THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GITT

Scott Chacon and Ben Straub

Apress®

Ce travail est sous licence Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. Pour voir une copie de cette licence, visitez ***http://creativecommons.org/licenses/by-nc-sa/3.0/*** ou envoyez une lettre à Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Introduction

Bienvenue à la seconde édition de Pro Git. La première édition a été publiée depuis plus de quatre ans maintenant. Depuis lors, beaucoup de choses ont changé et beaucoup de choses importantes non. Bien que la plupart des commandes et des concepts clés sont encore valables aujourd'hui vu que l'équipe du cœur de Git est assez fantastique pour garder la compatibilité ascendante, il y a eu quelques ajouts significatifs et des changements dans la communauté qui entoure Git. La seconde édition de ce livre est faite pour répondre à ces changements et mettre à jour le livre afin qu'il soit plus utile au nouvel utilisateur.

Quand j'ai écrit la première édition, Git était encore un outil relativement difficile à utiliser et n'avait pas percé chez les développeurs purs et durs. Il a commencé à gagner de la popularité dans certaines communautés, mais n'avait atteint nulle part l'ubiquité qu'il a aujourd'hui. Depuis, presque toutes les communautés open source l'ont adopté. Git a fait des progrès incroyables sur Windows, dans la multiplication des interfaces utilisateur graphiques sur toutes les plateformes, dans le support IDE et dans l'utilisation commerciale. Pro Git ne connaît rien de tout cela. Un des objectifs principaux de cette nouvelle édition est d'aborder toutes ces nouvelles frontières au sein de la communauté Git.

La communauté Open Source utilisant Git a elle aussi massivement augmenté. Quand je me suis assis pour écrire pour la première fois le livre il y a presque cinq ans de cela (ça m'a pris du temps pour sortir la première version), je venais juste de commencer à travailler dans une entreprise peu connue développant un site web hébergeant Git appelée GitHub. Au moment de la publication, il y avait peut-être quelques milliers de gens utilisant le site et que quatre d'entre nous travaillant dessus. Pendant que j'écris cette introduction, GitHub est en train d'annoncer son dix millionième projet hébergé, avec presque cinq millions de comptes développeur enregistrés et plus de deux-cent trente employés. Que vous l'aimiez ou que vous le détestiez, GitHub a grandement affecté une grande partie de la communauté Open Source d'une façon difficilement envisageable lorsque j'ai écrit la première édition.

J'ai écrit une petite section dans la version originale de Pro Git sur GitHub comme exemple de Git hébergé dont je n'ai jamais été très fier. Je n'ai pas

beaucoup aimé écrire sur ce que je considérais comme étant essentiellement une ressource communautaire et aussi de parler de mon entreprise. Bien que je n'aime toujours pas ce conflit d'intérêts, l'importance de GitHub dans la communauté Git est inévitable. Au lieu d'un exemple d'hébergement Git, j'ai décidé de transformer cette partie du livre en décrivant plus en détail ce que GitHub est et comment l'utiliser efficacement. Si vous êtes sur le point d'apprendre à utiliser Git, alors savoir utiliser GitHub vous aidera à prendre part à une immense communauté, ce qui est un atout, peu importe quel hébergement Git vous déciderez d'utiliser pour votre propre code.

L'autre grand changement depuis la dernière publication a été le développement et l'expansion du protocole HTTP pour les transactions Git de réseau. La plupart des exemples dans le livre ont été changé en HTTP depuis SSH parce que c'est beaucoup plus simple.

Ça a été stupéfiant de voir Git grandir au cours des dernières années en partant d'un système de contrôle de version relativement obscur jusqu'à dominer complètement le contrôle de version commercial et open source. Je suis très content que Pro Git ait aussi bien marché et qu'il ait été un des rares livres techniques du marché qui soit à la fois assez réussi et complètement open source.

J'espère que vous apprécierez cette édition mise à jour de Pro Git.

Table of Contents

Introduction **iii**

CHAPTER 1: Démarrage rapide **17**

À propos de la gestion de version 17

Les systèmes de gestion de version locaux 18

Les systèmes de gestion de version centralisés 19

Les systèmes de gestion de version distribués 20

Une rapide histoire de Git 22

Rudiments de Git 22

Des instantanés, pas des différences 23

Presque toutes les opérations sont locales 24

Git gère l'intégrité 25

Généralement, Git ne fait qu'ajouter des données 25

Les trois états 25

La ligne de commande 27

Installation de Git 27

Installation sur Linux 28

Installation sur Mac 28

Installation sur Windows 29

Installation depuis les sources 30

Paramétrage à la première utilisation de Git 30

Votre identité 31
Votre éditeur de texte 32
Vérifier vos paramètres 32
Obtenir de l'aide 33
Résumé 33

CHAPTER 2: Les bases de Git 35
Démarrer un dépôt Git 35
Initialisation d'un dépôt Git dans un répertoire existant 35
Cloner un dépôt existant 36
Enregistrer des modifications dans le dépôt 37
Vérifier l'état des fichiers 37
Placer de nouveaux fichiers sous suivi de version 38
Indexer des fichiers modifiés 39
Statut court 41
Ignorer des fichiers 42
Inspecter les modifications indexées et non indexées 43
Valider vos modifications 46
Passer l'étape de mise en index 48
Effacer des fichiers 48
Déplacer des fichiers 50
Visualiser l'historique des validations 51
Limiter la longueur de l'historique 56
Annuler des actions 58
Désindexer un fichier déjà indexé 59
Réinitialiser un fichier modifié 61
Travailler avec des dépôts distants 62
Afficher les dépôts distants 62
Ajouter des dépôts distants 63
Récupérer et tirer depuis des dépôts distants 64
Pousser son travail sur un dépôt distant 65

Inspecter un dépôt distant 65
Retirer et déplacer des branches distantes 67
Étiquetage 67
Lister vos étiquettes 67
Créer des étiquettes 68
Les étiquettes annotