution en supprimant le GlobalSection (TeamFoundationVersionControl) ..... EndGlobalSection

### Commit & Push

Terminez votre conversion en validant et en transférant votre référentiel local sur votre télécommande.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

### Migration de Mercurial à Git

On peut utiliser les méthodes suivantes pour importer un Repo Mercurial dans Git :

1. En utilisant l' [exportation rapide](#) :

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. Utiliser [Hg-Git](#) : Une réponse très détaillée ici: <https://stackoverflow.com/a/31827990/5283213>
3. Utilisation de l'[importateur de GitHub](#) : Suivez les instructions (détaillées) sur [GitHub](#) .

Lire Migration vers Git en ligne: <https://riptutorial.com/fr/git/topic/3026/migration-vers-git>



## Chapitre 37: Mise en scène

### Remarques

Il convient de noter que la mise en scène a peu à voir avec les «fichiers» eux-mêmes et tout ce qui concerne les modifications apportées à chaque fichier. Nous montons les fichiers qui contiennent des modifications et git suit les modifications en tant que modifications (même lorsque les modifications apportées à une validation sont effectuées sur plusieurs fichiers).

La distinction entre les fichiers et les commits peut sembler mineure, mais la compréhension de cette différence est fondamentale pour comprendre les fonctions essentielles telles que la sélection et la diffusion de cerises. (Voir la frustration dans les [commentaires concernant la complexité d'une réponse acceptée qui propose un choix sélectif comme outil de gestion de fichiers](#) .)

Quel est l'endroit idéal pour expliquer des concepts? Est-ce dans les remarques?

Concepts clés:

Un fichier est la métaphore la plus commune des deux dans la technologie de l'information. Les meilleures pratiques imposent qu'un nom de fichier ne change pas à mesure que son contenu change (avec quelques exceptions reconnues).

Un commit est une métaphore unique à la gestion du code source. Les validations sont des modifications liées à un effort spécifique, comme une correction de bogue. Les commits impliquent souvent plusieurs fichiers. Un seul correctif de bogue mineur peut impliquer des modifications des modèles et des CSS dans des fichiers uniques. À mesure que le changement est décrit, développé, documenté, examiné et déployé, les modifications apportées aux différents fichiers peuvent être annotées et traitées comme une seule unité. L'unité unique dans ce cas est la validation. Tout aussi important, se concentrer uniquement sur la validation lors d'une révision permet d'ignorer les lignes de code inchangées dans les différents fichiers affectés.

### Exemples

#### Mise en scène d'un seul fichier

Pour mettre en place un fichier pour commettre, exécutez

```
git add <filename>
```

#### Mise en place de toutes les modifications apportées aux fichiers

```
git add -A
```

2.0

```
git add .
```

Dans la version 2.x, `git add .` mettra en scène toutes les modifications apportées aux fichiers du répertoire en cours et de tous ses sous-répertoires. Cependant, dans 1.x, il ne mettra en scène [que des fichiers nouveaux et modifiés, pas des fichiers supprimés](#) .

Utilisez `git add -A` , ou sa commande équivalente `git add --all` , pour mettre en place toutes les modifications apportées aux fichiers dans n'importe quelle version de git.

#### Stade fichiers supprimés

```
git rm filename
```

Pour supprimer le fichier de git sans le retirer du disque, utilisez l'indicateur `--cached`

```
git rm --cached filename
```

### Dégager un fichier contenant des modifications

```
git reset <filePath>
```

### Ajout interactif

`git add -i` (ou `--interactive`) vous donnera une interface interactive où vous pourrez éditer l'index, pour préparer ce que vous voulez avoir dans le prochain commit. Vous pouvez ajouter et supprimer des modifications à des fichiers entiers, ajouter des fichiers non suivis et supprimer le suivi des fichiers, mais également sélectionner la sous-section de modifications à insérer dans l'index, en sélectionnant des fragments de modifications à ajouter, . De nombreux outils de création graphique pour Git (comme par exemple `git gui`) incluent une telle fonctionnalité; Cela peut être plus facile à utiliser que la version en ligne de commande.

C'est très utile (1) si vous avez des modifications empêchées dans le répertoire de travail que vous voulez mettre dans des commits séparés, et pas toutes dans un seul commit (2) si vous êtes en cours de rebase interactif et que vous voulez vous séparer gros commit.

```
$ git add -i
      staged      unstaged path
 1:   unchanged    +4/-4 index.js
 2:       +1/-0     nothing package.json

*** Commands ***
 1: status          2: update          3: revert          4: add untracked
 5: patch           6: diff            7: quit            8: help
What now>
```

La moitié supérieure de cette sortie indique l'état actuel de l'index divisé en colonnes à l'étage ou non:

1. `index.js` 4 lignes à `index.js` et suppression de 4 lignes. Il n'est actuellement pas mis en scène, car l'état actuel indique "inchangé". Lorsque ce fichier est mis en scène, le bit `+4/-4` sera transféré dans la colonne intermédiaire et la colonne non-stadonnée indiquera "rien".
2. Une ligne a été ajoutée à `package.json` et a été mise en scène. Il n'y a pas d'autres modifications car il a été mis en scène comme indiqué par la ligne "Nothing" sous la colonne non-staged.

La moitié inférieure montre ce que vous pouvez faire. Entrez un nombre (1-8) ou une lettre ( `s`, `u`, `r`, `a`, `p`, `d`, `q`, `h` ).

`status` indique une sortie identique à la partie supérieure de la sortie ci-dessus.

`update` à `update` vous permet d'apporter d'autres modifications aux commits par étapes avec une syntaxe supplémentaire.

`revert` rétablira les informations de validation par étapes sur `HEAD`.

`add untracked` vous permet d'ajouter des chemins de fichiers précédemment non suivis par le contrôle de version.

`patch` permet de sélectionner un chemin parmi une sortie similaire à l' `status` pour une analyse plus approfondie.

`diff` affiche ce qui sera commisé.

quit quitte la commande.

help présente une aide supplémentaire sur l'utilisation de cette commande.

### Ajouter des modifications par morceau

Vous pouvez voir ce que des "morceaux" de travail seraient mis en scène pour une validation en utilisant l'indicateur de patch:

```
git add -p
```

ou

```
git add --patch
```

Cela ouvre une invite interactive qui vous permet de regarder les diffs et vous permet de décider si vous souhaitez les inclure ou non.

```
Stage this hunk [y,n,q,a,d,/,s,e,]?
```

- y mettre en scène ce morceau pour le prochain commit
- n ne pas mettre en scène ce morceau pour le prochain commit
- q quitter; ne pas mettre en scène ce morceau ou l'un des autres mecs
- une scène ce morceau et tous les derniers mecs dans le fichier
- d ne pas mettre en scène ce morceau ou l'un des derniers morceaux du fichier
- g sélectionnez un morceau pour aller à
- / recherche d'un morceau correspondant à l'expression régulière donnée
- j laisse ce morceau indécis, vois le prochain morceau indécis
- J laisse ce morceau indécis, vois prochain morceau
- k laisser ce morceau indécis, voir le morceau précédent indécis
- K laisse ce morceau indécis, voir le morceau précédent
- s diviser le morceau actuel en plus petits mecs
- e modifier manuellement le morceau actuel
- ? aide à la copie

*Cela facilite la capture des modifications que vous ne voulez pas commettre.*

Vous pouvez également ouvrir cette page via `git add --interactive` et en sélectionnant `p`.

### Afficher les modifications par étapes

Pour afficher les points d'accès mis en attente pour la validation:

```
git diff --cached
```

Lire Mise en scène en ligne: <https://riptutorial.com/fr/git/topic/244/mise-en-scene>

## Chapitre 38: Montrer

### Syntaxe

- `git show [options] <objet> ...`

### Remarques

Affiche divers objets Git.

- Pour les commits, affiche le message de validation et diff
- Pour les balises, affiche le message de balise et l'objet référencé

### Exemples

#### Vue d'ensemble

`git show` montre divers objets Git.

---

#### Pour les commits:

Affiche le message de validation et un diff des modifications apportées.

| Commander                 | La description                       |
|---------------------------|--------------------------------------|
| <code>git show</code>     | montre l'engagement précédent        |
| <code>git show @~3</code> | montre le 3ème de dernier engagement |

#### Pour les arbres et les blobs:

Affiche l'arbre ou le blob.

| Commander                                | La description  |
|--|---|
| <code>git show @~3:</code>               | affiche le répertoire racine du projet tel qu'il était 3 commits (un arbre)           |
| <code>git show @~3:src/program.js</code> | montre <code>src/program.js</code> comme il y a 3 commits (un blob)                   |
| <code>git show @:a.txt @:b.txt</code>    | montre <code>a.txt</code> concaténé avec <code>b.txt</code> partir de commit en cours |

#### Pour les tags:

Affiche le message de balise et l'objet référencé.

Lire Montrer en ligne: <https://riptutorial.com/fr/git/topic/3030/montrer>

## Chapitre 39: Nom de la branche Git sur Bash Ubuntu

### Introduction

Cette documentation traite du **nom** de **branche** de git sur le terminal **bash** . Nous, développeurs, devons trouver le nom de la branche git très fréquemment. Nous pouvons ajouter le nom de la branche avec le chemin d'accès au répertoire en cours.

### Exemples

#### Nom de la succursale dans le terminal

#### Qu'est ce que c'est PS1

PS1 désigne la chaîne d'invite 1. C'est celle de l'invite disponible dans le shell Linux / UNIX. Lorsque vous ouvrez votre terminal, il affiche le contenu défini dans la variable PS1 dans votre invite bash. Pour ajouter un nom de branche à l'invite bash, nous devons éditer la variable PS1 (valeur de définition de PS1 dans ~ / .bash\_profile).

#### Afficher le nom de la branche git

Ajoutez les lignes suivantes à votre fichier ~ / .bash\_profile

```
git_branch() {  
  git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'  
}  
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\${git_branch}\[\033[00m\] $ "
```

Cette fonction git\_branch trouvera le nom de la branche sur laquelle nous sommes. Une fois que nous en avons terminé avec ces modifications, nous pouvons contourner le dépôt git sur le terminal et voir le nom de la branche.

Lire Nom de la branche Git sur Bash Ubuntu en ligne:

<https://riptutorial.com/fr/git/topic/8320/nom-de-la-branche-git-sur-bash-ubuntu>

## Chapitre 40: Parcourir l'historique

### Syntaxe

- `git log [options] [plage de révision] [[-] chemin ...]`

### Paramètres

| Paramètre                              | Explication  |
|--|--|
| <code>-q, --quiet</code>               | Silencieux, supprime la sortie diff  |
| <code>--la source</code>               | Affiche la source du commit  |
| <code>--use-mailmap</code>             | Utiliser un fichier de carte de messagerie (modifie les informations utilisateur pour l'utilisateur en cours de validation)  |
| <code>--décorer [= ...]</code>         | Options de décoration  |
| <code>--L &lt;n, m: fichier&gt;</code> | Afficher le journal pour une plage spécifique de lignes dans un fichier, en partant de 1. Commence à partir de la ligne n, passe à la ligne m. Affiche également diff. |
| <code>--show-signature</code>          | Afficher les signatures des validations signées  |
| <code>-i, --regexp-ignore-case</code>  | Faire correspondre les schémas de limitation des expressions régulières sans tenir compte de la casse des lettres  |

### Remarques

Références et **documentation** à jour: [documentation officielle de git-log](#)

### Exemples

"Git Log" régulier

```
git log
```

affichera tous vos commits avec l'auteur et le hachage. Cela sera affiché sur plusieurs lignes par validation. (Si vous souhaitez afficher une seule ligne par validation, consultez la [rubrique en ligne](#) ). Utilisez la touche q pour quitter le journal.

Par défaut, sans arguments, `git log` répertorie les validations effectuées dans ce référentiel dans l'ordre chronologique inverse, c'est-à-dire que les validations les plus récentes apparaissent en premier. Comme vous pouvez le voir, cette commande répertorie chaque validation avec sa somme de contrôle SHA-1, le nom et le courrier électronique de l'auteur, la date écrite et le message de validation. - [source](#)

Exemple (du référentiel [Free Code Camp](#) ):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian <sludge256@users.noreply.github.com>
Date: Thu Mar 24 15:52:07 2016 -0700
```

Merge pull request #7724 from BKinahan/fix/where-art-thou

Fix 'its' typo in Where Art Thou description

```
commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan <b.kinahan@gmail.com>
Date: Thu Mar 24 21:11:36 2016 +0000
```

Fix 'its' typo in Where Art Thou description

```
commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra <raisedadead@users.noreply.github.com>
Date: Thu Mar 24 14:26:04 2016 +0530
```

Merge pull request #7718 from deathsythe47/fix/unnecessary-comma

Remove unnecessary comma from CONTRIBUTING.md

Si vous souhaitez limiter votre commande au dernier n engage journal que vous pouvez simplement passer un paramètre. Par exemple, si vous souhaitez lister les 2 derniers journaux de validation

```
git log -2
```

### Journal en ligne

```
git log --oneline
```

affichera tous vos commits avec seulement la première partie du hachage et le message de validation. Chaque validation sera sur une seule ligne, comme l' oneline indicateur en ligne.

L'option en ligne imprime chaque validation sur une seule ligne, ce qui est utile si vous consultez beaucoup de validations. - [source](#)

Exemple (du référentiel [Free Code Camp](#) , avec la même section de code de l'autre exemple):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

Si vous souhaitez vous limiter ordonnées dernier n engage journal que vous pouvez simplement passer un paramètre. Par exemple, si vous souhaitez lister les 2 derniers journaux de validation

```
git log -2 --oneline
```

### Journal plus joli

Pour voir le journal dans une plus belle structure graphique, utilisez:

```
git log --decorate --oneline --graph
```

sortie de l'échantillon:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
* 9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* | f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Comme c'est une commande assez importante, vous pouvez assigner un alias:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

Pour utiliser la version alias:

```
# history of current branch :
git lol

# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master

# combined history of everything in your repo :
git lol --all
```

### Connectez-vous avec les modifications en ligne

Pour voir le journal avec les modifications en ligne, utilisez les options `-p` ou `--patch`.

```
git log --patch
```

Exemple (extrait du dépôt [Trello Scientist](#))

```
commit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date: Wed Mar 2 10:35:25 2016 -0800

    fix readme error link

diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but after testing the other functions and
reading
the logging. The criteria for matching errors is based on the constructor and
message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).
```



```
## Asynchronous behaviors
```

```
commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa  
:
```

## Recherche de journal

```
git log -S"#define SAMPLES"
```

Recherche l' **ajout** ou la **suppression** d'une chaîne spécifique ou la chaîne **correspondante** fournie par REGEXP. Dans ce cas, nous cherchons à ajouter / supprimer la chaîne #define SAMPLES . Par exemple:

```
+#define SAMPLES 100000
```

ou

```
+#define SAMPLES 100000
```

```
git log -G"#define SAMPLES"
```

Recherche les **modifications** dans les **lignes contenant** une chaîne spécifique ou la chaîne **correspondant à** REGEXP. Par exemple:

```
+#define SAMPLES 100000  
+#define SAMPLES 100000000
```

## Liste toutes les contributions regroupées par nom d'auteur

git shortlog résume git log et groupes par auteur

Si aucun paramètre n'est donné, une liste de tous les commits effectués par committer sera affichée dans l'ordre chronologique.

```
$ git shortlog  
Committer 1 (<number_of_commits>):  
  Commit Message 1  
  Commit Message 2  
  ...  
Committer 2 (<number_of_commits>):  
  Commit Message 1  
  Commit Message 2  
  ...
```

Pour voir simplement le nombre de commits et supprimer la description de la validation, passez l'option de résumé:

```
-s
```

```
--summary
```

```
$ git shortlog -s
```

```
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

---

Pour trier la sortie par nombre de validations plutôt que par ordre alphabétique par nom de composant, indiquez l'option numérotée:

```
-n
--numbered
```

---

Pour ajouter l'e-mail d'un committer, ajoutez l'option email:

```
-e
--email
```

---

Une option de format personnalisé peut également être fournie si vous souhaitez afficher des informations autres que le sujet de validation:

```
--format
```

Cela peut être n'importe quelle chaîne acceptée par l'option `--format` de `git log` .

Voir [Colorisation des journaux](#) ci-dessus pour plus d'informations à ce sujet.

### Filterer les journaux

```
git log --after '3 days ago'
```

Les dates spécifiques fonctionnent aussi:

```
git log --after 2016-05-01
```

Comme avec les autres commandes et indicateurs qui acceptent un paramètre de date, le format de date autorisé est pris en charge par la date GNU (très flexible).

Un alias `--after` est `--since` .

Les drapeaux existent pour l'inverse aussi: `--before` et `--until` .

Vous pouvez également filtrer les journaux par `author` . par exemple

```
git log --author=author
```

### Journal d'une plage de lignes dans un fichier

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date: Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
```

```

<!DOCTYPE HTML>
<html>
-   <head>
-   <meta charset="utf-8">
+
+<head>
+   <meta charset="utf-8">
+   <meta http-equiv="X-UA-Compatible" content="IE=edge">
+   <meta name="viewport" content="width=device-width, initial-scale=1">

```

## Coloriser les journaux

```

git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green) (%cr)
%C(yellow)<%an>%Creset '

```

L'option format vous permet de spécifier votre propre format de sortie de journal :

| Paramètre      | Détails   |
|----------------|---|
| %C(color_name) | l'option colore la sortie qui suit                                      |
| %h ou% H       | abréviations commettre le hachage (utiliser% H pour le hachage complet) |
| %Creset        | réinitialise la couleur à la couleur du terminal par défaut             |
| %d             | noms de ref   |
| %s             | sujet [commettre un message]  |
| %cr            | date d'engagement, par rapport à la date actuelle                       |
| %an            | nom de l'auteur   |

Une ligne indiquant le nom du commetteur et l'heure depuis la validation

```

tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"' --all --graph

```

Exemple

```

*   40554ac  3 months ago  Alexander Zolotov  Merge pull request #95 from
gmandnepr/external_plugins
|\
| *   e509f61  3 months ago  Ievgen Degtiarenko  Documenting new property
| *   46d4cb6  3 months ago  Ievgen Degtiarenko  Running idea with external plugins
| *   6253da4  3 months ago  Ievgen Degtiarenko  Resolve external plugin classes
| *   9fdb4e7  3 months ago  Ievgen Degtiarenko  Keep original artifact name as this may be
important for intellij
| *   22e82e4  3 months ago  Ievgen Degtiarenko  Declaring external plugin in intellij
section
|/
*   bc3d2cb  3 months ago  Alexander Zolotov  Ignore DTD in plugin.xml

```

## Git Log Entre Deux Branches

git log master..foo affichera les commits qui sont sur foo et non sur master . Utile pour voir ce que vous avez ajouté depuis le branchement!

## Journal affichant les fichiers validés

```
git log --stat
```

Exemple:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Mon Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk dependency

mltracking-poc/.gitignore | 1 +
mltracking-poc/pom.xml    | 14 ++++++++--
2 files changed, 13 insertions(+), 2 deletions(-)

commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Sat Jun 4 12:35:16 2016 -0300

    [manasses] generated by SpringBoot initializr

.gitignore | 42
+++++++
mltracking-poc/mvnw | 233
+++++++
mltracking-poc/mvnw.cmd | 145
+++++++
mltracking-poc/pom.xml | 74
+++++++
mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java | 12
++++
mltracking-poc/src/main/resources/application.properties | 0
mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java | 18
++++
7 files changed, 524 insertions(+)
```

## Afficher le contenu d'un seul commit

En utilisant `git show` on peut voir un seul commit

```
git show 48c83b3
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Exemple

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.
```

```
diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

                colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                colorMap.put(BuildStatus.SUCCESS, Color.decode( "#55FF55" ));
+                colorMap.put(BuildStatus.SUCCESS, Color.decode( "#33CC33" ));
                colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

### Recherche d'une chaîne de validation dans le journal git

Recherche dans le journal git en utilisant une chaîne de caractères dans le journal:

```
git log [options] --grep "search_string"
```

Exemple:

```
git log --all --grep "removed file"
```

Recherche la chaîne de removed file dans **tous les journaux** de **toutes les branches** .

---

À partir de git 2.4+, la recherche peut être inversée à l'aide de l'option `--invert-grep` .

Exemple:

```
git log --grep="add file" --invert-grep
```

Affiche tous les commits ne contenant pas de add file .

Lire [Parcourir l'historique en ligne](https://riptutorial.com/fr/git/topic/240/parcourir-l-historique): <https://riptutorial.com/fr/git/topic/240/parcourir-l-historique>

Syntaxe

- `git am [--signoff] [--keep] [- [no-] keep-cr] [- [no-] utf8] [--3way] [--interactive] [--committer-date-is -author-date] [--ignore-date] [--ignore-space-change | --ignore-whitespace] [--whitespace = <option>] [-C <n>] [-p <n>] [--directory = <dir>] [--exclude = <chemin>] [- include = <chemin>] [--reject] [-q | --quiet] [- [no-] ciseaux] [-S [<keyid>]] [--patch-format = <format>] [( <mbox> | <Maildir>) ...]`
- `git am (--continue | --skip | --abort)`

Paramètres

| Paramètre   | Détails   |
|---|---|
| <code>(&lt;mbox&gt;   &lt;Maildir&gt;) ...</code>   | La liste des fichiers de boîtes aux lettres à partir desquels lire les correctifs. Si vous ne fournissez pas cet argument, la commande lit à partir de l'entrée standard. Si vous fournissez des répertoires, ils seront traités comme des publipostages.   |
| <code>-s, --signoff</code>  | Ajoutez une ligne Signed-by-by: au message de validation, en utilisant l'identité du committer de vous-même.  |
| <code>-q, --quiet</code>  | Soyez silencieux. N'imprimez que les messages d'erreur.   |
| <code>-u, --utf8</code>   | Passer le drapeau <code>-u</code> à <code>git mailinfo</code> . Le message de journal de validation proposé à partir du courrier électronique est <code>i18n.commitencoding</code> en codage UTF-8 (la variable de configuration <code>i18n.commitencoding</code> peut être utilisée pour spécifier le codage préféré du projet s'il ne s'agit pas du format UTF-8). Vous pouvez utiliser <code>--no-utf8</code> pour remplacer ceci. |
| <code>--no-utf8</code>  | Passez le drapeau <code>-n</code> à <code>git mailinfo</code> .   |
| <code>-3, - 3 voies</code>  | Lorsque le correctif ne s'applique pas proprement, retombez sur la fusion 3-way si le patch enregistre l'identité des blobs auxquels il est supposé s'appliquer et que ces blobs sont disponibles localement.   |
| <code>--ignore-date, --ignore-space-change, --ignore-whitespace, --whitespace = &lt;option&gt;, -C &lt;n&gt;, -p &lt;n&gt;, --directory = &lt;dir&gt;, -exclude = &lt;chemin&gt;, --include = &lt;chemin&gt;, --reject</code> | Ces indicateurs sont transmis au programme d'application git qui applique le correctif.   |
| <code>--patch-format</code>   | Par défaut, la commande essaiera de détecter automatiquement le format du patch. Cette option permet à  |

| Paramètre   | Détails   |
|---|---|
|   | l'utilisateur de contourner la détection automatique et de spécifier le format de patch selon lequel les patches doivent être interprétés. Les formats valides sont <code>mbox</code> , <code>stgit</code> , <code>stgit-series</code> et <code>hg</code> .   |
| <code>-i, --interactif</code>                                 | Exécuter de manière interactive   |
| <code>--committer-date-is-author-date</code>                  | Par défaut, la commande enregistre la date du message électronique en tant que date de création de l'auteur et utilise l'heure de création de validation en tant que date du committer. Cela permet à l'utilisateur de mentir sur la date du committer en utilisant la même valeur que la date de l'auteur.   |
| <code>--nombre-date</code>                                    | Par défaut, la commande enregistre la date du message électronique en tant que date de création de l'auteur et utilise l'heure de création de validation en tant que date du committer. Cela permet à l'utilisateur de mentir sur la date de l'auteur en utilisant la même valeur que la date du committer.   |
| <code>--sauter</code>   | Ignorez le patch actuel. Cela n'a de sens que lors du redémarrage d'un patch abandonné.   |
| <code>-S [&lt;keyid&gt;], --gpg-sign [= &lt;keyid&gt;]</code> | GPG-sign s'engage.  |
| <code>--continuer, -r, --résolu</code>                        | Après un échec de patch (par exemple en essayant d'appliquer un patch en conflit), l'utilisateur l'a appliqué manuellement et le fichier d'index stocke le résultat de l'application. Effectuez une validation à l'aide du journal d'auteur et du journal de validation extrait du message électronique et du fichier d'index actuel, puis continuez. |
| <code>--resolvemsg = &lt;msg&gt;</code>                       | En cas de défaillance d'un patch, <code>&lt;msg&gt;</code> sera imprimé sur l'écran avant de quitter. Cela remplace le message standard vous informant d'utiliser <code>--continue</code> ou <code>--skip</code> pour gérer l'échec. Ceci est uniquement pour un usage interne entre <code>git rebase</code> et <code>git am</code> .                 |
| <code>--avorter</code>  | Restaurez la branche d'origine et abandonnez l'opération de correction.   |

## Exemples

Créer un patch

Pour créer un patch, il y a deux étapes.

1. Faites vos changements et engagez-les.
2. Exécutez `git format-patch <commit-reference>` pour convertir tous les commits depuis le commit <référence de validation> (sans l'inclure) en fichiers de correctifs.

Par exemple, si des correctifs doivent être générés à partir des deux dernières validations:

```
git format-patch HEAD~~
```

Cela va créer 2 fichiers, un pour chaque commit depuis HEAD~~ , comme ceci:

```
0001-hello_world.patch  
0002-beginning.patch
```

### Appliquer des patches

Nous pouvons utiliser `git apply some.patch` pour que les modifications du fichier `.patch` appliquées à votre répertoire de travail actuel. Ils ne seront pas mis en scène et devront être engagés.

Pour appliquer un patch en tant que commit (avec son message de validation), utilisez

```
git am some.patch
```

Pour appliquer tous les fichiers de correctifs à l'arborescence:

```
git am *.patch
```

Lire Patch Git en ligne: <https://riptutorial.com/fr/git/topic/4603/patch-git>



### Exemples

#### Annuler les fusions

##### Annulation d'une fusion pas encore poussée sur une télécommande

Si vous n'avez pas encore poussé votre fusion vers le référentiel distant, vous pouvez suivre la même procédure que pour [annuler la validation](#) bien qu'il y ait des différences subtiles.

Une réinitialisation est l'option la plus simple car elle annulera à la fois la validation de la fusion et tous les commits ajoutés depuis la branche. Cependant, vous devrez savoir ce que SHA doit restaurer, cela peut être difficile car votre git log affichera désormais les commits des deux branches. Si vous effectuez une remise incorrecte (par exemple, sur l'autre branche), **cela peut détruire le travail engagé.**

```
> git reset --hard <last commit from the branch you are on>
```

Ou, en supposant que la fusion était votre engagement le plus récent.

```
> git reset HEAD~
```

Un retour est plus sûr, dans la mesure où il ne détruit pas le travail engagé, mais implique plus de travail car vous devez annuler le retour avant de pouvoir réintégrer la branche (voir la section suivante).

##### Annulation d'une fusion poussée sur une télécommande

Supposons que vous fusionnez une nouvelle fonctionnalité (add-gremlins)

```
> git merge feature/add-gremlins
...
#Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
501b75d..17a51fd master -> master
```

Ensuite, vous découvrirez que la fonctionnalité que vous venez de fusionner a cassé le système pour les autres développeurs, elle doit être annulée immédiatement, et la correction de la fonctionnalité prendra trop de temps, vous souhaiterez donc simplement annuler la fusion.

```
> git revert -m 1 17a51fd
...
> git push
...
17a51fd..e443799 master -> master
```

A ce stade, les gremlins sont hors du système et vos collègues développeurs ont cessé de vous crier dessus. Cependant, nous n'avons pas encore fini. Une fois que vous avez résolu le problème avec la fonctionnalité add-gremlins, vous devez annuler ce retour avant de pouvoir le réintégrer.

```
> git checkout feature/add-gremlins
...
```

```
#Various commits to fix the bug.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
#Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

À ce stade, votre fonctionnalité est maintenant ajoutée avec succès. Cependant, étant donné que les bogues de ce type sont souvent introduits par des conflits de fusion, un flux de travail légèrement différent est parfois plus utile car il vous permet de résoudre le conflit de fusion sur votre branche.

```
> git checkout feature/add-gremlins
...
#Merge in master and revert the revert right away. This puts your branch in
#the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
#Now go ahead and fix the bug (various commits go here)
> git checkout master
...
#Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
#Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

### Utiliser le reflog

Si vous boussez une rebase, une option pour recommencer est de revenir à la validation (pre rebase). Vous pouvez le faire en utilisant le reflog (qui a l'historique de tout ce que vous avez fait au cours des 90 derniers jours - cela peut être configuré):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

Vous pouvez voir le commit avant que le rebase ne soit HEAD@{3} (vous pouvez également vérifier le hash):

```
git checkout HEAD@{3}
```

Maintenant, vous créez une nouvelle branche / supprimez l'ancienne / essayez à nouveau le rebase.

Vous pouvez également revenir directement à un point de votre reflog, mais ne le faites que si

vous êtes sûr à 100% que c'est ce que vous voulez faire:

```
git reset --hard HEAD@{3}
```

Cela définira votre arbre git actuel pour qu'il corresponde à ce qu'il était à ce moment-là (voir Annuler les modifications).

Cela peut être utilisé si vous voyez temporairement le fonctionnement d'une branche lors d'un rebasage sur une autre branche, mais vous ne voulez pas conserver les résultats.

### Retour à un engagement précédent

Pour revenir à un commit précédent, commencez par trouver le hachage du commit à l'aide de [git log](#) .

Pour revenir temporairement à cet engagement, détachez votre tête avec:

```
git checkout 789abcd
```

Cela vous place à 789abcd . Vous pouvez désormais créer de nouveaux commits en plus de cet ancien commit sans affecter la branche sur laquelle vous vous trouvez. Toute modification peut être effectuée dans une branche appropriée en utilisant [soit branch ou checkout -b](#) .

Pour revenir à un commit précédent tout en conservant les modifications:

```
git reset --soft 789abcd
```

Pour annuler le **dernier** commit:

```
git reset --soft HEAD~
```

Pour ignorer de manière permanente les modifications apportées après une validation spécifique, utilisez:

```
git reset --hard 789abcd
```

Pour ignorer de manière permanente les modifications apportées après la **dernière** validation:

```
git reset --hard HEAD~
```

**Attention:** Bien que vous puissiez [récupérer les reflog abandonnées en utilisant le reflog et la reset](#) , les modifications non validées ne peuvent pas être récupérées. Utilisez [git stash; git reset](#) au lieu de `git reset --hard` pour être sûr.

### Annuler les modifications

Annuler les modifications apportées à un fichier ou à un répertoire dans la **copie de travail** .

```
git checkout -- file.txt
```

Utilisé sur tous les chemins de fichiers, récursivement à partir du répertoire en cours, il annulera toutes les modifications de la copie de travail.

```
git checkout -- .
```

Pour annuler uniquement certaines parties des modifications, utilisez `--patch` . On vous demandera, pour chaque changement, si elle doit être annulée ou non.

```
git checkout --patch -- dir
```

Pour annuler les modifications ajoutées à l' **index** .

```
git reset --hard
```

Sans l'indicateur `--hard` , cela effectuera une réinitialisation `--hard` .

Avec les commits locaux que vous n'avez pas encore transmis à une télécommande, vous pouvez également effectuer une réinitialisation logicielle. Vous pouvez ainsi retravailler les fichiers puis les commits.

```
git reset HEAD~2
```

L'exemple ci-dessus détacherait vos deux derniers commits et renverrait les fichiers à votre copie de travail. Vous pourriez alors apporter d'autres modifications et de nouveaux commits.

**Attention:** toutes ces opérations, à l'exception des réinitialisations logicielles, suppriment définitivement vos modifications. Pour une option plus sûre, utilisez respectivement `git stash -p` ou `git stash` . Vous pouvez plus tard annuler avec de la `stash pop` ou la supprimer pour toujours avec la suppression de la `stash drop` .

#### Inverser certains commits existants

Utilisez `git revert` pour annuler les validations existantes, en particulier lorsque ces validations ont été envoyées dans un référentiel distant. Il enregistre de nouveaux commits pour inverser l'effet de certains commits antérieurs, que vous pouvez utiliser en toute sécurité sans réécrire l'historique.

**N'utilisez pas** `git push --force` moins que vous ne souhaitiez réduire l'opprobre de tous les autres utilisateurs de ce référentiel. Ne jamais réécrire l'histoire publique.

Si, par exemple, vous venez de lancer une validation qui contient un bogue et que vous devez la sauvegarder, procédez comme suit:

```
git revert HEAD~1
git push
```

Maintenant, vous êtes libre de revenir sur la validation de retour localement, de corriger votre code et de pousser le bon code:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

Si la validation que vous souhaitez rétablir est déjà plus ancienne, vous pouvez simplement passer le hachage de validation. Git créera une contre-validation annulant votre validation initiale, que vous pourrez transmettre à votre télécommande en toute sécurité.

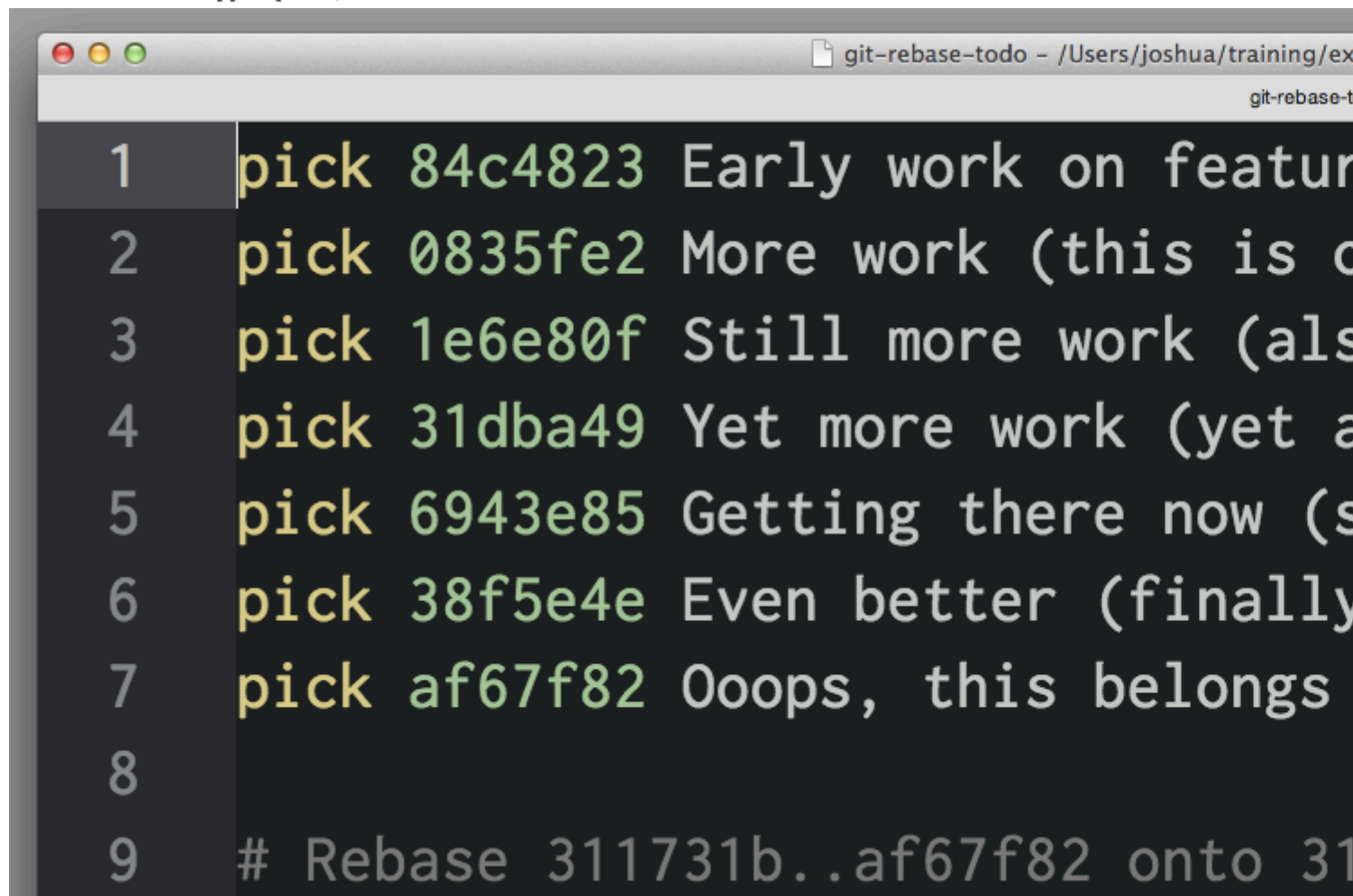
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

## Annuler / Refaire une série de commits

Supposons que vous vouliez annuler une douzaine de commits et que vous ne souhaitiez que certains d'entre eux.

```
git rebase -i <earlier SHA>
```

`-i` met rebase en "mode interactif". Il commence comme le rebase discuté ci-dessus, mais avant de rejouer les commits, il se met en pause et vous permet de modifier en douceur chaque validation lors de sa relecture. `rebase -i` s'ouvrira dans votre éditeur de texte par défaut, avec une liste de validations appliquées, comme ceci:



```
git-rebase-todo - /Users/joshua/training/ex
git-rebase-t
1 pick 84c4823 Early work on featur
2 pick 0835fe2 More work (this is o
3 pick 1e6e80f Still more work (als
4 pick 31dba49 Yet more work (yet a
5 pick 6943e85 Getting there now (s
6 pick 38f5e4e Even better (finally
7 pick af67f82 Ooops, this belongs
8
9 # Rebase 311731b..af67f82 onto 31
```

Pour supprimer un commit, supprimez simplement cette ligne dans votre éditeur. Si vous ne voulez plus les commits incorrects dans votre projet, vous pouvez supprimer les lignes 1 et 3-4 ci-dessus. Si vous souhaitez combiner deux commits, vous pouvez utiliser les commandes de squash ou de fixup

```
git-rebase-todo - /Users/joshua/training/ex
git-rebase-t
1 pick 0835fe2 More work (this is o
2 squash 6943e85 Getting there now
3 pick 38f5e4e Even better (finally
4 fixup| af67f82 Ooops, this belongs
5
6 # Rebase 311731b..af67f82 onto 31
```

Lire Perte en ligne: <https://riptutorial.com/fr/git/topic/285/perte>

### Syntaxe

- `git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]`
  - `git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]`
  - `git branch --unset-upstream [<branchname>]`
  - `git branch (-m | -M) [<oldbranch>] <newbranch>`
  - `git branch (-d | -D) [-r] <branchname>...`
  - `git branch --edit-description [<branchname>]`
  - `git branch [--color[=<when>] | --no-color] [-r | -a] [--list] [-v [--abbrev=<length> | --no-abbrev]] [--column[=<options>] | --no-column] [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>] [--points-at <object>] [<pattern>...]`

### Paramètres

| Paramètre                   | Détails  |
|-----------------------------|--|
| <code>-d, --delete</code>   | Supprimer une branche. La branche doit être complètement fusionnée dans sa branche amont, ou dans <code>HEAD</code> si aucun amont n'a été défini avec <code>--track</code> ou <code>--set-upstream</code>   |
| <code>-RE</code>            | Raccourci pour <code>--delete --force</code>   |
| <code>-m, --move</code>     | Déplacer / renommer une branche et le reflog correspondant   |
| <code>-M</code>             | Raccourci pour <code>--move --force</code>   |
| <code>-r, --remotes</code>  | Lister ou supprimer (si utilisé avec <code>-d</code> ) les branches de suivi à distance  |
| <code>-a, --tous</code>     | Liste à la fois les branches de suivi à distance et les succursales locales  |
| <code>--liste</code>        | Activer le mode liste. <code>git branch &lt;pattern&gt;</code> essaierait de créer une branche, utilisez <code>git branch --list &lt;pattern&gt;</code> pour lister les branches correspondantes   |
| <code>--set-upstream</code> | Si la branche spécifiée n'existe pas encore ou si <code>--force</code> a été donné, agit exactement comme <code>--track</code> . Sinon, configure la configuration comme <code>--track</code> lors de la création de la branche, sauf que la branche ne change pas |

### Remarques

Chaque dépôt git a une ou plusieurs *branches*. Une branche est une référence nommée à `HEAD` d'une séquence de commits.

Un git a une branche de *courant* (indiqué par un `*` dans la liste des noms de branche imprimés par la `git branch` commande), chaque fois que vous créez un nouveau commit avec le `git commit` commande, votre nouveau commit devient la `HEAD` de la branche actuelle, et le `HEAD` précédent devient le parent du nouvel commit.

Une nouvelle succursale aura le même `HEAD` que la succursale à partir de laquelle elle a été créée jusqu'à ce que quelque chose soit engagé dans la nouvelle succursale.

## Exemples

### Liste des succursales

Git fournit plusieurs commandes pour lister les branches. Toutes les commandes utilisent la fonction de `git branch`, qui fournira une liste de certaines branches, en fonction des options placées sur la ligne de commande. Si possible, Git indiquera la branche actuellement sélectionnée avec une étoile à côté.

| Objectif  | Commander   |
|---|---|
| Liste des branches locales                                  | <code>git branch</code>                                     |
| Répertorier les branches locales                            | <code>git branch -v</code>                                  |
| Liste des branches distantes et locales                     | <code>git branch -a</code> OR <code>git branch --all</code> |
| Liste des branches distantes et locales (verbose)           | <code>git branch -av</code>                                 |
| Liste des branches distantes                                | <code>git branch -r</code>                                  |
| Liste des succursales distantes avec la dernière validation | <code>git branch -rv</code>                                 |
| Liste des branches fusionnées                               | <code>git branch --merged</code>                            |
| Liste des branches non fusionnées                           | <code>git branch --no-merged</code>                         |
| Liste des branches contenant un commit                      | <code>git branch --contains [&lt;commit&gt;]</code>         |

### Notes :

- Ajouter un `v` supplémentaire à `-v` par exemple `$ git branch -avv` ou `$ git branch -vv` affichera également le nom de la branche en amont.
- Les branches affichées en rouge sont des branches éloignées

### Créer et vérifier de nouvelles branches

Pour créer une nouvelle branche tout en restant sur la branche en cours, utilisez:

```
git branch <name>
```

Généralement, le nom de la succursale ne doit pas contenir d'espaces et est soumis à d'autres spécifications répertoriées [ici](#). Pour basculer vers une branche existante:

```
git checkout <name>
```

Pour créer une nouvelle branche et y basculer:

```
git checkout -b <name>
```

Pour créer une branche à un point autre que le dernier commit de la branche en cours (également



appelée HEAD), utilisez l'une des commandes suivantes:

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

Le <start-point> peut être toute [révision](#) connue de git (par exemple, un autre nom de branche, commit SHA ou une référence symbolique telle que HEAD ou un nom de tag):

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

Pour créer une branche à partir d'une [branche distante](#) (la valeur par défaut <remote\_name> est l'origine):

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

Si un nom de branche donné est trouvé sur une seule télécommande, vous pouvez simplement utiliser

```
git checkout -b <branch_name>
```

ce qui équivaut à

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

Parfois, vous devrez peut-être déplacer plusieurs de vos commits récents vers une nouvelle succursale. Ceci peut être réalisé en ramifiant et en "roulant", comme ceci:

```
git branch <new_name>
git reset --hard HEAD~2 # Go back 2 commits, you will lose uncommitted work.
git checkout <new_name>
```

Voici une explication illustrative de cette technique:

| Initial state    | After git branch <new_name> | After git reset --hard HEAD~2 |
|------------------|-----------------------------|-------------------------------|
|                  | newBranch                   | newBranch                     |
|                  | ↓                           | ↓                             |
| A-B-C-D-E (HEAD) | A-B-C-D-E (HEAD)            | A-B-C-D-E (HEAD)              |
| ↑                | ↑                           | ↑                             |
| master           | master                      | master                        |

### Supprimer une branche localement

```
$ git branch -d dev
```

Supprime la branche nommée dev si ses modifications sont fusionnées avec une autre branche et ne seront pas perdues. Si la branche dev contient des modifications qui n'ont pas encore été fusionnées et qui seraient perdues, git branch -d échouera:

```
$ git branch -d dev
error: The branch 'dev' is not fully merged.
```

```
If you are sure you want to delete it, run 'git branch -D dev'.
```

Par le message d'avertissement, vous pouvez forcer la suppression de la branche (et perdre les modifications non fusionnées dans cette branche) en utilisant l'indicateur -D :

```
$ git branch -D dev
```

### Découvrez une nouvelle branche de suivi d'une succursale distante

Il existe trois façons de créer une nouvelle feature branche qui suit l' origin/feature branche distante:

- `git checkout --track -b feature origin/feature ,`
- `git checkout -t origin/feature ,`
- `git checkout feature` - en supposant qu'il n'y a pas de branche d' feature locale et qu'il n'y a qu'une seule télécommande avec la branche de feature .

Pour définir en amont le suivi de la branche distante - tapez:

- `git branch --set-upstream-to=<remote>/<branch> <branch>`
  - `git branch -u <remote>/<branch> <branch>`

où:

- `<remote>` peut être: `origin` , `develop` ou celui créé par l'utilisateur,
- `<branch>` est la branche de l'utilisateur à suivre sur la télécommande.

Pour vérifier quelles succursales distantes vos branches locales suivent:

- `git branch -vv`

### Renommer une branche

Renommez la branche que vous avez extraite:

```
git branch -m new_branch_name
```

Renommez une autre branche:

```
git branch -m branch_you_want_to_rename new_branch_name
```

### Écraser un seul fichier dans le répertoire de travail en cours avec le même fichier d'une autre branche

Le fichier extrait **écrasera les** modifications que vous n'aviez pas encore effectuées dans ce fichier.

Cette commande extrait le fichier `file.example` (situé dans le répertoire `path/to/` ) et **remplace toutes les modifications** apportées à ce fichier.

```
git checkout some-branch path/to/file
```

*some-branch* peut être n'importe quel *tree-ish* connu (voir [Sélection de révision](#) et [gitrevisions](#) pour plus de détails)

Vous devez ajouter `--` avant le chemin si votre fichier pourrait être confondu avec un fichier (facultatif). Aucune autre option ne peut être fournie après le `--`.

```
git checkout some-branch -- some-file
```

Le second `some-file` est un fichier dans cet exemple.

### Supprimer une branche distante

Pour supprimer une branche sur le référentiel distant d' `origin`, vous pouvez utiliser pour Git version 1.5.0 et ultérieure

```
git push origin :<branchName>
```

et à partir de la version 1.7.0 de Git, vous pouvez supprimer une branche distante en utilisant

```
git push origin --delete <branchName>
```

Pour supprimer une branche de suivi à distance locale:

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # Shorter

git fetch <remote> --prune # Delete multiple obsolete tracking branches
git fetch <remote> -p      # Shorter
```

Pour supprimer une branche localement. Notez que cela ne supprimera pas la branche si elle a des modifications non fusionnées:

```
git branch -d <branchName>
```

Pour supprimer une branche, même si elle comporte des modifications non fusionnées:

```
git branch -D <branchName>
```

### Créer une branche orpheline (c'est-à-dire une branche sans engagement parent)

```
git checkout --orphan new-orphan-branch
```

Le premier commit effectué sur cette nouvelle branche n'aura pas de parents et ce sera la racine d'un nouvel historique totalement déconnecté de toutes les autres branches et validations.

[la source](#)

### Poussez la branche à distance

Utilisez pour pousser les validations effectuées sur votre branche locale vers un référentiel distant.

La commande `git push` prend deux arguments:

- Un nom distant, par exemple, `origin`
- Un nom de branche, par exemple, `master`

Par exemple:

```
git push <REMOTENAME> <BRANCHNAME>
```

Par exemple, vous exécutez généralement `git push origin master` pour transmettre vos modifications locales à votre référentiel en ligne.

Utiliser `-u` (abréviation de `--set-upstream`) configurera les informations de suivi pendant la diffusion.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

Par défaut, git envoie la branche locale à une branche distante du même nom. Par exemple, si vous avez une `new-feature` appelée locale, si vous appuyez sur la branche locale, elle créera également une `new-feature` branche distante. Si vous souhaitez utiliser un autre nom pour la branche à distance, ajoutez le nom distant après le nom de la branche locale, séparés par `:` :

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

### Déplacer la branche actuelle HEAD vers un commit arbitraire

Une branche est juste un pointeur sur un commit, vous pouvez donc le déplacer librement. Pour que la branche fasse référence au commit `aabbcc`, `aabbcc` la commande

```
git reset --hard aabbcc
```

Veillez noter que cela écrasera le commit actuel de votre branche et, par conséquent, tout son historique. Vous pourriez perdre du travail en émettant cette commande. Si tel est le cas, vous pouvez utiliser le [reflog](#) pour récupérer les commits perdus. Il est conseillé d'exécuter cette commande sur une nouvelle branche au lieu de celle en cours.

Cependant, cette commande peut être particulièrement utile lors de la modification ou de la modification de l'historique.

### Passage rapide à la branche précédente

Vous pouvez rapidement passer à la branche précédente en utilisant

```
git checkout -
```

### Recherche dans les branches

Pour répertorier les branches locales contenant une validation ou une balise spécifique

```
git branch --contains <commit>
```

Pour répertorier les branches locales et distantes contenant une validation ou une balise spécifique

```
git branch -a --contains <commit>
```

Lire Ramification en ligne: <https://riptutorial.com/fr/git/topic/415/ramification>

## Chapitre 44: Rangement de votre référentiel local et distant

### Exemples

Supprimer les branches locales qui ont été supprimées sur la télécommande

Pour le suivi à distance entre les branches distantes locales et supprimées, utilisez

```
git fetch -p
```

vous pouvez alors utiliser

```
git branch -vv
```

pour voir quelles branches ne sont plus suivies.

Les branches qui ne sont plus suivies seront dans la forme ci-dessous, contenant «disparu»

```
branch          12345e6 [origin/branch: gone] Fixed bug
```

vous pouvez alors utiliser une combinaison des commandes ci-dessus, en cherchant où "git branch -vv" renvoie "à partir", puis en utilisant "-d" pour supprimer les branches

```
git fetch -p && git branch -vv | awk '/: gone/{print $1}' | xargs git branch -d
```

Lire Rangement de votre référentiel local et distant en ligne:

<https://riptutorial.com/fr/git/topic/10934/rangement-de-votre-referentiel-local-et-distant>

### Syntaxe

- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]`
  - `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]`
  - `git rebase --continue | --skip | --abort | --edit-todo`

### Paramètres

| Paramètre                        | Détails   |
|----------------------------------|---|
| <code>--continuer</code>         | Redémarrez le processus de rebasage après avoir résolu un conflit de fusion.  |
| <code>--avorter</code>           | Abandonnez l'opération de réinitialisation et réinitialisez HEAD dans la branche d'origine. Si la branche a été fournie lorsque l'opération de rebase a été lancée, HEAD sera réinitialisé sur la branche. Sinon, HEAD sera réinitialisé à l'endroit où l'opération de rebase a été lancée.   |
| <code>- maintenir- vide</code>   | Conservez les commits qui ne changent rien de ses parents dans le résultat.   |
| <code>--sauter</code>            | Redémarrez le processus de rebasage en ignorant le correctif actuel.  |
| <code>-m, -- merge</code>        | Utilisez des stratégies de fusion pour rebaser. Lorsque la stratégie de fusion récursive (par défaut) est utilisée, cela permet à rebase de prendre connaissance des noms de domaine en amont. Notez qu'une fusion rebase fonctionne en relisant chaque validation depuis la branche de travail située en haut de la branche en amont. À cause de cela, quand un conflit de fusion se produit, la partie rapportée comme la nôtre est la série rebasée jusqu'ici, en commençant par l'amont, et la leur est la branche de travail. En d'autres termes, les côtés sont échangés. |
| <code>--stat</code>              | Montrer un différent de ce qui a changé en amont depuis le dernier rebase. Le diffstat est également contrôlé par l'option de configuration <code>rebase.stat</code> .  |
| <code>-X, command - -exec</code> | Effectuer un rebase interactif, arrêter entre chaque <code>command</code> validation et d'exécution   |

### Remarques

N'oubliez pas que rebase réécrit efficacement l'historique du référentiel.

Les remises de réaffectation existant dans le référentiel distant pourraient réécrire les nœuds de référentiel utilisés par d'autres développeurs en tant que nœud de base pour leurs développements. À moins que vous ne sachiez vraiment ce que vous faites, il est recommandé de repasser avant de pousser vos modifications.

### Exemples

## Rebranchement de branche locale

**Rebasing** réapplique une série de commits au-dessus d'un autre commit.

Pour rebase une branche, rebase la branche, puis rebase -la sur une autre branche.

```
git checkout topic
git rebase master # rebase current branch onto master branch
```

Cela provoquerait:

```
    A---B---C topic
   /
  D---E---F---G master
```

Se transformer en:

```
    A'--B'--C' topic
   /
  D---E---F---G master
```

Ces opérations peuvent être combinées en une seule commande qui extrait la branche et la rebase immédiatement:

```
git rebase master topic # rebase topic branch onto master branch
```

**Important:** Après le rebase, les validations appliquées auront un hachage différent. Vous ne devez pas rebaser les commits que vous avez déjà envoyés sur un hôte distant. Une conséquence peut être une incapacité à git push votre branche rebasée locale vers un hôte distant, en laissant votre seule option à git push --force .

## Rebase: les nôtres et les leurs, local et distant

Une rebase change le sens de "notre" et "leurs":

```
git checkout topic
git rebase master # rebase topic branch on top of master branch
```

## Quelle que soit la direction de HEAD, c'est "la nôtre"

La première chose que fait un rebase est de réinitialiser le HEAD pour le master ; avant picorage engage de l'ancienne branche topic à un nouveau (tous les commits dans l'ancien topic branche sera réécrite et seront identifiés par un hachage différent).

En ce qui concerne les terminologies utilisées par les outils de fusion (à ne pas confondre avec [ref local](#) ou [ref distant](#) )

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

Cela signifie qu'un outil de fusion / diff présentera la branche en amont comme local ( master : la branche au-dessus de laquelle vous rebasez), et la branche de travail comme remote ( topic : la branche en cours de rebasage)

```
+-----+
```

```
| LOCAL:master |     BASE     | REMOTE:topic |
+-----+
|               MERGED              |
+-----+
```

---

## Inversion illustrée

### Sur une fusion:

```
c--c--x--x--x(*) <- current branch topic ('*' = HEAD)
 \
  \
   \--y--y--y <- other branch to merge
```

Nous ne changeons pas le topic branche actuelle, donc ce que nous avons est toujours ce sur quoi nous travaillions (et nous fusionnons à partir d'une autre branche)

```
c--c--x--x--x-----o(*)  MERGE, still on branch topic
 \      ^
  \     ours
   \   /
    \ /
     \--y--y--y--/
      ^
     theirs
```

---

### Sur un rebase:

Mais **sur une rebase**, nous changeons de côté car la première chose que fait une rebase est de récupérer la branche en amont pour rejouer les commits en cours!

```
c--c--x--x--x(*) <- current branch topic ('*' = HEAD)
 \
  \
   \--y--y--y <- upstream branch
```

Un **git rebase upstream** abord HEAD sur la branche en amont, d'où le changement de «ours» et de «leurs» par rapport à la branche de travail «actuelle» précédente.

```
c--c--x--x--x <- former "current" branch, new "theirs"
 \
  \
   \--y--y--y(*) <- set HEAD to this commit, to replay x's on it
                    ^
                    |
                    this will be the new "ours"
                    |
                    upstream
```

Le rebasage sera ensuite rejouer « leur » engage sur le nouveau « notre » topic branche:

```
c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"
 \
  \
   \--y--y--y--x'--x'--x(*) <- topic once all x's are replayed,
                    ^
                    point branch topic to this commit
```



```
|  
upstream branch
```

## Rebase interactif

Cet exemple a pour but de décrire comment on peut utiliser le git rebase en mode interactif. On s'attend à ce que l'on ait une compréhension de base de ce qu'est le git rebase et de ce qu'il fait.

Le rebase interactif est lancé à l'aide de la commande suivante:

```
git rebase -i
```

L'option `-i` fait référence au *mode interactif*. En utilisant le rebase interactif, l'utilisateur peut modifier les messages de validation, ainsi que réorganiser, diviser et / ou squash (combinaison en un seul) commits.

Disons que vous souhaitez réorganiser vos trois derniers commits. Pour ce faire, vous pouvez exécuter:

```
git rebase -i HEAD~3
```

Après avoir exécuté les instructions ci-dessus, un fichier sera ouvert dans votre éditeur de texte où vous pourrez sélectionner la manière dont vos commits seront rebasés. Pour les besoins de cet exemple, changez simplement l'ordre de vos commits, enregistrez le fichier et fermez l'éditeur. Cela déclenchera un rebase avec l'ordre que vous avez appliqué. Si vous cochez git log vous verrez vos commits dans la nouvelle commande que vous avez spécifiée.

---

### Reformulation des messages de validation

Maintenant, vous avez décidé que l'un des messages de validation est vague et que vous souhaitez qu'il soit plus descriptif. Examinons les trois derniers commits en utilisant la même commande.

```
git rebase -i HEAD~3
```

Au lieu de réorganiser la commande, les commits seront rebasés, cette fois nous changerons le pick, la valeur par défaut, pour reword un commit où vous voudriez changer le message.

Lorsque vous fermez l'éditeur, le rebase se lance et s'arrête au message de validation spécifique que vous souhaitez reformuler. Cela vous permettra de changer le message de validation selon votre souhait. Après avoir modifié le message, fermez simplement l'éditeur pour continuer.

---

### Changer le contenu d'un commit

Outre la modification du message de validation, vous pouvez également adapter les modifications apportées par le commit. Pour ce faire, changez simplement pick pour edit pour un commit. Git s'arrêtera lorsqu'il arrivera à cette validation et fournira les modifications d'origine de la validation dans la zone de mise en attente. Vous pouvez maintenant adapter ces modifications en les désinstallant ou en ajoutant de nouvelles modifications.

Dès que la zone de transfert contient toutes les modifications souhaitées, validez les modifications. L'ancien message de validation sera affiché et pourra être adapté pour refléter le nouveau commit.

---

### Fractionnement d'un seul engagement en plusieurs

Supposons que vous ayez fait un commit mais que, plus tard, vous avez décidé de diviser ce commit en deux ou plusieurs commits. En utilisant la même commande qu'auparavant, remplacez plutôt pick par edit et appuyez sur enter.

Maintenant, git s'arrêtera à la validation que vous avez marquée pour l'édition et place tout son contenu dans la zone de transfert. A partir de là, vous pouvez lancer git reset HEAD^ pour placer le commit dans votre répertoire de travail. Ensuite, vous pouvez ajouter et valider vos fichiers dans un ordre différent - en fin de compte, diviser un seul commit en  $n$  commits.

### Écraser plusieurs commets en un

Supposons que vous ayez fait du travail et que vous ayez plusieurs commits qui, selon vous, pourraient être un simple commit. Pour cela, vous pouvez effectuer git rebase -i HEAD~3 , en remplaçant 3 par une quantité appropriée de commits.

Cette fois, remplacez pick par squash place. Au cours du rebase, la validation à laquelle vous avez demandé d'être écrasé sera écrasée par-dessus la validation précédente; les transformer en un seul engagement à la place.

### Abandon d'une base de données interactive

Vous avez démarré un rebase interactif. Dans l'éditeur où vous choisissez vos commits, vous décidez que quelque chose ne va pas (par exemple, une validation est manquante ou vous avez choisi la mauvaise destination de rebase) et vous souhaitez abandonner le rebase.

Pour ce faire, supprimez simplement tous les commits et actions (c.-à-d. Toutes les lignes ne commençant pas par le signe # ) et la rebase sera annulée!

Le texte d'aide dans l'éditeur fournit effectivement cette astuce:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

### Pousser après un rebase

Parfois, vous avez besoin de réécrire l'histoire avec une rebase, mais git push plaint de le faire parce que vous avez réécrit l'histoire.

Cela peut être résolu avec un git push --force , mais considérez git push --force-with-lease , indiquant que vous voulez que le push échoue si la branche de suivi à distance locale diffère de la branche sur la télécommande, par exemple, quelqu'un sinon poussé à la télécommande après la dernière extraction. Cela évite d'écraser par inadvertance la poussée récente de quelqu'un d'autre.

**Remarque :** git push --force - et même --force-with-lease pour cette question - peut être une commande dangereuse car elle réécrit l'historique de la branche. Si une autre personne avait

tiré la branche avant la poussée forcée, son / sa git pull ou git fetch aurait des erreurs parce que l'histoire locale et l'histoire distante sont divergentes. Cela peut entraîner des erreurs inattendues. Lorsque l'on regarde suffisamment les reflocs, le travail de l'autre utilisateur peut être récupéré, mais cela peut entraîner beaucoup de temps perdu. Si vous devez effectuer une poussée forcée vers une branche avec d'autres contributeurs, essayez de vous coordonner avec eux afin qu'ils n'aient pas à gérer les erreurs.

## Rebase à la validation initiale

Depuis Git 1.7.12, il est possible de rebaser à la validation racine. La validation racine est la première validation jamais effectuée dans un référentiel et ne peut normalement pas être modifiée. Utilisez la commande suivante:

```
git rebase -i --root
```

## Rebasing avant une revue de code

### Résumé

Cet objectif est de réorganiser tous vos commits dispersés en des commits plus significatifs pour des revues de code plus faciles. S'il y a trop de couches de changements sur trop de fichiers à la fois, il est plus difficile de faire une révision du code. Si vous pouvez réorganiser vos commits créés chronologiquement en commits d'actualité, le processus de révision du code est plus facile (et peut-être moins de bogues au cours du processus de révision du code).

Cet exemple trop simplifié n'est pas la seule stratégie pour utiliser git pour faire de meilleures révisions de code. C'est comme ça que je le fais, et c'est quelque chose qui inspire les autres à réfléchir à la manière de rendre les révisions de code et l'histoire de Git plus faciles / meilleures.

Cela démontre également pédagogiquement le pouvoir de rebase en général.

Cet exemple suppose que vous connaissez le rebasage interactif.

---

### En supposant:

- vous travaillez sur une branche de fonctionnalités de maître
- votre fonctionnalité comporte trois couches principales: front-end, back-end, DB
- Vous avez fait beaucoup de commits en travaillant sur une branche de fonctionnalités. Chaque validation touche plusieurs couches à la fois
- vous voulez (au final) que trois commits dans votre branche
  - un contenant tous les changements frontaux
  - un contenant tous les changements de back-end
  - un contenant tous les changements de base de données

---

### Stratégie:

- nous allons changer nos engagements chronologiques en engagements "topiques".
- Tout d'abord, divisez tous les commits en plusieurs commits plus petits - chacun ne contenant qu'un seul sujet à la fois (dans notre exemple, les sujets sont les modifications front-end, back-end, DB)
- Réorganisez ensuite nos commits topiques et «écrasez-les» en des engagements uniques

---

### Exemple:

```
$ git log --oneline master..  
975430b db adding works: db.sql logic.rb  
3702650 trying to allow adding todo items: page.html logic.rb  
43b075a first draft: page.html and db.sql  
$ git rebase -i master
```

Cela sera affiché dans l'éditeur de texte:

```
pick 43b075a first draft: page.html and db.sql  
pick 3702650 trying to allow adding todo items: page.html logic.rb  
pick 975430b db adding works: db.sql logic.rb
```

Changez la en ceci:

```
e 43b075a first draft: page.html and db.sql  
e 3702650 trying to allow adding todo items: page.html logic.rb  
e 975430b db adding works: db.sql logic.rb
```

Ensuite, git appliquera un engagement à la fois. Après chaque validation, il affichera une invite, puis vous pourrez effectuer les opérations suivantes:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... first draft: page.html and db.sql  
You can amend the commit now, with
```

```
git commit --amend
```

Once you are satisfied with your changes, run

```
git rebase --continue
```

```
$ git status  
rebase in progress; onto 4975ae9  
You are currently editing a commit while rebasing branch 'feature' on '4975ae9'.  
  (use "git commit --amend" to amend the current commit)  
  (use "git rebase --continue" once you are satisfied with your changes)
```

```
nothing to commit, working directory clean  
$ git reset HEAD^ #This 'uncommits' all the changes in this commit.  
$ git status -s  
M db.sql  
M page.html  
$ git add db.sql #now we will create the smaller topical commits  
$ git commit -m "first draft: db.sql"  
$ git add page.html  
$ git commit -m "first draft: page.html"  
$ git rebase --continue
```

Ensuite, vous répéterez ces étapes pour chaque validation. Au final, vous avez ceci:

```
$ git log --oneline  
0309336 db adding works: logic.rb  
06f81c9 db adding works: db.sql  
3264de2 adding todo items: page.html  
675a02b adding todo items: logic.rb  
272c674 first draft: page.html  
08c275d first draft: db.sql
```

Maintenant, nous exécutons une nouvelle fois rebase pour réorganiser et écraser:

```
$ git rebase -i master
```

Cela sera affiché dans l'éditeur de texte:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Changez la en ceci:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s 3264de2 adding todo items: page.html
```

AVIS: assurez-vous que vous indiquez à git rebase d'appliquer / écraser les plus petits commits d'actualité *dans l'ordre dans lequel ils ont été engagés chronologiquement* . Sinon, vous pourriez avoir de faux conflits de fusion inutiles.

Lorsque tout cela est dit et fait, vous obtenez ceci:

```
$ git log --oneline master..
74bdd5f adding todos: GUI layer
e8d8f7e adding todos: business logic layer
121c578 adding todos: DB layer
```

---

## résumer

Vous avez maintenant rebaptisé vos commits chronologiques en commits d'actualité. Dans la vraie vie, vous n'avez peut-être pas besoin de le faire à chaque fois, mais quand vous voulez ou devez le faire, vous pouvez le faire maintenant. De plus, j'espère que vous en avez appris plus sur le rebat de Git.

### Configuration de git-pull pour effectuer automatiquement une rebase au lieu d'une fusion

Si votre équipe suit un flux de travail basé sur une base de données, il peut être avantageux de configurer git pour que chaque branche nouvellement créée effectue une opération de rebase, au lieu d'une opération de fusion, pendant une git pull .

Pour configurer chaque *nouvelle* branche afin qu'elle se réinitialise automatiquement, ajoutez ce qui suit à votre .gitconfig ou .git/config :

```
[branch]
autosetuprebase = always
```

Ligne de commande: `git config [--global] branch.autosetuprebase always`

Vous pouvez également configurer la commande git pull pour toujours vous comporter comme si l'option --rebase était transmise:

```
[pull]
rebase = true
```

Ligne de commande: `git config [--global] pull.rebase true`

### Tester tous les commits pendant le rebase

Avant de faire une demande d'extraction, il est utile de s'assurer que la compilation est réussie et que les tests réussissent pour chaque validation dans la branche. Nous pouvons le faire automatiquement en utilisant le paramètre `-x`.

Par exemple:

```
git rebase -i -x make
```

effectuera le rebase interactif et s'arrêtera après chaque engagement à exécuter `make`. Dans le cas où `make` échoue, `git` cessera de vous donner l'occasion de corriger les problèmes et modifier le commit avant de procéder à la cueillette de la suivante.

### Configuration de l'autostash

Autostash est une option de configuration très utile lorsque vous utilisez `rebase` pour des modifications locales. Souvent, vous devrez peut-être importer des commits de la branche en amont, mais vous n'êtes pas encore prêt à vous engager.

Cependant, `Git` ne permet pas à une `rebase` de démarrer si le répertoire de travail n'est pas propre. `Autostash` à la rescousse:

```
git config --global rebase.autostash # one time configuration
git rebase @{u} # example rebase on upstream branch
```

L'autostash sera appliqué chaque fois que le `rebase` est terminé. Peu importe que la réinitialisation se termine avec succès ou qu'elle soit interrompue. De toute façon, l'autostash sera appliqué. Si le `rebase` a réussi et que la validation de base a donc changé, il peut y avoir un conflit entre l'autostash et les nouveaux commits. Dans ce cas, vous devrez résoudre les conflits avant de vous engager. Ce n'est pas différent que si vous aviez manuellement masqué, puis appliqué, donc il n'y a aucun inconvénient à le faire automatiquement.

Lire **Rebasing** en ligne: <https://riptutorial.com/fr/git/topic/355/rebasing>

## Chapitre 46: Récupérer

### Exemples

#### Récupérer d'un engagement perdu

Au cas où vous seriez revenu à un engagement passé et perdu un nouvel engagement, vous pouvez récupérer le commit perdu en exécutant

```
git reflog
```

Ensuite, trouvez votre commit perdu et recommencez en faisant

```
git reset HEAD --hard <shal-of-commit>
```

#### Restaurer un fichier supprimé après une validation

Dans le cas où vous avez accidentellement commis une suppression sur un fichier et réalisé par la suite que vous en avez besoin.

Recherchez d'abord l'ID de validation du commit qui a supprimé votre fichier.

```
git log --diff-filter=D --summary
```

Vous donnera un résumé trié des commits qui ont supprimé les fichiers.

Ensuite, procédez à la restauration du fichier par

```
git checkout 81eccc~1 <your-lost-file-name>
```

(Remplacez 81eccc par votre propre identifiant de validation)

#### Restaurer le fichier vers une version précédente

Pour restaurer un fichier à une version précédente, vous pouvez utiliser la reset .

```
git reset <shal-of-commit> <file-name>
```

Si vous avez déjà apporté des modifications locales au fichier (que vous n'avez pas besoin!), Vous pouvez également utiliser l'option --hard

#### Récupérer une branche supprimée

Pour récupérer une branche supprimée, vous devez rechercher le commit qui était la tête de votre branche supprimée en exécutant

```
git reflog
```

Vous pouvez ensuite recréer la branche en cours d'exécution

```
git checkout -b <branch-name> <shal-of-commit>
```

Vous ne serez pas en mesure de récupérer les branches supprimées si le commit effacé par le

`ramasse - miettes` de git - sans réf. Ayez toujours une sauvegarde de votre référentiel, en particulier lorsque vous travaillez dans une petite équipe / un projet propriétaire

## Récupération d'une réinitialisation

### Avec Git, vous pouvez (presque) toujours revenir en arrière

N'ayez pas peur d'expérimenter des commandes qui réécrivent l'historique \*. Git ne supprime pas vos commits pendant 90 jours par défaut et pendant ce temps, vous pouvez facilement les récupérer depuis le renvoi:

```
$ git reset @~3 # go back 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: moving to @~3
2c52489 HEAD@{1}: commit: more changes
4a5246d HEAD@{2}: commit: make important changes
e8571e4 HEAD@{3}: commit: make some changes
... earlier commits ...
$ git reset 2c52489
... and you're back where you started
```

\* Surveillez les options comme `--hard` et `--force` si - elles peuvent rejeter des données.

\* Evitez également de réécrire l'historique sur les branches sur lesquelles vous collaborez.

## Récupérer de Git Stash

Pour obtenir votre plus récente réserve après avoir exécuté `git stash`, utilisez

```
git stash apply
```

Pour voir une liste de vos caches, utilisez

```
git stash list
```

Vous obtiendrez une liste qui ressemble à quelque chose comme ça

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Choisissez un autre git stash à restaurer avec le numéro qui apparaît pour la réserve que vous voulez

```
git stash apply stash@{2}
```

Vous pouvez également choisir «`git stash pop`», cela fonctionne de la même manière que «`git stash apply`», par exemple.

```
git stash pop
```

ou

```
git stash pop stash@{2}
```

Différence dans Git Stash s'applique et `git stash pop` ...



**Git Stash Pop** : - Les données de stash seront supprimées de la pile de la liste de stash.

Ex:-

```
git stash list
```

Vous obtiendrez une liste qui ressemble à quelque chose comme ça

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop  
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Maintenant, pop des données de stockage en utilisant la commande

```
git stash pop
```

Vérifiez à nouveau la liste de réserve

```
git stash list
```

Vous obtiendrez une liste qui ressemble à quelque chose comme ça

```
stash@{0}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Vous pouvez voir qu'une des données de la mémoire cache est supprimée de la liste de dissimulation et que la réserve @ {1} est devenue stash @ {0}.

Lire Récupérer en ligne: <https://riptutorial.com/fr/git/topic/725/recuperer>

### Syntaxe

- `clone git [<options>] [-] <repo> [<dir>]`
- `git clone [--template = <répertoire_de_modèles>] [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror] [-o <nom> ] [-b <nom>] [-u <paquet de téléchargement>] [--référence <repository>] [--dissociate] [--separate-git-dir <dir git>] [--depth <profondeur> ] [- [no-] single-branch] [--recursive | --recurse-submodules] [- [no-] shallow-submodules] [--jobs <n>] [-] <repository> [<répertoire>]`

### Exemples

#### Clone peu profond

Cloner un énorme référentiel (comme un projet avec plusieurs années d'historique) peut prendre beaucoup de temps ou échouer en raison de la quantité de données à transférer. Dans les cas où vous n'avez pas besoin d'avoir l'historique complet disponible, vous pouvez faire un clone superficiel:

```
git clone [repo_url] --depth 1
```

La commande ci-dessus va récupérer uniquement le dernier commit du référentiel distant.

Sachez que vous ne pourrez peut-être pas résoudre les fusions dans un référentiel superficiel. C'est souvent une bonne idée de prendre au moins autant de commits que vous allez avoir besoin de revenir en arrière pour résoudre les fusions. Par exemple, pour obtenir les 50 derniers commits:

```
git clone [repo_url] --depth 50
```

Plus tard, si nécessaire, vous pouvez récupérer le reste du dépôt:

#### 1.8.3

```
git fetch --unshallow      # equivalent of git fetch --depth=2147483647
                           # fetches the rest of the repository
```

#### 1.8.3

```
git fetch --depth=1000    # fetch the last 1000 commits
```

#### Clone Régulier

Pour télécharger l'intégralité du référentiel, y compris l'historique complet et toutes les branches, tapez:

```
git clone <url>
```

L'exemple ci-dessus le placera dans un répertoire portant le même nom que le nom du référentiel.

Pour télécharger le référentiel et l'enregistrer dans un répertoire spécifique, tapez:

```
git clone <url> [directory]
```

Pour plus de détails, visitez [Cloner un référentiel](#) .

## Cloner une branche spécifique

Pour cloner une branche spécifique d'un référentiel, tapez `--branch <branch name>` avant l'URL du référentiel :

```
git clone --branch <branch name> <url> [directory]
```

Pour utiliser l'option abrégée pour `--branch` , tapez `-b` . Cette commande télécharge le référentiel entier et extrait `<branch name>` .

Pour économiser de l'espace disque, vous pouvez cloner l'historique en ne menant qu'à une seule branche avec :

```
git clone --branch <branch_name> --single-branch <url> [directory]
```

Si `--single-branch` n'est pas ajouté à la commande, l'historique de toutes les branches sera cloné dans `[directory]` . Cela peut être un problème avec les grands dépôts.

---

Pour annuler ultérieurement l' `--single-branch` flag et récupérer le reste de la commande use `repository` :

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"  
git fetch origin
```

## Cloner récursivement

1.6.5

```
git clone <url> --recursive
```

Clone le référentiel et clone tous les sous-modules. Si les sous-modules eux-mêmes contiennent des sous-modules supplémentaires, Git les clonera également.

## Cloner en utilisant un proxy

Si vous devez télécharger des fichiers avec git sous un proxy, la configuration du serveur proxy à l'échelle du système ne pourrait pas suffire. Vous pouvez également essayer ce qui suit :

```
git config --global http.proxy http://<proxy-server>:<port>/
```

Lire Référentiels de clonage en ligne: <https://riptutorial.com/fr/git/topic/1405/referentiels-de-clonage>

## Chapitre 48: Reflog - Restauration des commits non affichés dans le journal git

### Remarques

Le reflog de Git enregistre la position de HEAD (la référence de l'état actuel du référentiel) chaque fois qu'il est modifié. Généralement, chaque opération qui pourrait être destructive implique de déplacer le pointeur HEAD (car si quelque chose est modifié, y compris dans le passé, le hachage de la commande tip change), il est toujours possible de revenir à un état antérieur à une opération dangereuse, en trouvant la bonne ligne dans le reflog.

Les objets qui ne sont référencés par aucune référence sont généralement récupérés en 30 jours environ, de sorte que le renvoi peut ne pas toujours être utile.

### Exemples

#### Se remettre d'une mauvaise rebase

Supposons que vous ayez démarré un rebase interactif:

```
git rebase --interactive HEAD~20
```

et par erreur, vous avez écrasé ou laissé tomber des commits que vous ne vouliez pas perdre, mais vous avez ensuite terminé le rebase. Pour récupérer, faites `git reflog`, et vous pourriez voir des résultats comme ceci:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error
...
ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...
...
```

Dans ce cas, le dernier commit, ddddddd (ou HEAD@{n+1}) est la pointe de votre branche de *pré-rebase*. Ainsi, pour récupérer ce commit (et tous les commits parents, y compris ceux écrasés ou supprimés accidentellement), faites:

```
$ git checkout HEAD@{n+1}
```

Vous pouvez ensuite créer une nouvelle branche à cette validation avec `git checkout -b [branch]`. Voir [Branchement](#) pour plus d'informations.

Lire [Reflog - Restauration des commits non affichés dans le journal git en ligne](https://riptutorial.com/fr/git/topic/5149/reflog---restauration-des-commits-non-affiches-dans-le-journal-git):  
<https://riptutorial.com/fr/git/topic/5149/reflog---restauration-des-commits-non-affiches-dans-le-journal-git>

## Chapitre 49: Renommer

### Syntaxe

- `git mv <source> <destination>`
  - `git mv -f <source> <destination>`

### Paramètres

| Paramètre                               | Détails  |
|---|--|
| <code>-f</code> ou <code>--force</code> | Forcer le renommage ou le déplacement d'un fichier même si la cible existe |

### Exemples

#### Renommer les dossiers

Pour renommer un dossier de `oldName` à `newName`

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

Suivi de `git commit` et `/` ou `git push`

Si cette erreur se produit:

```
fatal: le renommage de 'directoryToFolder / oldName' a échoué: argument non valide
```

Utilisez la commande suivante:

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

#### Renommer une succursale locale

Vous pouvez renommer une branche dans le référentiel local en utilisant cette commande:

```
git branch -m old_name new_name
```

#### renommer une branche locale et distante

le plus simple est de faire vérifier la succursale locale:

```
git checkout old_branch
```

renommez ensuite la branche locale, supprimez l'ancienne télécommande et définissez la nouvelle branche renommée en amont:

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

Lire Renommer en ligne: <https://riptutorial.com/fr/git/topic/1814/renommer>

## Chapitre 50: Répertoires vides dans Git

### Exemples

#### Git ne suit pas les répertoires

Supposons que vous avez initialisé un projet avec la structure de répertoires suivante:

```
/build
app.js
```

Ensuite, vous ajoutez tout ce que vous avez créé jusqu'à présent et vous vous engagez:

```
git init
git add .
git commit -m "Initial commit"
```

Git ne fera que suivre le fichier app.js.

Supposons que vous ayez ajouté une étape de construction à votre application et que vous utilisiez le répertoire "build" comme répertoire de sortie (et vous ne voulez pas en faire une instruction de configuration que chaque développeur doit suivre), une *convention* doit inclure ".gitkeep" fichier dans le répertoire et laissez Git suivre ce fichier.

```
/build
.gitkeep
app.js
```

Puis ajoutez ce nouveau fichier:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Git va maintenant suivre le fichier build / .gitkeep et par conséquent le dossier de compilation sera disponible à la caisse.

Encore une fois, il ne s'agit que d'une convention et non d'une fonctionnalité Git.

Lire Répertoires vides dans Git en ligne: <https://riptutorial.com/fr/git/topic/2680/repertoires-vides-dans-git>

## Chapitre 51: Résoudre les conflits de fusion

### Exemples

#### Résolution manuelle

git merge vous effectuez une git merge vous pouvez constater que git signale une erreur de "conflit de fusion". Il vous indiquera quels fichiers sont en conflit et vous devrez résoudre les conflits.

Un git status à tout moment vous aidera à voir ce qui doit encore être modifié avec un message utile comme

```
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")
```

Git laisse des marqueurs dans les fichiers pour vous dire où le conflit est survenu:

```
<<<<<<<< HEAD: index.html #indicates the state of your current branch
<div id="footer">contact : email@somedomain.com</div>
===== #indicates break between conflicts
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)
```

Pour résoudre les conflits, vous devez modifier la zone entre les marqueurs <<<<<< et >>>>>> de manière appropriée, supprimez les lignes de statut (les <<<<<<, >>>>> >>, et ===== lignes) complètement. Puis git add index.html pour le marquer comme résolu et git commit pour terminer la fusion.

Lire Résoudre les conflits de fusion en ligne:

<https://riptutorial.com/fr/git/topic/3233/resoudre-les-conflits-de-fusion>

### Introduction

S'engager avec Git fournit la responsabilité en attribuant aux auteurs des modifications au code. Git offre de multiples fonctionnalités pour la spécificité et la sécurité des commits. Cette rubrique explique et démontre les bonnes pratiques et procédures lors de la validation avec Git.

### Syntaxe

- `git commit [drapeaux]`

### Paramètres

| Paramètre  | Détails   |
|--|---|
| <code>--message, -m</code>   | Message à inclure dans le commit. La spécification de ce paramètre contourne le comportement normal de Git d'ouvrir un éditeur.   |
| <code>--modifier</code>  | Indiquez que les modifications en cours doivent être ajoutées (modifiées) à la validation <i>précédente</i> . Attention, cela peut réécrire l'histoire!                                 |
| <code>--Pas de modification</code>                                 | Utilisez le message de validation sélectionné sans lancer un éditeur. Par exemple, <code>git commit --amend --no-edit</code> modifie un commit sans modifier son message de validation. |
| <code>--all, -a</code>   | Validez toutes les modifications, y compris celles qui ne sont pas encore mises en scène.   |
| <code>--rendez-vous amoureux</code>                                | Définissez manuellement la date qui sera associée à la validation.  |
| <code>--seulement</code>   | Ne validez que les chemins spécifiés. Cela ne commettra pas ce que vous avez actuellement organisé sauf si vous y êtes invité.  |
| <code>--patch, -p</code>   | Utilisez l'interface de sélection de correctifs interactive pour choisir les modifications à valider.   |
| <code>--Aidez-moi</code>   | Affiche la page de manuel pour <code>git commit</code>  |
| <code>-S [keyid], -S --gpg-sign [= keyid], -S --no-gpg-sign</code> | Sign commit, validation du signe GPG, countermand <code>commit.gpgSign</code> variable de configuration   |
| <code>-n, --no-verify</code>                                       | Cette option contourne les hooks pre-commit et commit-msg. Voir aussi les <a href="#">crochets</a>  |

### Exemples



## S'engager sans ouvrir un éditeur

Git ouvre généralement un éditeur (comme vim ou emacs ) lorsque vous lancez git commit . Passez l'option -m pour spécifier un message à partir de la ligne de commande:

```
git commit -m "Commit message here"
```

Votre message de validation peut dépasser plusieurs lignes:

```
git commit -m "Commit 'subject line' message here  
More detailed description follows here (after a blank line)."
```

Vous pouvez également transmettre plusieurs arguments -m :

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

Voir [Comment écrire un message Git Commit](#) .

[Guide de style de message Udacity Git Commit](#)

## Modifier un engagement

Si votre **dernière validation n'est pas encore publiée** (pas envoyée dans un référentiel en amont), vous pouvez modifier votre commit.

```
git commit --amend
```

Cela mettra les modifications en cours sur le commit précédent.

**Remarque:** Ceci peut également être utilisé pour modifier un message de validation incorrect. Il affichera l'éditeur par défaut (généralement vi / vim / emacs ) et vous permettra de modifier le message précédent.

Pour spécifier le message de validation en ligne:

```
git commit --amend -m "New commit message"
```

Ou utiliser le message de validation précédent sans le modifier:

```
git commit --amend --no-edit
```

La modification met à jour la date de validation mais laisse la date de l'auteur intacte. Vous pouvez dire à git de rafraîchir les informations.

```
git commit --amend --reset-author
```

Vous pouvez également changer l'auteur de la validation avec:

```
git commit --amend --author "New Author <email@address.com>"
```

**Remarque:** sachez que la modification de la validation la plus récente la remplace entièrement et que la validation précédente est supprimée de l'historique de la branche. Cela doit être pris en compte lorsque vous travaillez avec des référentiels publics et sur des succursales avec

d'autres collaborateurs.

Cela signifie que si la validation précédente avait déjà été poussée, après modification, vous devrez `push --force` .

### Commettre des changements directement

Généralement, vous devez utiliser `git add` ou `git rm` pour ajouter des modifications à l'index avant de pouvoir les `git commit` . Passez l'option `-a` ou `--all` pour ajouter automatiquement toutes les modifications (aux fichiers suivis) à l'index, y compris les suppressions:

```
git commit -a
```

Si vous souhaitez également ajouter un message de validation, procédez comme suit:

```
git commit -a -m "your commit message goes here"
```

Vous pouvez également joindre deux drapeaux:

```
git commit -am "your commit message goes here"
```

Vous n'avez pas nécessairement besoin de valider tous les fichiers à la fois. Ignorez l'indicateur `-a` ou `--all` et spécifiez le fichier que vous souhaitez valider directement:

```
git commit path/to/a/file -m "your commit message goes here"
```

Pour valider directement plusieurs fichiers spécifiques, vous pouvez également spécifier un ou plusieurs fichiers, répertoires et modèles:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

### Créer un commit vide

En règle générale, les commits vides (ou validés avec un état identique au parent) constituent une erreur.

Cependant, lorsque vous testez des crochets de construction, des systèmes CI et d'autres systèmes qui déclenchent une validation, il est pratique de pouvoir créer facilement des validations sans avoir à modifier / toucher un fichier factice.

La `--allow-empty` contournera la vérification.

```
git commit -m "This is a blank commit" --allow-empty
```

### Stage et validation des modifications

#### Les bases

Après avoir modifié votre code source, vous devez **mettre en place** ces modifications avec Git avant de pouvoir les valider.

Par exemple, si vous modifiez `README.md` et `program.py` :

```
git add README.md program.py
```

Cela indique à git que vous voulez ajouter les fichiers à la prochaine validation que vous faites.

Ensuite, validez vos modifications avec

```
git commit
```

Notez que cela ouvrira un éditeur de texte, [souvent vim](#) . Si vous n'êtes pas familier avec vim, vous pouvez savoir que vous pouvez appuyer sur `i` pour passer en mode *insertion* , écrire votre message de validation, puis appuyer sur `Esc` et sur `:wq` pour enregistrer et quitter. Pour éviter d'ouvrir l'éditeur de texte, incluez simplement le drapeau `-m` avec votre message

```
git commit -m "Commit message here"
```

Les messages de [validation](#) suivent souvent des règles de formatage spécifiques, voir [Bons messages de validation](#) pour plus d'informations.

---

### Raccourcis

Si vous avez changé beaucoup de fichiers dans le répertoire, plutôt que de les énumérer, vous pouvez utiliser:

```
git add --all          # equivalent to "git add -a"
```

Ou pour ajouter toutes les modifications, à l' *exception des fichiers supprimés* , du répertoire et des sous-répertoires de niveau supérieur:

```
git add .
```

Ou pour ajouter uniquement des fichiers actuellement suivis ("update"):

```
git add -u
```

Si vous le souhaitez, passez en revue les modifications par étapes:

```
git status            # display a list of changed files
git diff --cached     # shows staged changes inside staged files
```

Enfin, validez les modifications:

```
git commit -m "Commit message here"
```

Alternativement, si vous avez seulement modifié des fichiers existants ou des fichiers supprimés, et que vous n'en avez pas créé de nouveaux, vous pouvez combiner les actions de git add et git commit dans une seule commande:

```
git commit -am "Commit message here"
```

Notez que ceci mettra en scène **tous** les fichiers modifiés de la même manière que git add --all .

---

### Données sensibles

Vous ne devez jamais commettre de données sensibles, telles que des mots de passe ou même des clés privées. Si ce cas se produit et que les modifications sont déjà transmises à un serveur central, considérez les données sensibles comme compromises. Sinon, il est possible de supprimer ces données par la suite. Une solution simple et rapide est l'utilisation du "BFG Repo-Cleaner": <https://rtyley.github.io/bfg-repo-cleaner/> .

La commande `bfgr --replace-text passwords.txt my-repo.git` lit les mots de `passwords.txt` fichier `passwords.txt` et les remplace par `***REMOVED***` . Cette opération prend en compte tous les commits précédents du référentiel entier.

### S'engager au nom de quelqu'un d'autre

Si quelqu'un a écrit le code que vous avez `--author` , vous pouvez lui donner le crédit avec l'option `--author` :

```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

Vous pouvez également fournir un modèle que Git utilisera pour rechercher les auteurs précédents:

```
git commit -m "msg" --author "John"
```

Dans ce cas, les informations sur l'auteur de la dernière validation avec un auteur contenant "John" seront utilisées.

Sur GitHub, les commits effectués de l'une des manières ci-dessus afficheront la vignette d'un grand auteur, avec le committer plus petit et devant:

#### Commits on Apr 17, 2015



**Further improvements to soundness proof for addition.**

RenuAB committed with gebn on 23 Feb 2015

### Commencer des modifications dans des fichiers spécifiques

Vous pouvez valider les modifications apportées à des fichiers spécifiques et les ignorer à l'aide de `git add` :

```
git commit file1.c file2.h
```

Ou vous pouvez d'abord mettre en scène les fichiers:

```
git add file1.c file2.h
```

et les commettre plus tard:

```
git commit
```

### Bon commettre des messages

Il est important pour une personne parcourant le `git log` de Git de comprendre facilement ce qu'est chaque validation. Les messages de validation corrects incluent généralement un numéro de tâche ou un problème dans un outil de suivi et une description concise de ce qui a été fait et pourquoi, et parfois aussi de la manière dont cela a été fait.

De meilleurs messages peuvent ressembler à :

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

Considérant que les messages suivants ne seraient pas aussi utiles :

```
fix                                // What has been fixed?
just a bit of a change            // What has changed?
TASK-371                           // No description at all, reader will need to look at the tracker
themselves for an explanation
Implemented IFoo in IBar          // Why it was needed?
```

Une façon de tester si un message de validation est écrit dans le bon état d'esprit est de remplacer le blanc par le message et de voir si cela a du sens :

**Si j'ajoute ce commit, je vais \_\_\_ dans mon référentiel.**

### Les sept règles d'un grand message de validation de git

1. Séparer la ligne d'objet du corps avec une ligne vide
2. Limite la ligne d'objet à 50 caractères
3. Capitaliser la ligne d'objet
4. Ne pas terminer la ligne d'objet avec un point
5. Utilisez le caractère **impératif** dans la ligne d'objet
6. Enroulez manuellement chaque ligne du corps à 72 caractères
7. Utilisez le corps pour expliquer *quoi* et *pourquoi* plutôt que *comment*

[7 règles du blog de Chris Beam](#) .

### S'engager à une date précise

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

Le paramètre `--date` définit la *date de l'auteur* . Cette date apparaîtra dans la sortie standard de `git log` , par exemple.

Pour forcer la *date de validation* aussi :

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

Le paramètre `date` accepte les formats flexibles pris en charge par la date GNU, par exemple :

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

Lorsque la date ne spécifie pas l'heure, l'heure actuelle sera utilisée et seule la date sera remplacée.

### Sélection des lignes à mettre en scène pour la validation

Supposons que vous ayez beaucoup de changements dans un ou plusieurs fichiers mais que, dans chaque fichier, vous ne souhaitez que commettre certaines des modifications, vous pouvez

sélectionner les modifications souhaitées en utilisant:

```
git add -p
```

ou

```
git add -p [file]
```

Chacune de vos modifications sera affichée individuellement et pour chaque modification, vous serez invité à choisir l'une des options suivantes:

```
y - Yes, add this hunk
n - No, don't add this hunk
d - No, don't add this hunk, or any other remaining hunks for this file.
    Useful if you've already added what you want to, and want to skip over the rest.
s - Split the hunk into smaller hunks, if possible
e - Manually edit the hunk. This is probably the most powerful option.
    It will open the hunk in a text editor and you can edit it as needed.
```

Cela mettra en scène les parties des fichiers que vous choisirez. Ensuite, vous pouvez commettre toutes les modifications par étapes comme ceci:

```
git commit -m 'Commit Message'
```

Les modifications qui n'ont pas été transférées ou validées apparaîtront toujours dans vos fichiers de travail et pourront être validées ultérieurement si nécessaire. Ou si les modifications restantes sont indésirables, elles peuvent être supprimées avec:

```
git reset --hard
```

En plus de diviser un grand changement en de plus petits commits, cette approche est également utile pour *examiner* ce que vous êtes sur le point de commettre. En confirmant individuellement chaque modification, vous avez la possibilité de vérifier ce que vous avez écrit et d'éviter le transfert accidentel de code indésirable, tel que des instructions `println` / logging.

#### Modifier le temps d'un engagement

Vous modifiez le temps d'une validation en utilisant

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

ou même

```
git commit --amend --date="now"
```

#### Modifier l'auteur d'un commit

Si vous faites un commit en tant que mauvais auteur, vous pouvez le modifier, puis le modifier

```
git config user.name "Full Name"
```

```
git config user.email "email@example.com"

git commit --amend --reset-author
```

## La signature GPG s'engage

### 1. Déterminez votre identifiant de clé

```
gpg --list-secret-keys --keyid-format LONG

/Users/davidcondrey/.gnupg/secring.gpg
-----
sec    2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
```

Votre identifiant est un code alphanumérique à 16 chiffres après la première barre oblique.

### 2. Définissez votre identifiant de clé dans votre git config

```
git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
```

### 3. Depuis la version 1.7.9, git commit accepte l'option -S pour attacher une signature à vos commits. L'utilisation de cette option vous demandera votre phrase de passe GPG et ajoutera votre signature au journal de validation.

```
git commit -S -m "Your commit message"
```

Lire S'engager en ligne: <https://riptutorial.com/fr/git/topic/323/s-engager>

Syntaxe

- `git stash list [<options>]`
  - `git stash show [<stash>]`
  - `git stash drop [-q|--quiet] [<stash>]`
  - `git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]`
  - `git stash branch <branchname> [<stash>]`
  - `git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet] [-u|--include-untracked] [-a|--all] [<message>]]`
  - `git stash clear`
  - `git stash create [<message>]`
  - `git stash store [-m|--message <message>] [-q|--quiet] <commit>`

Paramètres

| Paramètre      | Détails  |
|----------------|--|
| montrer        | Affiche les modifications enregistrées dans le cache sous forme de diff entre l'état masqué et son parent d'origine. Lorsqu'aucun <stash> n'est donné, affiche le dernier.   |
| liste          | Listez les caches que vous avez actuellement. Chaque stash est répertorié avec son nom (par exemple, <code>stash @ {0}</code> est le dernier stash, <code>stash @ {1}</code> est celui précédent, etc.), le nom de la branche qui était à l'origine du stash Description de la validation sur laquelle la réserve était basée. |
| pop            | Supprimez un seul état caché de la liste de dissimulation et appliquez-le au-dessus de l'état actuel de l'arborescence de travail.   |
| appliquer      | Comme la <code>pop</code> , mais ne supprimez pas l'état de la liste de réserve.   |
| clair          | Supprimez tous les états cachés. Notez que ces états seront alors sujets à l'élagage, et peuvent être impossibles à récupérer.   |
| laissez tomber | Supprimer un seul état caché de la liste de réserve. Lorsqu'aucun <stash> n'est donné, il supprime le dernier. c.-à-d. <code>stash @ {0}</code> , sinon <stash> doit être une référence de journal stash valide du formulaire <code>stash @ {&lt;revision&gt;}</code> .  |
| créer          | Créez un stash (qui est un objet de validation normal) et retournez son nom d'objet, sans le stocker nulle part dans l'espace de noms de référence. Ceci est destiné à être utile pour les scripts. Ce n'est probablement pas la commande que vous souhaitez utiliser; voir "enregistrer" ci-dessus.                           |
| le magasin     | Stocker un stock donné créé via <code>git stash create</code> (qui est un commit de fusion en suspens) dans la réserve stash, mettant à jour le stog reflog. Ceci est destiné à être utile pour les scripts. Ce n'est probablement pas la commande que vous souhaitez utiliser; voir "enregistrer" ci-dessus.                  |



## Remarques

Le stashing nous permet d'avoir un répertoire de travail propre sans perdre aucune information. Ensuite, il est possible de commencer à travailler sur quelque chose de différent et / ou de changer de branche.

## Exemples

### Qu'est-ce que Stashing?

Lorsque vous travaillez sur un projet, vous pouvez être à mi-chemin d'un changement de branche lorsque le bogue est déclenché contre maître. Vous n'êtes pas prêt à commettre votre code, mais vous ne voulez pas non plus perdre vos modifications. C'est là que git stash est utile.

Exécutez git status sur une branche pour afficher vos modifications non validées:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   business/com/test/core/actions/Photo.c

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

Ensuite, lancez git stash pour enregistrer ces modifications dans une pile:

```
(master) $ git stash
Saved working directory and index state WIP on master:
2f2a6e1 Merge pull request #1 from test/test-branch
HEAD is now at 2f2a6e1 Merge pull request #1 from test/test-branch
```

Si vous avez ajouté des fichiers à votre répertoire de travail, ils peuvent également être cachés. Vous devez juste les mettre en scène d'abord.

```
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    NewPhoto.c

nothing added to commit but untracked files present (use "git add" to track)
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
(master) $
```

Votre répertoire de travail ne contient plus aucune modification apportée. Vous pouvez le voir en relançant le git status :

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Pour appliquer le tout dernier cache, lancez `git stash apply` (vous pouvez également appliquer et supprimer le dernier fichier modifié avec `git stash pop`):

```
(master) $ git stash apply
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   business/com/test/core/actions/Photo.c

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

Notez, cependant, que le stockage ne retient pas la branche sur laquelle vous travailliez. Dans les exemples ci-dessus, l'utilisateur se trouvait sur **Master**. S'ils passent à la branche **dev**, **dev**, et exécutent `git stash apply` la dernière réserve est placée dans la branche **dev**.

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   business/com/test/core/actions/Photo.c

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

### Créer une cachette

Enregistrez l'état actuel du répertoire de travail et l'index (également appelé zone de transfert) dans une pile de stashes.

```
git stash
```

Pour inclure tous les fichiers non suivis dans la réserve, utilisez les `--include-untracked` ou `-u`.

```
git stash --include-untracked
```

Inclure un message avec votre cachet pour le rendre plus facilement identifiable ultérieurement

```
git stash save "<whatever message>"
```

Pour laisser la zone de mise en `--keep-index` dans l'état actuel après le `--keep-index` utilisez les `--keep-index` ou `-k`.

```
git stash --keep-index
```

## Liste des caches enregistrés

```
git stash list
```

Cela listera tous les cachets dans la pile dans l'ordre chronologique inverse. Vous obtiendrez une liste qui ressemble à ceci :

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Vous pouvez faire référence à un cache spécifique par son nom, par exemple `stash@{1}` .

## Montrer la cachette

Affiche les modifications enregistrées dans la dernière réserve

```
git stash show
```

Ou une cachette spécifique

```
git stash show stash@{n}
```

Pour afficher le contenu des modifications enregistrées pour la réserve spécifique

```
git stash show -p stash@{n}
```

## Supprimer la cachette

Supprimer toutes les cachettes

```
git stash clear
```

Supprime la dernière cachette

```
git stash drop
```

Ou une cachette spécifique

```
git stash drop stash@{n}
```

## Appliquez et enlevez la cachette

Pour appliquer la dernière réserve et la retirer de la pile, tapez :

```
git stash pop
```

Pour appliquer une réserve spécifique et la retirer de la pile - tapez :

```
git stash pop stash@{n}
```

## Appliquer stash sans le retirer

Applique la dernière cachette sans la retirer de la pile

```
git stash apply
```

Ou une cachette spécifique

```
git stash apply stash@{n}
```

### Récupération des modifications antérieures de la réserve

Pour obtenir votre plus récente réserve après avoir exécuté `git stash`, utilisez

```
git stash apply
```

Pour voir une liste de vos caches, utilisez

```
git stash list
```

Vous obtiendrez une liste qui ressemble à quelque chose comme ça

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Choisissez un autre `git stash` à restaurer avec le numéro qui apparaît pour la réserve que vous voulez

```
git stash apply stash@{2}
```

### Cachette partielle

Si vous souhaitez ne conserver que *quelques* différences dans votre jeu de travail, vous pouvez utiliser une réserve partielle.

```
git stash -p
```

Et puis sélectionnez interactivement les objets à cacher.

A partir de la version 2.13.0, vous pouvez également éviter le mode interactif et créer un cache partiel avec `pathspec` en utilisant le nouveau mot - clé **push** .

```
git stash push -m "My partial stash" -- app.config
```

### Appliquer une partie d'une cachette avec la caisse

Vous avez fait une réserve et souhaitez ne récupérer que certains des fichiers de cette réserve.

```
git checkout stash@{0} -- myfile.txt
```

### Stashing interactif

Stashing prend l'état sale de votre répertoire de travail, c'est-à-dire vos fichiers de suivi modifiés et vos modifications par étapes, et les enregistre sur une pile de modifications

inachevées que vous pouvez réappliquer à tout moment.

### Stocker uniquement les fichiers modifiés:

Supposons que vous ne vouliez pas cacher les fichiers mis en scène et que vous ne gardiez que les fichiers modifiés pour pouvoir utiliser:

```
git stash --keep-index
```

Qui ne cachera que les fichiers modifiés.

### Stocker les fichiers non suivis:

Stash n'enregistre jamais les fichiers non suivis, il ne cache que les fichiers modifiés et mis en scène. Donc, supposez que si vous avez besoin de cacher les fichiers non suivis, alors vous pouvez utiliser ceci:

```
git stash -u
```

Cela permettra de suivre les fichiers non suivis, mis en scène et modifiés.

### Ne conservez que quelques modifications particulières:

Supposons que vous ne deviez cacher qu'une partie du code du fichier ou seulement certains fichiers de tous les fichiers modifiés et cachés, vous pouvez le faire comme ceci:

```
git stash --patch
```

Git ne stocke pas tout ce qui est modifié, mais vous demande de manière interactive quelles modifications vous souhaitez conserver et que vous souhaitez conserver dans votre répertoire de travail.

### Déplacez votre travail en cours vers une autre branche

Si, tout en travaillant, vous réalisez que vous êtes sur une mauvaise branche et que vous n'avez pas encore créé de commits, vous pouvez facilement déplacer votre travail pour corriger une branche en utilisant un cache:

```
git stash
git checkout correct-branch
git stash pop
```

Rappelez-vous que `git stash pop` appliquera la dernière réserve et la supprimera de la liste de stockage. Pour garder la cachette dans la liste et ne concerne que certaines branches, vous pouvez utiliser:

```
git stash apply
```

### Récupérer une cachette tombée

Si vous venez juste de l'ouvrir et que le terminal est toujours ouvert, vous aurez toujours la valeur de hachage imprimée par `git stash pop` à l'écran:

```
$ git stash pop
[...]
Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

(Notez que la chute de git produit également la même ligne.)

Sinon, vous pouvez le trouver en utilisant ceci:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

Cela vous montrera tous les commits sur les bouts de votre graphe de commit qui ne sont plus référencés depuis une branche ou une balise - chaque commit perdu, y compris chaque commit de cache que vous avez déjà créé, sera quelque part dans ce graphique.

La manière la plus simple de trouver l'engagement que vous souhaitez est probablement de passer cette liste à gitk :

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

Cela lancera un navigateur de référentiel vous montrant *chaque engagement unique dans le référentiel* , qu'il soit accessible ou non.

Vous pouvez remplacer gitk par quelque chose comme git log --graph --oneline --decorate si vous préférez un graphique agréable sur la console par rapport à une application GUI séparée.

Pour repérer les commits cachés, recherchez les messages de validation de ce formulaire:

WIP sur *somebranch* : *commithash* Un ancien message de commit

Une fois que vous connaissez le hachage de la validation souhaitée, vous pouvez l'appliquer comme une réserve:

```
git stash apply $stash_hash
```

Ou vous pouvez utiliser le menu contextuel de gitk pour créer des branches pour tous les commits inaccessibles qui vous intéressent. Après cela, vous pouvez faire ce que vous voulez avec tous les outils habituels. Lorsque vous avez terminé, il suffit de faire sauter ces branches à nouveau.

Lire Se cacher en ligne: <https://riptutorial.com/fr/git/topic/1440/se-cacher>

---

## Chapitre 54: Sous-arbres

### Syntaxe

- `git subtree add -P <prefix> <commit>`
  - `git subtree add -P <prefix> <repository> <ref>`
  - `git subtree pull -P <prefix> <repository> <ref>`
  - `git subtree push -P <prefix> <repository> <ref>`
  - `git subtree merge -P <prefix> <commit>`
  - `git subtree split -P <prefix> [OPTIONS] [<commit>]`

### Remarques

Ceci est une alternative à l'utilisation d'un `submodule` - `submodule`

### Exemples

Créer, extraire et sous-déplacer

---

#### Créer un sous-arbre

Ajoutez une nouvelle télécommande appelée plugin pointant vers le référentiel du plugin:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Ensuite, créez un sous-arbre en spécifiant le nouveau dossier `plugins/demo` prefix. `plugin` est le nom distant, et `master` fait référence à la branche principale du référentiel du sous-arbre:

```
git subtree add --prefix=plugins/demo plugin master
```

---

#### Pull Subtree Updates

Tirez les commits normaux effectués dans le plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

---

#### Mises à jour de sous-arborescence Backport

1. Spécifiez les commits effectués dans le superprojet à renvoyer:

```
git commit -am "new changes to be backported"
```

2. Extraire une nouvelle branche pour la fusion, définir le suivi du référentiel de sous-arborescence:

```
git checkout -b backport plugin/master
```

3. Backports Cherry-pick:

```
git cherry-pick -x --strategy=subtree master
```

4. Poussez les modifications sur la source du plugin:

```
git push plugin backport:master
```

Lire Sous-arbres en ligne: <https://riptutorial.com/fr/git/topic/1634/sous-arbres>



### Syntaxe

- `git log [<options>] [<plage de révision>] [[-] <chemin>]`
- `git log --pretty = short | git shortlog [<options>]`
- `git shortlog [<options>] [<plage de révision>] [[-] <chemin>]`

### Paramètres

| Paramètre   | Détails  |
|---|--|
| <code>-n , --numbered</code>  | Trier la sortie en fonction du nombre de commits par auteur au lieu de l'ordre alphabétique  |
| <code>-s , --summary</code>   | Fournir uniquement un résumé du nombre de validations  |
| <code>-e , --email</code>   | Afficher l'adresse email de chaque auteur  |
| <code>--format [= &lt;format&gt;]</code>                                | Au lieu du sujet de la validation, utilisez d'autres informations pour décrire chaque validation. <format> peut être n'importe quelle chaîne acceptée par l'option <code>--format</code> de <code>git log</code> .                         |
| <code>-w [&lt;width&gt; [, &lt;indent1&gt; [, &lt;indent2&gt;]]]</code> | Linewrap la sortie en enveloppant chaque ligne en <code>width</code> . La première ligne de chaque entrée est indentée par le nombre d' <code>indent1</code> et les lignes suivantes sont indentées par des espaces <code>indent2</code> . |
| <code>&lt;plage de révision&gt;</code>                                  | Afficher uniquement les validations dans la plage de révision spécifiée. Valeur par défaut pour tout l'historique jusqu'à la validation actuelle.  |
| <code>[ -- ] &lt;chemin&gt;</code>                                      | N'affichez que les commits expliquant comment le <code>path</code> correspondance des fichiers a été créé. Il peut être nécessaire de préfixer les chemins avec "-" pour les séparer des options ou de la plage de révision.               |

### Exemples

#### Commit par développeur

Git shortlog est utilisé pour résumer les sorties du journal git et regrouper les commits par auteur.

Par défaut, tous les messages de validation sont affichés, mais l'argument `--summary` ou `-s` ignore les messages et fournit une liste d'auteurs avec leur nombre total de validations.

`--numbered` ou `-n` change l'ordre de l'ordre alphabétique (par auteur croissant) au nombre de commits décroissant.

```
git shortlog -sn          #Names and Number of commits
git shortlog -sne        #Names along with their email ids and the Number of commits
```

ou

```
git log --pretty=format:%ae \  
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

**Remarque:** les engagements de la même personne ne peuvent pas être regroupés lorsque leur nom et / ou leur adresse électronique ont été orthographiés différemment. Par exemple, John Doe et Johnny Doe apparaîtront séparément dans la liste. Pour résoudre ce problème, reportez-vous à la fonctionnalité `.mailmap`.

#### Commit par date

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

#### Nombre total de commits dans une succursale

```
git log --pretty=oneline |wc -l
```

#### Liste de chaque branche et la date de sa dernière révision

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci  
%Cblue%cr%Creset" $k --`"\t"$k";done | sort
```

#### Lignes de code par développeur

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \  
while read filename; do file "$filename"; done | \  
grep -E ': .*text' | sed -E -e 's/: .*//' | \  
while read filename; do git blame --line-porcelain "$filename"; done | \  
sed -n 's/^author //p' | \  
sort | uniq -c | sort -rn
```

#### Liste tous les commits dans un joli format

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn %Cgreen%cr%Creset %s"
```

Cela donnera un bon aperçu de tous les commits (1 par ligne) avec la date, l'utilisateur et le message de validation.

L'option `--pretty` comporte de nombreux espaces réservés, chacun commençant par `%`. Toutes les options peuvent être trouvées [ici](#)

#### Trouver tous les dépôts de Git locaux sur ordinateur

Pour répertorier tous les emplacements du dépôt git sur votre ordinateur, vous pouvez exécuter les opérations suivantes:

```
find $HOME -type d -name ".git"
```

En supposant que vous avez `locate`, cela devrait être beaucoup plus rapide:

```
locate .git |grep git$
```

Si vous avez `gnu locate` ou `mlocate`, cela sélectionnera uniquement les `mlocate git`:

```
locate -ber \\.git$
```

### Affiche le nombre total de commits par auteur

Pour obtenir le nombre total de validations que chaque développeur ou contributeur a effectuées sur un référentiel, vous pouvez simplement utiliser le git shortlog :

```
git shortlog -s
```

qui fournit les noms d'auteur et le nombre d'engagements de chacun.

De plus, si vous souhaitez que les résultats soient calculés sur toutes les branches, ajoutez l'`--all` à la commande:

```
git shortlog -s --all
```

Lire Statistiques Git en ligne: <https://riptutorial.com/fr/git/topic/4609/statistiques-git>

## Chapitre 56: Syntaxe des révisions Git

### Remarques

De nombreuses commandes Git prennent les paramètres de révision comme arguments. Selon la commande, ils désignent un commit spécifique ou, pour les commandes qui parcourent le graphe de révision (comme `git-log (1)`), tous les commits pouvant être atteints à partir de ce commit. Ils sont généralement appelés `<commit>` ou `<rev>` ou `<revision>` dans la description de la syntaxe.

La documentation de référence pour la syntaxe des révisions Git est la page de [manuel gitrevisions \(7\)](#).

Toujours absent de cette page:

- `[_]` La sortie de `git describe`, par exemple `v1.7.4.2-679-g3bee7fb`
- `[_]` `@` seul comme raccourci pour `HEAD`
- `[_]` `@{<n>}`, par exemple `@{-1}`, et `-` signifiant `@{-1}`
- `[_]` `<branchname>@{push}`
- `[_]` `<rev>^@`, pour tous les parents de `<rev>`

Nécessite une documentation séparée:

- `[_]` Référence aux blobs et aux arborescences dans le référentiel et dans l'index: `<rev>:<path>` et `:<n>:<path>` syntaxe `:<n>:<path>`
- `[_]` Les plages de révision comme `A..B`, `A...B`, `B ^A`, `A^1` et la limitation de révision comme `-<n>`, `--since`

### Exemples

#### Spécification de la révision par nom d'objet

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

Vous pouvez spécifier la révision (ou en réalité tout objet: tag, arborescence, contenu du répertoire, blob, c'est-à-dire le contenu du fichier) en utilisant le nom d'objet SHA-1, soit une chaîne hexadécimale de 40 octets, soit une sous-chaîne unique.

#### Noms de référence symboliques: branches, tags, branches de suivi à distance

```
$ git log master      # specify branch
$ git show v1.0       # specify tag
$ git show HEAD       # specify current branch
$ git show origin     # specify default remote-tracking branch for remote 'origin'
```

Vous pouvez spécifier une révision en utilisant un nom de référence symbolique, qui inclut des branches (par exemple, «`master`», «`next`», «`maint`»), des balises (par exemple, «`v1.0`», «`v0.6.3-rc2`»), `remote-` suivi des branches (par exemple «`origine`», «`origine / maître`») et références spéciales telles que «`HEAD`» pour la branche actuelle.

Si le nom de référence symbolique est ambigu, par exemple si vous avez à la fois une branche et une balise nommée 'fix' (il est déconseillé d'avoir une branche et une balise du même nom), vous devez spécifier le type de référence que vous voulez utiliser:

```
$ git show heads/fix    # or 'refs/heads/fix', to specify branch
$ git show tags/fix     # or 'refs/tags/fix', to specify tag
```

La révision par défaut: HEAD

```
$ git show          # equivalent to 'git show HEAD'
```

'HEAD' nomme le commit sur lequel vous avez basé les modifications dans l'arborescence de travail et est généralement le nom symbolique de la branche en cours. Beaucoup (mais pas toutes) de commandes qui prennent le paramètre de révision par défaut à 'HEAD' s'il est manquant.

Références de reflog: @ { }

```
$ git show @{1}          # uses reflog for current branch
$ git show master@{1}    # uses reflog for branch 'master'
$ git show HEAD@{1}      # uses 'HEAD' reflog
```

Une ref, généralement une branche ou HEAD, suivie du suffixe @ avec une spécification ordinaire incluse dans une paire d'accolades (par exemple {1} , {15} ) spécifie la n-ième valeur antérieure de cette référence dans votre référentiel **local** . Vous pouvez vérifier les entrées de reflog récentes avec la commande `git reflog` , ou l' `--walk-reflogs / -g` pour `git log` .

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving from af40944bda352190f05d22b7cb8fe88beb17f3a7 to master
af40944 HEAD@{4}: checkout: moving from master to v2.6.3

$ git reflog gitweb-docs
4ebf58d gitweb-docs@{0}: branch: Created from master
```

Remarque : l'utilisation de reflogs a pratiquement remplacé les anciens mécanismes d'utilisation de la référence ORIG\_HEAD (à peu près équivalente à HEAD@{1} ).

Références de reflog: @ { }

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago} # or HEAD@{5.minutes.ago}
```

Une ref suivie du suffixe @ avec une spécification de date dans une accolade (par exemple {yesterday} , {1 month 2 weeks 3 days 1 hour 1 second ago} ou {1979-02-26 18:30:00} ) spécifie la valeur de la référence à un moment antérieur (ou le point le plus proche). Notez que cela recherche l'état de votre ref **local** à un moment donné; Par exemple, ce qui était dans votre branche locale «*maître*» la semaine dernière.

Vous pouvez utiliser `git reflog` avec un spécificateur de date pour rechercher l'heure exacte à laquelle vous avez fait quelque chose dans le référentiel local.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: moving to HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Fast-forward
```

Branche suivie / en amont: @ {en amont}

```
$ git log @{upstream}..      # what was done locally and not yet published, current branch
$ git show master@{upstream} # show upstream of branch 'master'
```

Le suffixe `@{upstream}` ajouté à un nom de branche (forme `<branchname>@{u}` ) fait référence à la branche sur laquelle la branche spécifiée par `branchname` est configurée (configurée avec `branch.<name>.remote` et `branch.<name>.merge` ou avec `git branch --set-upstream-to=<branch>` ). Un nom de branche manquant est celui par défaut.

Avec la syntaxe des plages de révision, il est très utile de voir les validations de votre branche en amont (validations dans votre référentiel local non encore présent en amont) et ce que vous êtes en retard (validation en amont non fusionnée dans une branche locale) ou tous les deux:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}... # same as ...@{u}
```

**Commit la chaîne d'ascendance:** `^`, `~` , etc.

```
$ git reset --hard HEAD^          # discard last commit
$ git rebase --interactive HEAD~5 # rebase last 4 commits
```

Un suffixe `^` à un paramètre de révision signifie le premier parent de cet objet de validation. `^<n>` signifie le `<n>` -ième parent (c.-à-d. que `<rev>^` est équivalent à `<rev>^1` ).

Un suffixe `~<n>` à un paramètre de révision signifie l'objet de validation qui est l'ancêtre `<n>` de génération de l'objet de validation nommé, en suivant uniquement les premiers parents. Cela signifie que, par exemple, `<rev>~3` équivaut à `<rev>^^^` . Comme raccourci, `<rev>~` signifie `<rev>~1` , et équivaut à `<rev>^1` , ou `<rev>^` en bref.

Cette syntaxe est composable.

---

Pour trouver ces noms symboliques, vous pouvez utiliser la commande `git name-rev` :

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Notez que `--pretty=oneline` et non `--oneline` doit être utilisé dans l'exemple suivant

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

**Déréférencement des branches et des tags:** `^ 0`, `^ { }`

Dans certains cas, le comportement d'une commande dépend de son nom, de son nom ou d'une révision arbitraire. Vous pouvez utiliser la syntaxe de "dé-référencement" si vous en avez besoin.

Un suffixe `^` suivi d'un nom de type d'objet ( `tag` , `commit` , `tree` , `blob` ) placé entre accolades (par exemple, `v0.99.8^{commit}` ) signifie déréférencer l'objet à `<rev>` manière récursive jusqu'à ce qu'un objet de type `<type>` est trouvé ou l'objet ne peut plus être déréférencé. `<rev>^0` est un raccourci pour `<rev>^{commit}` .

```
$ git checkout HEAD^0 # equivalent to 'git checkout --detach' in modern Git
```

Un suffixe ^ suivi d'une paire d'accolades vide (par exemple, v0.99.8^{ } ) signifie déréférencer la balise récursivement jusqu'à ce qu'un objet non-balise soit trouvé.

Comparer

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

avec

```
$ git show v1.0^{ }
$ git cat-file -p v1.0^{ }
$ git replace --edit v1.0^{ }
```

**Plus jeune engagement correspondant: ^ {/ },: /**

```
$ git show HEAD^{/fix nasty bug} # find starting from HEAD
$ git show ':/fix nasty bug' # find starting from any branch
```

Un deux-points ( ' : ' ), suivi d'une barre oblique ( ' / ' ), suivi d'un texte, nomme un commit dont le message de validation correspond à l'expression régulière spécifiée. Ce nom retourne le commit le plus jeune correspondant qui est accessible depuis *n'importe quelle* référence. L'expression régulière peut correspondre à n'importe quelle partie du message de validation. Pour faire correspondre les messages commençant par une chaîne, on peut utiliser par exemple `:/^foo` . La séquence spéciale `:/!` est réservé aux modificateurs à ce qui est apparié. `:/!-foo` effectue une correspondance négative, alors que `:/!!foo` correspond à un littéral! personnage, suivi de `foo` .

Un suffixe ^ à un paramètre de révision, suivi d'une paire d'accolades contenant un texte dirigé par une barre oblique, est identique à la syntaxe `:/<text>` ci-dessous: il retourne la validation la plus récente accessible depuis la `<rev>` avant ^ .

Lire Syntaxe des révisions Git en ligne: <https://riptutorial.com/fr/git/topic/3735/syntaxe-des-revisions-git>

### Introduction

Contrairement à l'envoi par Git où vos modifications locales sont envoyées au serveur du référentiel central, tirer avec Git prend le code actuel sur le serveur et le «tire» du serveur du référentiel vers votre ordinateur local. Cette rubrique explique le processus d'extraction du code à partir d'un référentiel à l'aide de Git, ainsi que les situations rencontrées lors de l'extraction d'un code différent dans la copie locale.

### Syntaxe

- `git pull [options] [<repository> [<refspec> ...]]`

### Paramètres

| Paramètres   | Détails  |
|--|--|
| <code>--quiet</code>   | Pas de sortie de texte   |
| <code>-q</code>  | sténographie pour <code>--quiet</code>   |
| <code>--verbose</code>   | sortie textuelle verbeuse. Passé pour récupérer et fusionner / rebaser les commandes respectivement. |
| <code>-v</code>  | sténographie pour <code>--verbose</code>   |
| <code>--[no-]recurse-submodules[=<br/>  on-demand   no]</code> | Récupère de nouveaux commits pour les sous-modules?<br>(Pas que ce n'est pas un pull / checkout)     |

### Remarques

`git pull` lance `git fetch` avec les paramètres donnés et appelle `git merge` pour fusionner les branches récupérées dans la branche en cours.

### Exemples

#### Mise à jour avec les modifications locales

Lorsque des modifications locales sont présentes, la commande `git pull` abandonne la création de rapports:

```
erreur: vos modifications locales apportées aux fichiers suivants seraient écrasées  
par la fusion
```

Pour mettre à jour (comme `svn update` fait avec `subversion`), vous pouvez lancer:

```
git stash  
git pull --rebase  
git stash pop
```

Un moyen pratique pourrait être de définir un alias en utilisant:



```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

Ensuite, vous pouvez simplement utiliser:

```
git up
```

Tirez le code de la télécommande

```
git pull
```

Tirez, écrasez local

```
git fetch
git reset --hard origin/master
```

**Attention:** Bien que les `reset --hard` aide de la commande `reset --hard` puissent être récupérées à l'aide de `reflog` et `reset`, les modifications non validées sont définitivement supprimées.

Modifiez l' `origin` et le `master` sur la télécommande et la branche vers lesquelles vous voulez forcer, respectivement, s'ils sont nommés différemment.

Garder une histoire linéaire en tirant

### Rebasing en tirant

Si vous récupérez de nouveaux commits depuis le référentiel distant et que vous avez des modifications locales sur la branche en cours, git fusionne automatiquement la version distante et votre version. Si vous souhaitez réduire le nombre de fusions sur votre succursale, vous pouvez [demander](#) à git de [redéfinir](#) vos commits sur la version distante de la succursale.

```
git pull --rebase
```

### En faisant le comportement par défaut

Pour en faire le comportement par défaut pour les branches nouvellement créées, tapez la commande suivante:

```
git config branch.autosetuprebase always
```

Pour changer le comportement d'une branche existante, utilisez ceci:

```
git config branch.BRANCH_NAME.rebase true
```

Et

```
git pull --no-rebase
```

Pour effectuer une traction normale de fusion.

---

### Vérifiez si fast-forwardable

Pour autoriser uniquement la diffusion rapide de la branche locale, vous pouvez utiliser:

```
git pull --ff-only
```

Cela affichera une erreur lorsque la branche locale ne peut pas être transférée rapidement et doit être rebasée ou fusionnée avec la branche amont.

### Pull, "permission refusée"

Certains problèmes peuvent survenir si le dossier `.git` dispose pas des droits `.git` . Résoudre ce problème en définissant le propriétaire du dossier complet `.git` . Parfois, il arrive qu'un autre utilisateur récupère et modifie les droits du ou des fichiers `.git` .

Pour résoudre le problème:

```
chown -R youruser:yourgroup .git/
```

### Extraction de modifications dans un référentiel local

#### Traction simple

Lorsque vous travaillez sur un référentiel distant (disons GitHub) avec une autre personne, vous souhaitez à un moment donné partager vos modifications avec eux. Une fois qu'ils ont **poussé** leurs changements à un dépôt distant, vous pouvez récupérer ces changements en *tirant* à partir de ce référentiel.

```
git pull
```

Le fera, dans la majorité des cas.

---

#### Tirez d'une autre télécommande ou branche

Vous pouvez extraire des modifications d'une autre télécommande ou succursale en spécifiant leur nom

```
git pull origin feature-A
```

Tirera l' origin formulaire de branche feature-A dans votre agence locale. Notez que vous pouvez fournir directement une URL au lieu d'un nom distant et un nom d'objet tel qu'un commit SHA au lieu d'un nom de branche.

---

#### Traction manuelle

Pour imiter le comportement d'un pull git, vous pouvez utiliser git fetch puis git merge

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

Cela peut vous donner plus de contrôle et vous permet d'inspecter la branche distante avant de la fusionner. En effet, après avoir récupéré, vous pouvez voir les branches distantes avec `git branch -a`, et les vérifier avec

```
git checkout -b local-branch-name origin/feature-A # checkout the remote branch
# inspect the branch, make commits, squash, ammend or whatever
git checkout merging-branches # moving to the destination branch
git merge local-branch-name # performing the merge
```

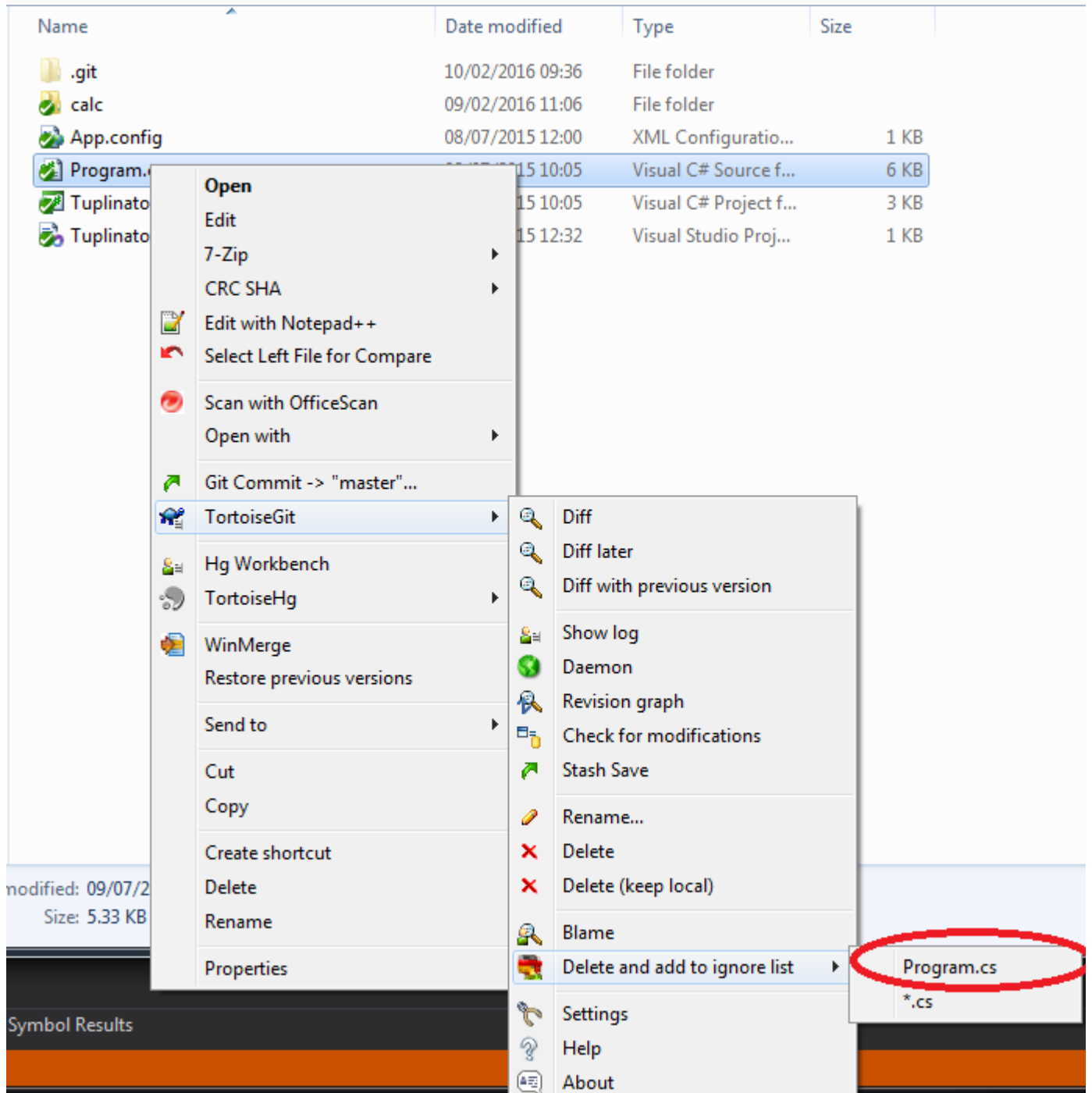
Cela peut être très pratique lors du traitement des requêtes d'extraction.

Lire Tirant en ligne: <https://riptutorial.com/fr/git/topic/1308/tirant>

Exemples

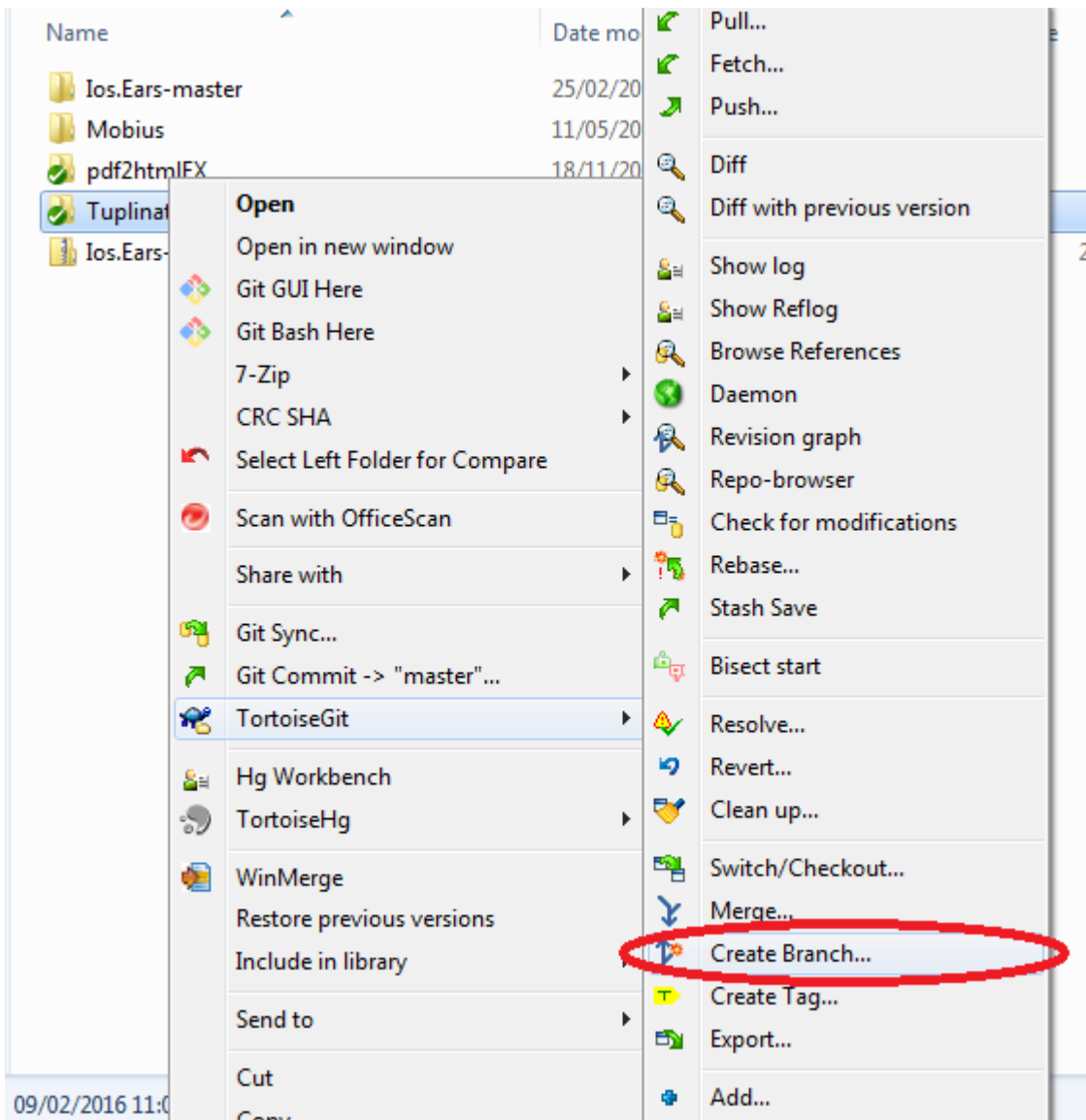
Ignorer les fichiers et les dossiers

Ceux qui utilisent TortoiseGit UI cliquez avec le bouton droit de la souris sur le fichier (ou le dossier) que vous voulez ignorer -> TortoiseGit -> Delete and add to ignore list ignorés. va sortir Cliquez sur Ok et vous devriez avoir terminé.

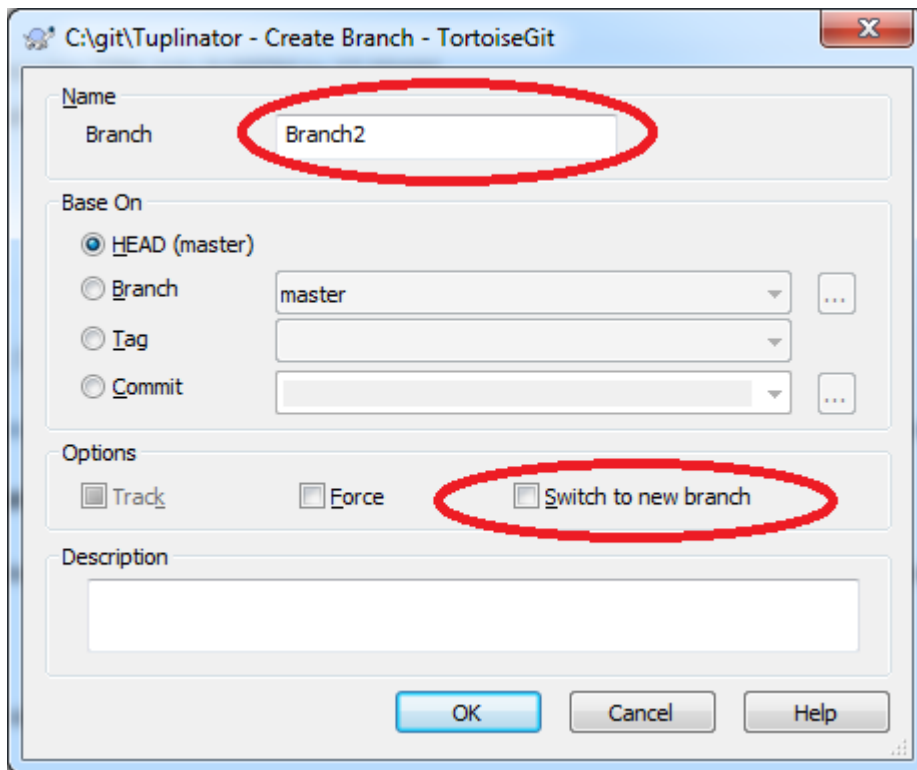


Ramification

Pour ceux qui utilisent l'interface utilisateur pour Create Branch... branche, cliquez avec le bouton droit de la souris sur le dépôt, puis sur Tortoise Git -> Create Branch...

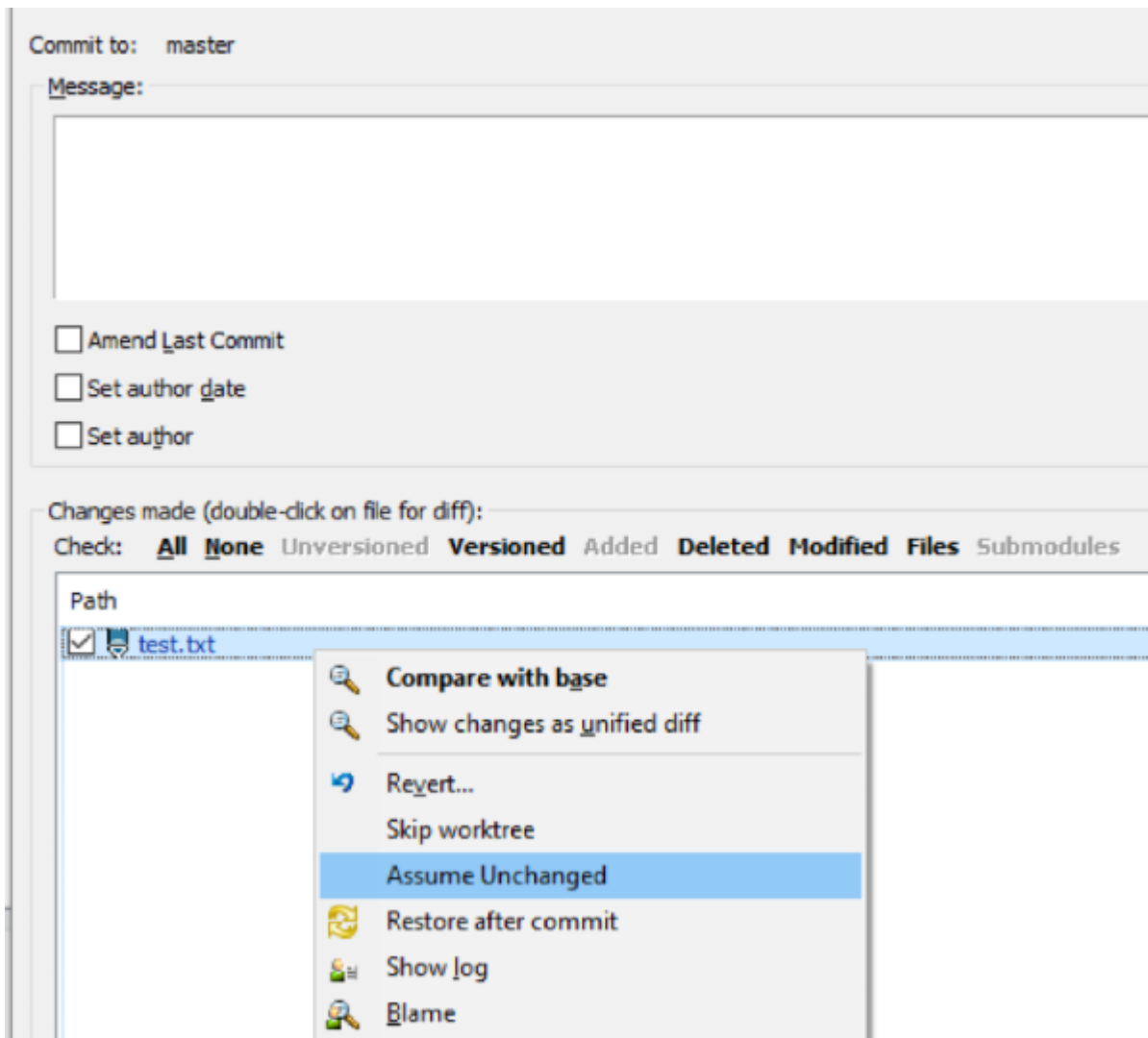


Une nouvelle fenêtre s'ouvrira -> Give branch a name -> Cochez la case Switch to new branch (il est probable que vous souhaitiez commencer à travailler avec cette branche après la Switch to new branch ). -> Cliquez sur OK et vous devriez avoir terminé.



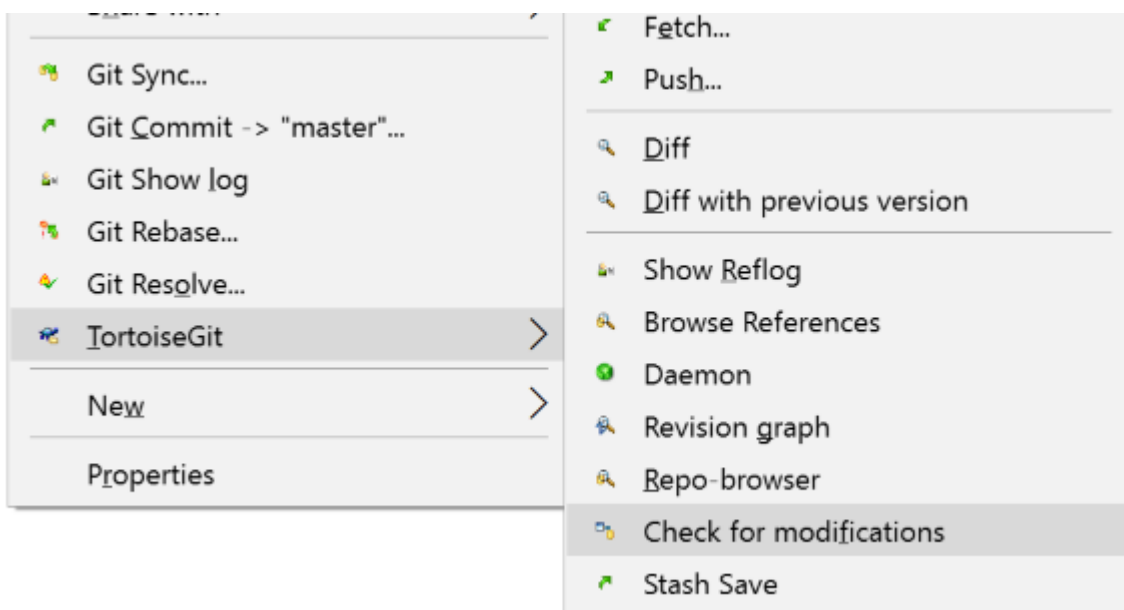
Assumer inchangé

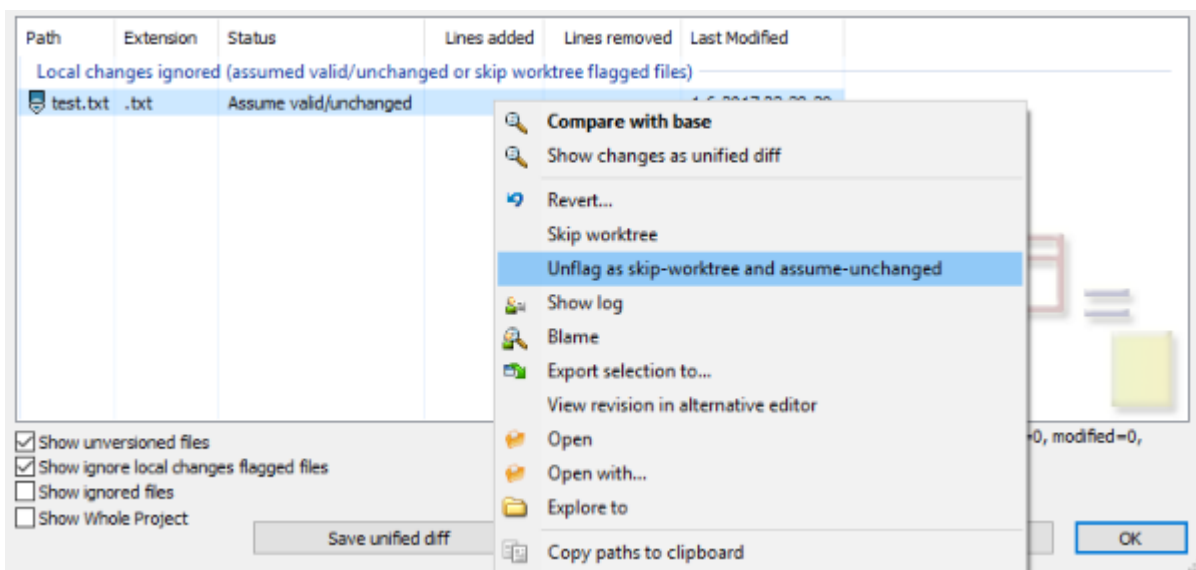
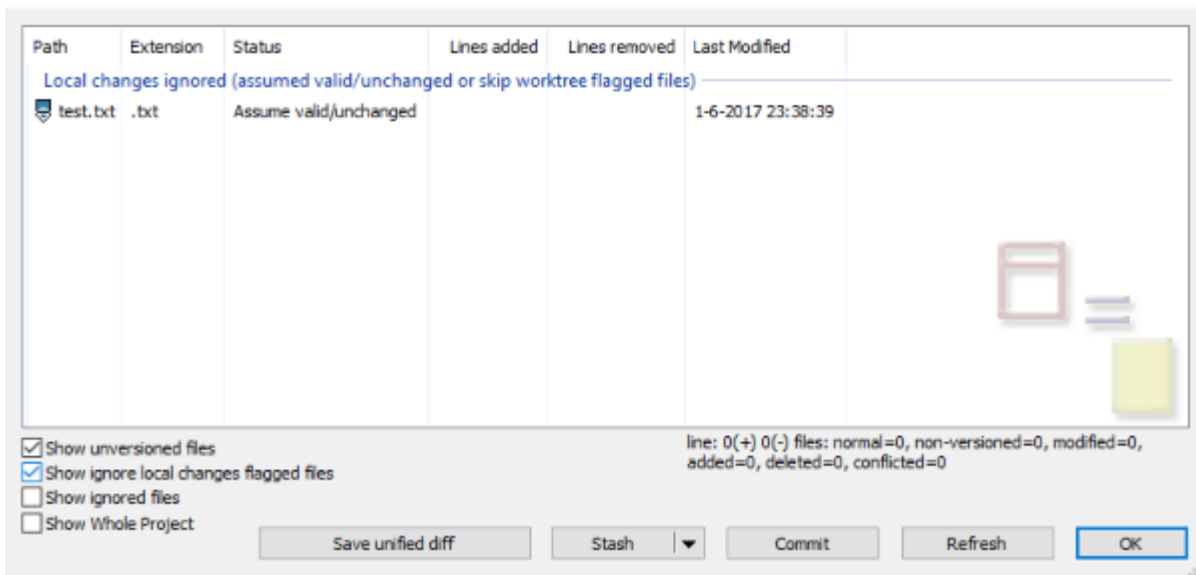
Si un fichier est modifié, mais que vous n'aimez pas le valider, définissez le fichier comme étant "inchangé"



Revenir "Assumer inchangé"

Besoin de quelques étapes:





Squash commet

### Le moyen facile

Cela ne fonctionnera pas s'il y a des commits de fusion dans votre sélection



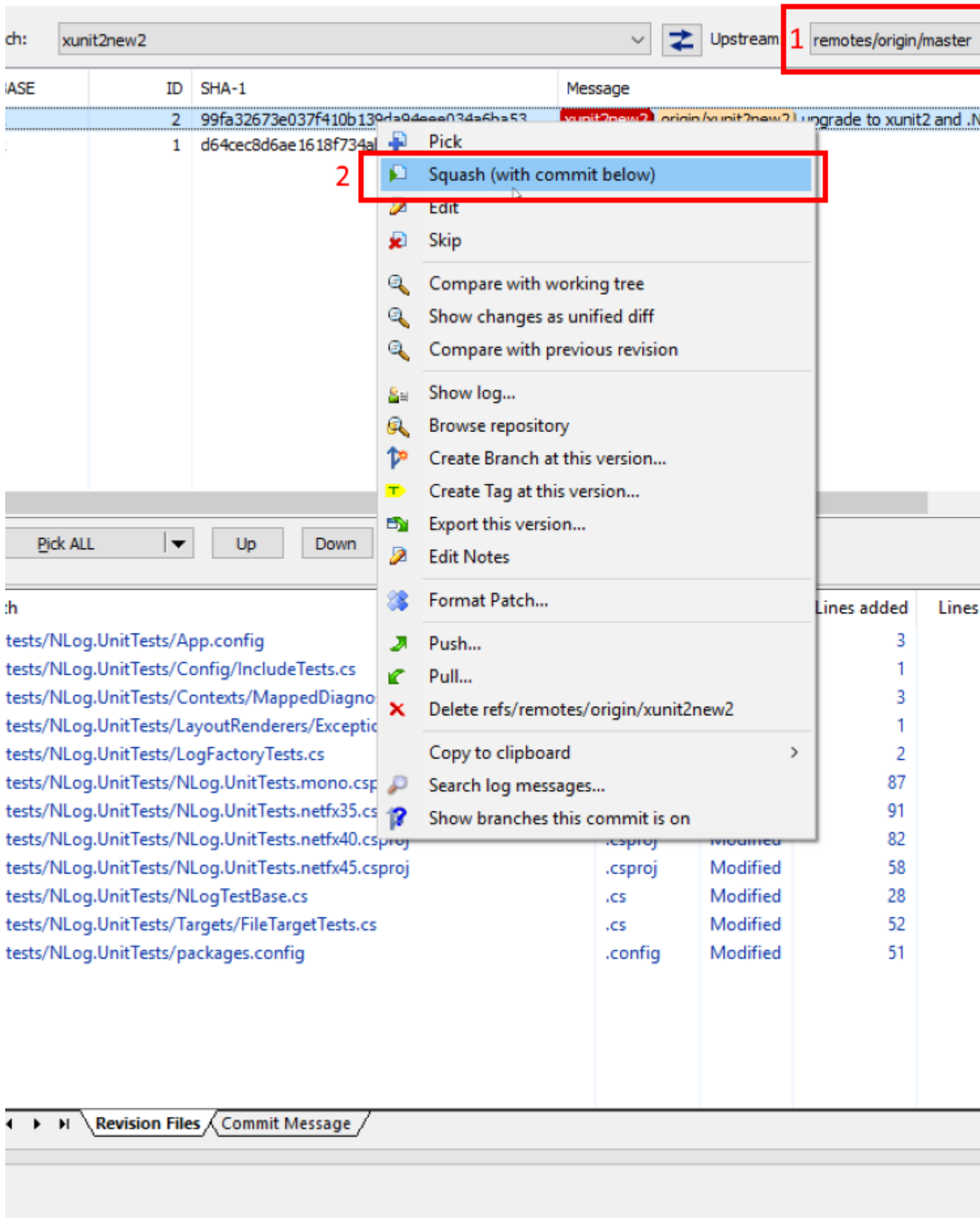
xunit2new2 From: 11- 8-2004 To: 19- 6-2017

| Graph | SHA-1                   | Actions | Message   |
|-------|-------------------------|---------|---|
|       | 00000000000000000000... |         | Working tree changes                                    |
|       | 99fa32673e03741...      | !       | <b>xunit2new2</b> origin/xunit2new2 upgrade to xunit2 a |
|       | d64cec8d6ae1618f73...   | !       | config AppVeor and Travis:                              |
|       | 6354a596ff1e533f2af...  | !       | v4.4.11 Update CHANGELOG.md                             |
|       | 24a2f57d6cb3a78816...   | !       | Update appveyor.yml                                     |
|       | 04aacc1b4cccf1c63dd...  | ! +     | master Merge pull request #2164 from s                  |
|       | c651f88ad0bfb76bc64...  | !       | JsonLayout - IncludeMdc and IncludeMdc                  |
|       | f0a8adc354ad66d165...   | ! +     | Merge pull request #2171 from NLog/son                  |
|       | 7855f63175bedaacf3d...  | ! +     | sonar-fork origin/sonar-fork Don't run S                |
|       | db5355b1e65b81d46a...   | !       | Merge pull request #2153 from NLog/fix-                 |
|       | 4eb939979266f74a0e...   | !       | fix sonar cache   |
|       | 83ee61133a4f653c4c...   | !       | v4.4.10   |
|       | 95851865a82758e46a...   | !       | v4.4.10 update changelog                                |

- Compare re
- Revert chan
- Combine to
- Format Pat
- Copy to clip
- Search log r

**Le moyen avancé**

Lancez la boîte de dialogue de rebase:



Lire TortoiseGit en ligne: <https://riptutorial.com/fr/git/topic/5150/tortoisegit>

### Syntaxe

- `git remote [-v | --verbose]`
  - `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>`
  - `git remote rename <old> <new>`
  - `git remote remove <name>`
  - `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
  - `git remote set-branches [--add] <name> <branch>...`
  - `git remote get-url [--push] [--all] <name>`
  - `git remote set-url [--push] <name> <newurl> [<oldurl>]`
  - `git remote set-url --add [--push] <name> <newurl>`
  - `git remote set-url --delete [--push] <name> <url>`
  - `git remote [-v | --verbose] show [-n] <name>...`
  - `git remote prune [-n | --dry-run] <name>...`
  - `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]`

### Exemples

#### Ajout d'un nouveau référentiel distant

```
git remote add upstream git-repository-url
```

Ajoute le dépôt git distant représenté par `git-repository-url` tant que nouvelle télécommande nommée en `upstream` du dépôt `git`

#### Mise à jour à partir du référentiel en amont

En supposant que vous définissiez le flux amont (comme dans la configuration d'un référentiel en amont)

```
git fetch remote-name
git merge remote-name/branch-name
```

La commande `pull` combine une `fetch` et une `merge` .

```
git pull
```

La commande `pull` with `--rebase` flag combine une `fetch` et une `rebase` au lieu d'une `merge` .

```
git pull --rebase remote-name branch-name
```

#### `ls-remote`

`git ls-remote` est une commande unique qui vous permet d'interroger un dépôt à distance sans avoir à le cloner / le récupérer en premier .

Il listera les refs / heads et les refs / tags de ce repo distant.

Vous verrez parfois des `refs/tags/v0.1.6` et des `refs/tags/v0.1.6 refs/tags/v0.1.6^{}` : le `^{}` pour lister le tag annoté déréférencé (ie le commit sur lequel pointe ce tag)

Depuis git 2.8 (mars 2016), vous pouvez éviter cette double saisie pour une balise et lister

directement les balises déréférencées avec:

```
git ls-remote --ref
```

Il peut également aider à résoudre l'url réel utilisé par un dépôt à distance lorsque vous avez le paramètre de configuration " url.<base>.insteadOf ".

Si git remote --get-url <aremotename> renvoie <https://server.com/user/repo> et que vous avez défini git config url.ssh://git@server.com:.insteadOf https://server.com/ :

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

### Suppression d'une succursale distante

Pour supprimer une branche distante dans Git:

```
git push [remote-name] --delete [branch-name]
```

ou

```
git push [remote-name] :[branch-name]
```

### Suppression des copies locales des branches distantes supprimées

Si une branche distante a été supprimée, votre référentiel local doit être averti de l'élaguer.

Pour éliminer les branches supprimées d'une télécommande spécifique:

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

Pour éliminer les branches supprimées de toutes les télécommandes:

```
git fetch --all --prune
```

### Afficher des informations sur une télécommande spécifique

Produire des informations sur une télécommande connue: origin

```
git remote show origin
```

Imprimer uniquement l'URL de la télécommande:

```
git config --get remote.origin.url
```

Avec 2.7+, il est également possible de faire, ce qui est sans doute meilleur que le précédent qui utilise la commande config .

```
git remote get-url origin
```

### Liste des télécommandes existantes

Liste de toutes les télécommandes existantes associées à ce référentiel:

```
git remote
```

Liste toutes les télécommandes existantes associées à ce référentiel en détail , y compris les fetch et push URL:

```
git remote --verbose
```

ou simplement

```
git remote -v
```

## Commencer

### Syntaxe pour pousser vers une branche distante

```
git push <remote_name> <branch_name>
```

### Exemple

```
git push origin master
```

### Définir en amont sur une nouvelle succursale

Vous pouvez créer une nouvelle branche et y accéder en utilisant

```
git checkout -b AP-57
```

Après avoir utilisé `git checkout` pour créer une nouvelle branche, vous devez définir cette origine en amont pour utiliser

```
git push --set-upstream origin AP-57
```

Après cela, vous pouvez utiliser `git push` pendant que vous êtes sur cette branche.

### Changer un référentiel distant

Pour modifier l'URL du référentiel vers lequel vous souhaitez que votre télécommande pointe, vous pouvez utiliser l'option `set-url` , comme ceci:

```
git remote set-url <remote_name> <remote_repository_url>
```

Exemple:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

### Changer l'URL de Git Remote

Vérifiez la télécommande existante

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
```

```
# origin https://github.com/username/repo.git (push)
```

Modification de l'URL du référentiel

```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```

Vérifier la nouvelle URL distante

```
git remote -v
# origin https://github.com/username/repo2.git (fetch)
# origin https://github.com/username/repo2.git (push)
```

### Renommer une télécommande

Pour renommer la télécommande, utilisez la commande `git remote rename`

La commande `git remote rename` prend deux arguments:

- Un nom distant existant, par exemple: **origine**
- Un nouveau nom pour la télécommande, par exemple: **destination**

Obtenir le nom distant existant

```
git remote
# origin
```

Vérifier la télécommande existante avec l'URL

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/username/repo.git (push)
```

Renommer la télécommande

```
git remote rename origin destination
# Change remote name from 'origin' to 'destination'
```

Vérifier le nouveau nom

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/username/repo.git (push)
```

### === Erreurs possibles ===

1. Impossible de renommer la section de configuration 'remote. [Old name]' en 'remote. [New name]'

Cette erreur signifie que la télécommande que vous avez essayée avec l'ancien nom distant ( **origine** ) n'existe pas.

2. Remote [nouveau nom] existe déjà.

Le message d'erreur est explicite.

### Définir l'URL d'une télécommande spécifique

Vous pouvez changer l'URL d'une télécommande existante par la commande

```
git remote set-url remote-name url
```

### Obtenir l'URL d'une télécommande spécifique

Vous pouvez obtenir l'URL d'une télécommande existante en utilisant la commande

```
git remote get-url <name>
```

Par défaut, ce sera

```
git remote get-url origin
```

Lire [Travailler avec des télécommandes en ligne](https://riptutorial.com/fr/git/topic/243/travailler-avec-des-telecommandes):

<https://riptutorial.com/fr/git/topic/243/travailler-avec-des-telecommandes>

## Chapitre 60: Utiliser un fichier .gitattributes

### Exemples

#### Désactiver la normalisation de fin de ligne

Créez un fichier .gitattributes dans la racine du projet contenant:

```
* -text
```

Cela équivaut à définir `core.autocrlf = false` .

#### Normalisation automatique de fin de ligne

Créez un fichier .gitattributes dans la racine du projet contenant:

```
* text=auto
```

Cela entraînera la validation de tous les fichiers texte (identifiés par Git) avec LF, mais leur extraction en fonction de la valeur par défaut du système d'exploitation hôte.

Ceci est équivalent aux valeurs `core.autocrlf` défaut `core.autocrlf` recommandées de:

- `input` sur Linux / macOS
- `true` sur Windows

#### Identifier les fichiers binaires

Git est assez efficace pour identifier les fichiers binaires, mais vous pouvez spécifier explicitement quels fichiers sont binaires. Créez un fichier .gitattributes dans la racine du projet contenant:

```
*.png binary
```

`binary` est un attribut de macro `-diff -merge -text` équivalent à `-diff -merge -text` .

#### Modèles .gitattribute pré-remplis

Si vous ne savez pas quelles règles lister dans votre fichier .gitattributes ou si vous souhaitez simplement ajouter des attributs généralement acceptés à votre projet, vous pouvez choisir ou générer un fichier .gitattributes à l' `l'` .gitattributes :

- <https://gitattributes.io/>
- <https://github.com/alexkaratarakis/gitattributes>

Lire Utiliser un fichier .gitattributes en ligne:

<https://riptutorial.com/fr/git/topic/1269/utiliser-un-fichier--gitattributes>



### Syntaxe

- `git worktree ajouter [-f] [--detach] [--checkout] [-b <nouvelle-branche>] <chemin> [<branche>]`
- `git worktree prune [-n] [-v] [--expire <expire>]`
- `git worktree list [--porcelain]`

### Paramètres

| Paramètre  | Détails  |
|--|--|
| <code>-f --force</code>                                  | Par défaut, add refuse de créer un nouvel arbre de travail lorsque <code>&lt;branche&gt;</code> est déjà extrait par un autre arbre de travail. Cette option remplace cette sauvegarde.  |
| <code>-b &lt;new-branch&gt; -B &lt;new-branch&gt;</code> | Avec add, créez une nouvelle branche nommée <code>&lt;new-branch&gt;</code> partir de <code>&lt;branche&gt;</code> et extrayez <code>&lt;new-branch&gt;</code> dans le nouvel arbre de travail. Si <code>&lt;branche&gt;</code> est omis, la valeur par défaut est <code>HEAD</code> . Par défaut, <code>-b</code> refuse de créer une nouvelle branche si elle existe déjà. <code>-B</code> remplace cette sauvegarde, en réinitialisant <code>&lt;new-branch&gt;</code> à <code>&lt;branche&gt;</code> . |
| <code>--détacher</code>                                  | Avec add, détachez <code>HEAD</code> dans le nouvel arbre de travail.  |
| <code>- [no-] checkout</code>                            | Par défaut, add <code>--no-checkout out &lt;branche&gt;</code> , cependant, <code>--no-checkout</code> peut être utilisé pour supprimer la récupération afin de réaliser des personnalisations, telles que la configuration de la récupération partielle.  |
| <code>-n --dry-run</code>                                | Avec le pruneau, ne retirez rien; il suffit de rapporter ce que cela supprimerait.   |
| <code>--porcelaine</code>                                | Avec liste, sortie dans un format facile à analyser pour les scripts. Ce format restera stable pour toutes les versions de Git et quelle que soit la configuration de l'utilisateur.   |
| <code>-v --verbose</code>                                | Avec prune, signalez tous les retraits.  |
| <code>--expire &lt;time&gt;</code>                       | Avec Prune, n'expire que les arbres de travail inutilisés plus anciens que <code>&lt;time&gt;</code> .   |

### Remarques

Voir la documentation officielle pour plus d'informations: <https://git-scm.com/docs/git-worktree>

### Exemples

Utiliser un worktree

Vous êtes en train de travailler sur une nouvelle fonctionnalité, et votre patron vient vous demander de réparer quelque chose immédiatement. Vous pouvez généralement utiliser `git stash` pour stocker vos modifications temporairement. Cependant, à ce stade, votre arbre de travail est dans un état de désordre (avec des fichiers nouveaux, déplacés et supprimés, ainsi que d'autres éléments) et vous ne souhaitez pas perturber votre progression.

En ajoutant un `worktree`, vous créez une arborescence de travail liée temporaire pour effectuer le correctif d'urgence, supprimez-le une fois terminé, puis reprenez votre session de codage antérieure:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... work work work ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

Remarque: dans cet exemple, le correctif est toujours dans la branche de correctif d'urgence. À ce stade, vous souhaiterez probablement `git merge` ou `git format-patch` et ensuite supprimer la branche de correctif d'urgence.

### Déplacement d'un `worktree`

Actuellement (à partir de la version 2.11.0), aucune fonctionnalité intégrée ne permet de déplacer un `Worktree` existant. Ceci est répertorié comme un bug officiel (voir [https://git-scm.com/docs/git-worktree#\\_bugs](https://git-scm.com/docs/git-worktree#_bugs)) .

Pour contourner cette limitation, il est possible d'effectuer des opérations manuelles directement dans les fichiers de référence `.git` .

Dans cet exemple, la copie principale du dépôt réside dans `/home/user/project-main` et le `worktree` secondaire se trouve dans `/home/user/project-1` et nous voulons le déplacer dans `/home/user/project-2` .

N'effectuez aucune commande `git` entre ces étapes, sinon le ramasse-miettes peut être déclenché et les références à l'arborescence secondaire peuvent être perdues. Effectuez ces étapes du début jusqu'à la fin sans interruption:

1. Modifiez le fichier `.git` du `.git` pour qu'il pointe vers le nouvel emplacement dans l'arborescence principale. Le fichier `/home/user/project-1/.git` devrait maintenant contenir les éléments suivants:

```
gitdir: /home/user/project-main/.git/worktrees/project-2
```

2. Renommez le `worktree` à l'intérieur du répertoire `.git` du projet principal en déplaçant le répertoire `worktree` qui s'y trouve:

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-main/.git/worktrees/project-2
```

3. Modifiez la référence dans `/home/user/project-main/.git/worktrees/project-2/gitdir` pour pointer vers le nouvel emplacement. Dans cet exemple, le fichier aurait le contenu suivant:

```
/home/user/project-2/.git
```

4. Enfin, déplacez votre `worktree` vers le nouvel emplacement:

```
$ mv /home/user/project-1 /home/user/project-2
```

Si vous avez tout fait correctement, la liste des worktrees existants doit faire référence au nouvel emplacement:

```
$ git worktree list
/home/user/project-main 23f78ad [master]
/home/user/project-2    78ac3f3 [branch-name]
```

Il devrait maintenant aussi être sûr de lancer `git worktree prune` .

Lire Worktrees en ligne: <https://riptutorial.com/fr/git/topic/3801/worktrees>

# Crédits

| S. No | Chapitres   | Contributeurs  |
|-------|---|--|
| 1     | Démarrer avec Git   | Ajedi32, Ala Eddine JEBALI, Allan Burleson, Amitay Stern, Andy Hayden, AnimiVulpis, ArtOfWarfare, bahrep, Boggin, Brian, Community, Craig Brett, Dan Hulme, ericdwang, eykanal, Fernando Hoces De La Guardia, Fred Barclay, Henrique Barcelos, intboolstring, Irfan, Jackson Blankenship, janos, Jav_Rock, jeffdill12, JonasCz, JonyD, Joseph Dasenbrock, Kageetai, Karthik, KartikKannapur, Kayvan N, Knu, Lambda Ninja, maccard, Marek Skiba, Mateusz Piotrowski, Mingle Li, mouche, Nathan Arthur, Neui, NRKirby, obl, ownsourcing dev training, Pod, Prince J, RamenChef, Rick, Roald Nefs, ronnyfm, Sazzad Hissain Khan, Scott Weldon, Sibi Raj, TheDarkKnight, theheadofabroom, ʌolæz æʊʌ qoq, Tot Zam, Tyler Zika, tymspy, Undo, VonC |
| 2     | Afficher l'historique des validations graphiquement avec Gitk | orkoden  |
| 3     | Alias   | AesSedail01, Ajedi32, Andy, Anthony Staunton, Asenar, bstpierre, erewok, eush77, fracz, Gaelan, jrf, jtbandes, madhead, Michael Deardeuff, mickeyandkaka, nus, penguinocoder, riyadhalmur, thanksd, Tom Hale, Wojciech Kazior, zinking   |
| 4     | Analyse des types de workflows                                | Boggin, Configure, Daniel Käfer, Dimitrios Mistriotis, forresthopkinsa, hardmooth, Horen, Kissaki, Majid, Sardathrion, Scott Weldon  |
| 5     | arbre diff  | fybw id  |
| 6     | Archiver  | Dartmouth, forevergenin, Neto Buenrostro, RamenChef  |
| 7     | Bisecting / Finding défectueux commits                        | 4444, Hannoun Yassir, jornh, Kissaki, MrTux, Scott Weldon, Simone Carletti, zebediah49   |
| 8     | Blâmer  | fracz, Matthew Hallatt, nighthawk454, Priyanshu Shekhar, WPrecht   |
| 9     | Changer le nom du dépôt git                                   | xiaoyaoworm  |
| 10    | Configuration   | APerson, Asenar, Cache Staheli, Chris Rasys, e.doroskevic, Julian, Liyan Chang, Majid, Micah Smith, Ortomala Lokni, Peter Mitrano, Priyanshu Shekhar, Scott Weldon, VonC, Wolfgang   |
| 11    | Crochets  | AesSedail01, AnoE, Christiaan Maks, Configure, Eidolon, Flows, fracz, kaartic, lostphilosopher, mwarsco  |
| 12    | Crochets côté client Git                                      | Kelum Senanayake, kiamlaluno   |
| 13    | Cueillette De Cerises   | Atul Khanduri, Braiam, bud-e, dubek, Florian Hämmerle, intboolstring, Julian, kisanme, Lochlan, mpromonet, RedGreenCode  |
| 14    | Des sous-modules  | 321hendrik, Chin Huang, ComicSansMS, foraidt, intboolstring, J   |

|    |  |   |
|----|--|---|
|    |  | <a href="#">F</a> , <a href="#">kowsky</a> , <a href="#">mpromonet</a> , <a href="#">PaladiN</a> , <a href="#">tinlyx</a> , <a href="#">Undo</a> , <a href="#">VonC</a>   |
| 15 | Écrasement   | <a href="#">adarsh</a> , <a href="#">ams</a> , <a href="#">AndiDog</a> , <a href="#">bandi</a> , <a href="#">Braiam</a> , <a href="#">Caleb Brinkman</a> , <a href="#">eush77</a> , <a href="#">georgebrock</a> , <a href="#">jpkrohling</a> , <a href="#">Julian</a> , <a href="#">Mateusz Piotrowski</a> , <a href="#">Ortomala Lokni</a> , <a href="#">RamenChef</a> , <a href="#">Tall Sam</a> , <a href="#">WMios</a>  |
| 16 | En poussant  | <a href="#">AER</a> , <a href="#">Cody Guldner</a> , <a href="#">cringe</a> , <a href="#">frlan</a> , <a href="#">Guillaume</a> , <a href="#">intboolstring</a> , <a href="#">Mário Meyrelles</a> , <a href="#">Marvin</a> , <a href="#">Matt S</a> , <a href="#">MayeulC</a> , <a href="#">pcm</a> , <a href="#">pogosama</a> , <a href="#">Thomas Gerot</a> , <a href="#">Tomás Cañibano</a>  |
| 17 | Fichier .mailmap:<br>Associant<br>contributeur et alias<br>de messagerie | <a href="#">Mario</a> , <a href="#">Michael Plotke</a>  |
| 18 | Fusion   | <a href="#">brentonstrine</a> , <a href="#">Liam Ferris</a> , <a href="#">Noah</a> , <a href="#">penguincoder</a> , <a href="#">Undo</a> , <a href="#">Vogel612</a> , <a href="#">Wolfgang</a>  |
| 19 | Fusion externe et<br>difftools   | <a href="#">AesSedail01</a> , <a href="#">Micha Wiedenmann</a>  |
| 20 | GFS Large File<br>Storage (LFS)  | <a href="#">Alex Stuckey</a> , <a href="#">Matthew Hallatt</a> , <a href="#">shoelzer</a>   |
| 21 | Git Clean  | <a href="#">gnis</a> , <a href="#">MayeulC</a> , <a href="#">n0shadow</a> , <a href="#">pktangyue</a> , <a href="#">Priyanshu Shekhar</a> , <a href="#">Ralf Rafael Frix</a>  |
| 22 | Git Diff   | <a href="#">Aaron Critchley</a> , <a href="#">Abhijeet Kasurde</a> , <a href="#">Adi Lester</a> , <a href="#">anderas</a> , <a href="#">apidae</a> , <a href="#">Brett</a> , <a href="#">Charlie Egan</a> , <a href="#">eush77</a> , <a href="#">000000</a> , <a href="#">intboolstring</a> , <a href="#">Jack Ryan</a> , <a href="#">JakeD</a> , <a href="#">Jakub Narębski</a> , <a href="#">jeffdill12</a> , <a href="#">Joseph K. Strauss</a> , <a href="#">khanmizan</a> , <a href="#">Luke Taylor</a> , <a href="#">Majid</a> , <a href="#">mnoronha</a> , <a href="#">Nathaniel Ford</a> , <a href="#">Ogre Psalm33</a> , <a href="#">orkoden</a> , <a href="#">Ortomala Lokni</a> , <a href="#">penguincoder</a> , <a href="#">pylang</a> , <a href="#">SurDin</a> , <a href="#">Will</a> , <a href="#">ydaetskcoR</a> , <a href="#">Zaz</a>  |
| 23 | git envoyer-email  | <a href="#">Aaron Skomra</a> , <a href="#">Dong Thang</a> , <a href="#">fybw id</a> , <a href="#">Jav_Rock</a> , <a href="#">kofemann</a>   |
| 24 | Git GUI Clients  | <a href="#">Alu</a> , <a href="#">Daniel Käfer</a> , <a href="#">Greg Bray</a> , <a href="#">Nemanja Trifunovic</a> , <a href="#">Pedro Pinheiro</a>  |
| 25 | Git Remote   | <a href="#">AER</a> , <a href="#">ambes</a> , <a href="#">Dániel Kis</a> , <a href="#">Dartmouth</a> , <a href="#">Elizabeth</a> , <a href="#">Jav_Rock</a> , <a href="#">Kalpit</a> , <a href="#">RamenChef</a> , <a href="#">sonali</a> , <a href="#">sunkuet02</a>   |
| 26 | Git rerere   | <a href="#">Isak Combrinck</a>  |
| 27 | git-svn  | <a href="#">Bryan</a> , <a href="#">Randy</a> , <a href="#">Ricardo Amores</a> , <a href="#">RobPethi</a>   |
| 28 | git-tfs  | <a href="#">Boggin</a> , <a href="#">Kissaki</a>  |
| 29 | Historique de<br>réécriture avec<br>filtre-branche                       | <a href="#">gavinbeatty</a> , <a href="#">gavv</a> , <a href="#">Glenn Smith</a>  |
| 30 | Ignorer les fichiers<br>et les dossiers                                  | <a href="#">AER</a> , <a href="#">AesSedail01</a> , <a href="#">agilob</a> , <a href="#">Alex</a> , <a href="#">Amitay Stern</a> , <a href="#">AnimiVulpis</a> , <a href="#">Ates Goral</a> , <a href="#">Aukhan</a> , <a href="#">Avamander</a> , <a href="#">Ben</a> , <a href="#">bpoiss</a> , <a href="#">Braiam</a> , <a href="#">bwegs</a> , <a href="#">Cache Staheli</a> , <a href="#">Collin M</a> , <a href="#">Community</a> , <a href="#">Dartmouth</a> , <a href="#">David Grayson</a> , <a href="#">Devesh Saini</a> , <a href="#">Dheeraj vats</a> , <a href="#">eckes</a> , <a href="#">Ed Cottrell</a> , <a href="#">enrico.bacis</a> , <a href="#">Everettss</a> , <a href="#">Fabio</a> , <a href="#">fracz</a> , <a href="#">Franck Dernoncourt</a> , <a href="#">Fred Barclay</a> , <a href="#">Functino</a> , <a href="#">geek1011</a> , <a href="#">Guillaume Pascal</a> , <a href="#">HerrSerker</a> , <a href="#">intboolstring</a> , <a href="#">Irfan</a> , <a href="#">Jakub Narębski</a> , <a href="#">Jeff Puckett</a> , <a href="#">Jens</a> , <a href="#">joaquinlpereyra</a> , <a href="#">John Slegers</a> , <a href="#">JonasCz</a> , <a href="#">Jörn Hees</a> , <a href="#">joshng</a> , <a href="#">Kačer</a> , <a href="#">Kapep</a> , <a href="#">Kissaki</a> , <a href="#">knut</a> , <a href="#">LeftRight92</a> , <a href="#">Mackattack</a> , <a href="#">Marvin</a> , <a href="#">Matt</a> , <a href="#">MayeulC</a> , <a href="#">Mitch Talmadge</a> , <a href="#">Narayan Acharya</a> , <a href="#">Nathan Arthur</a> , <a href="#">Neui</a> , <a href="#">nouϭAdAzexD</a> |

|    |   |  |
|----|---|--|
|    |   | Nuri Tasdemir, Ortomala Lokni, PaladiN, Panda, pecil, pktangyue, poke, pylang, RhysO, Rick, rokonoid, Sascha, Scott Weldon, Sebastianb, SeeuDl, sjas, Slayther, SnoringFrog, spikeheap, theJollySin, Toby, ʌolɛɛz əʊʔ qoq, Tom Gijselinck, Tomasz Bak, Vi., Victor Schröder, VonC, Wilfred Hughes, Wolfgang, ydaetskcoR, Yosvel Quintero, Yury Fedorov, Zaz, Zeeker  |
| 31 | Internes  | nighthawk454   |
| 32 | Liasses   | jwd630   |
| 33 | Liste de révocation                               | mkasberg   |
| 34 | Marquage Git                                      | Atul Khanduri, demonplus, TheDarkKnight  |
| 35 | Mettre à jour le nom de l'objet dans la référence | Keyur Ramoliya, RamenChef  |
| 36 | Migration vers Git                                | AesSedai101, Boggin, Configure, Guillaume Pascal, Indregard, Rick, TheDarkKnight   |
| 37 | Mise en scène                                     | AesSedai101, Andy Hayden, Asaph, Configure, intboolstring, Jakub Narębski, jkdev, Muhammad Abdullah, Nathan Arthur, ownsourcing dev training, Richard Dally, Wolfgang  |
| 38 | Montrer   | Zaz  |
| 39 | Nom de la branche Git sur Bash Ubuntu             | Manishh  |
| 40 | Parcourir l'historique                            | Ahmed Metwally, Andy Hayden, Aratz, Atif Hussain, Boggin, Brett, Configure, davidcondrey, Fabio, Flows, fracz, Fred Barclay, guleria, intboolstring, janos, jaredr, Kamiccolo, Kraigh, LeGEC, manasouza, Matt Clark, Matthew Hallatt, MByD, mpromonet, Muhammad Abdullah, Noah, Oleander, Pedro Pinheiro, RedGreenCode, Toby Allen, Vogel612, ydaetskcoR   |
| 41 | Patch Git   | Dartmouth, Liju Thomas   |
| 42 | Perte   | Adi Lester, AesSedai101, Alexander Bird, Andy Hayden, Boggin, brentonstrine, Brian, Colin D Bennett, ericdwang, Karan Desai, Matthew Hallatt, Nathan Arthur, Nathaniel Ford, Nithin K Anil, Pace, Rick, textshell, Undo, Zaz   |
| 43 | Ramification                                      | Amitay Stern, Andrew Kay, AnimiVulpis, Bad, BobTuckerman, Community, dan, Daniel Käfer, Daniel Stradowski, Deepak Bansal, djb, Don Kirkby, Duncan X Simpson, Eric Bouchut, forevergenin, fracz, Franck Dernoncourt, Fred Barclay, Frodon, gavv, Irfan, james large, janos, Jason, Joel Cornett, Jon Schneider, Jonathan, Joseph Dasenbrock, jrf, kartik, KartikKannapur, khanmizan, kirrmann, kisanme, Majid, Martin, MayeulC, Michael Richardson, Mihai, Mitch Talmadge, mkasberg, nepda, Noah, Noushad PP, Nowhere man, olegtaranenko, Ortomala Lokni, Ozair Kafray, PaladiN, ΠΑΝΥΠΤΙΣ, Priyanshu Shekhar, Ralf Rafael Frix, Richard Hamilton, Robin, RudolphEst, Siavas, Simone Carletti, the12, Uwe, Vlad, wintersolider, Wojciech Kazior, Wolfgang, Yerko Palma, Yury Fedorov, zygimantus |
| 44 | Rangement de votre référentiel local et distant   | Thomas Crowley   |

|    |  |   |
|----|--|---|
| 45 | Rebasing   | <a href="#">AER</a> , <a href="#">Alexander Bird</a> , <a href="#">anderas</a> , <a href="#">Ashwin Ramaswami</a> , <a href="#">Braiam</a> , <a href="#">BusyAnt</a> , <a href="#">Configure</a> , <a href="#">Daniel Käfer</a> , <a href="#">Derek Liu</a> , <a href="#">Dunno</a> , <a href="#">e.doroskevic</a> , <a href="#">Enrico Campidoglio</a> , <a href="#">eskwayrd</a> , <a href="#">000000</a> , <a href="#">Hugo Ferreira</a> , <a href="#">intboolstring</a> , <a href="#">Jeffrey Lin</a> , <a href="#">Joel Cornett</a> , <a href="#">Joseph K. Strauss</a> , <a href="#">jtbandes</a> , <a href="#">Julian</a> , <a href="#">Kissaki</a> , <a href="#">LeGEC</a> , <a href="#">Libin Varghese</a> , <a href="#">Luca Putzu</a> , <a href="#">lucash</a> , <a href="#">madhukar93</a> , <a href="#">Majid</a> , <a href="#">Matt</a> , <a href="#">Matthew Hallatt</a> , <a href="#">Menasheh</a> , <a href="#">Michael Mrozek</a> , <a href="#">Nemanja Boric</a> , <a href="#">Ortomala Lokni</a> , <a href="#">Peter Mitrano</a> , <a href="#">pylang</a> , <a href="#">Richard</a> , <a href="#">takteek</a> , <a href="#">Travis</a> , <a href="#">Victor Schröder</a> , <a href="#">VonC</a> , <a href="#">Wasabi Fan</a> , <a href="#">yarons</a> , <a href="#">Zaz</a>   |
| 46 | Récupérer  | <a href="#">Creative John</a> , <a href="#">Hardik Kanjariya</a> ♪, <a href="#">Julie David</a> , <a href="#">kisanme</a> , <a href="#">0ANAY0</a> <a href="#">ms</a> , <a href="#">Scott Weldon</a> , <a href="#">strangeqargo</a> , <a href="#">Zaz</a>   |
| 47 | Référentiels de clonage  | <a href="#">AER</a> , <a href="#">Andrea Romagnoli</a> , <a href="#">Andy Hayden</a> , <a href="#">Blundering Philosopher</a> , <a href="#">Dartmouth</a> , <a href="#">Ezra Free</a> , <a href="#">ganesshkumar</a> , <a href="#">000000</a> , <a href="#">kartik</a> , <a href="#">KartikKannapur</a> , <a href="#">mnoronha</a> , <a href="#">Peter Mitrano</a> , <a href="#">pkowalczyk</a> , <a href="#">Rick</a> , <a href="#">Undo</a> , <a href="#">Wojciech Kazior</a>   |
| 48 | Reflog - Restauration des commits non affichés dans le journal git | <a href="#">Braiam</a> , <a href="#">Peter Amidon</a> , <a href="#">Scott Weldon</a>  |
| 49 | Renommer   | <a href="#">bud-e</a> , <a href="#">Karan Desai</a> , <a href="#">P.J.Meisch</a> , <a href="#">PhotometricStereo</a>  |
| 50 | Répertoires vides dans Git   | <a href="#">Ates Goral</a>  |
| 51 | Résoudre les conflits de fusion                                    | <a href="#">Braiam</a> , <a href="#">Dartmouth</a> , <a href="#">David Ben Knoble</a> , <a href="#">Fabio</a> , <a href="#">nus</a> , <a href="#">Vivin George</a> , <a href="#">Yury Fedorov</a>   |
| 52 | S'engager  | <a href="#">Aaron Critchley</a> , <a href="#">AER</a> , <a href="#">Alan</a> , <a href="#">Allan Burleson</a> , <a href="#">Amitay Stern</a> , <a href="#">Andrew Sklyarevsky</a> , <a href="#">Andy Hayden</a> , <a href="#">Anonymous Entity</a> , <a href="#">APerson</a> , <a href="#">bandi</a> , <a href="#">Cache Staheli</a> , <a href="#">Chris Forrence</a> , <a href="#">Cody Guldner</a> , <a href="#">cormacrelf</a> , <a href="#">davidcondrey</a> , <a href="#">Deep</a> , <a href="#">depperm</a> , <a href="#">ericdwang</a> , <a href="#">Ethunxxx</a> , <a href="#">Fred Barclay</a> , <a href="#">George Brighton</a> , <a href="#">Igor Ivancha</a> , <a href="#">intboolstring</a> , <a href="#">JacobLeach</a> , <a href="#">James Taylor</a> , <a href="#">janos</a> , <a href="#">joeytwiddle</a> , <a href="#">Jordan Knott</a> , <a href="#">KartikKannapur</a> , <a href="#">kisanme</a> , <a href="#">Majid</a> , <a href="#">Matt Clark</a> , <a href="#">Matthew Hallatt</a> , <a href="#">MayeulC</a> , <a href="#">Micah Smith</a> , <a href="#">Pod</a> , <a href="#">Rick</a> , <a href="#">Scott Weldon</a> , <a href="#">SommerEngineering</a> , <a href="#">Sonny Kim</a> , <a href="#">Thomas Gerot</a> , <a href="#">Undo</a> , <a href="#">user1990366</a> , <a href="#">vguzmanp</a> , <a href="#">Vladimir F</a> , <a href="#">Zaz</a> |
| 53 | Se cacher  | <a href="#">aavrug</a> , <a href="#">AesSedail01</a> , <a href="#">Asaph</a> , <a href="#">Brian Hinchey</a> , <a href="#">bud-e</a> , <a href="#">Cache Staheli</a> , <a href="#">Deep</a> , <a href="#">e.doroskevic</a> , <a href="#">fracz</a> , <a href="#">GingerPlusPlus</a> , <a href="#">Guillaume</a> , <a href="#">inkista</a> , <a href="#">Jakub Narębski</a> , <a href="#">Jarede</a> , <a href="#">jeffdill12</a> , <a href="#">joeytwiddle</a> , <a href="#">Julie David</a> , <a href="#">Kara</a> , <a href="#">Koraktor</a> , <a href="#">Majid</a> , <a href="#">manasouza</a> , <a href="#">Ortomala Lokni</a> , <a href="#">Patrick</a> , <a href="#">Peter Mitrano</a> , <a href="#">Ralf Rafael Frix</a> , <a href="#">Sebastianb</a> , <a href="#">Tomás Cañibano</a> , <a href="#">Wojciech Kazior</a>  |
| 54 | Sous-arbres  | <a href="#">4444</a> , <a href="#">Jeff Puckett</a>   |
| 55 | Statistiques Git   | <a href="#">Dartmouth</a> , <a href="#">Farhad Faghihi</a> , <a href="#">Hugo Buff</a> , <a href="#">KartikKannapur</a> , <a href="#">lxxr</a> , <a href="#">penguincoder</a> , <a href="#">RamenChef</a> , <a href="#">SashaZd</a> , <a href="#">Tyler Hyndman</a> , <a href="#">vkluge</a>  |
| 56 | Syntaxe des révisions Git  | <a href="#">Dartmouth</a> , <a href="#">Jakub Narębski</a>  |
| 57 | Tirant   | <a href="#">Kissaki</a> , <a href="#">MayeulC</a> , <a href="#">mpromonet</a> , <a href="#">rene</a> , <a href="#">Ryan</a> , <a href="#">Scott Weldon</a> , <a href="#">Shog9</a> , <a href="#">Stony</a> , <a href="#">Thamilan</a> , <a href="#">Thomas Gerot</a> , <a href="#">Zaz</a>  |
| 58 | TortoiseGit  | <a href="#">Julian</a> , <a href="#">Matas Vaitkevicius</a>   |
| 59 | Travailler avec des  | <a href="#">Boggin</a> , <a href="#">Caleb Brinkman</a> , <a href="#">forevergenin</a> , <a href="#">heitortsergent</a> ,   |

|    |                                    |   |
|----|------------------------------------|---|
|    | télécommandes                      | <a href="#">intboolstring</a> , <a href="#">jeffdill12</a> , <a href="#">Julie David</a> , <a href="#">Kalpit</a> , <a href="#">Matt Clark</a> , <a href="#">MByD</a> , <a href="#">mnoronha</a> , <a href="#">mpromonet</a> , <a href="#">mystarocks</a> , <a href="#">Pascalz</a> , <a href="#">Raghav</a> , <a href="#">Ralf Rafael Frix</a> , <a href="#">Salah Eddine Lahniche</a> , <a href="#">Sam</a> , <a href="#">Scott Weldon</a> , <a href="#">Stony</a> , <a href="#">Thamilan</a> , <a href="#">Vivin George</a> , <a href="#">VonC</a> , <a href="#">Zaz</a> |
| 60 | Utiliser un fichier .gitattributes | <a href="#">Chin Huang</a> , <a href="#">dahlbyk</a> , <a href="#">Toby</a>   |
| 61 | Worktrees                          | <a href="#">andipla</a> , <a href="#">Configure</a> , <a href="#">Victor Schröder</a>   |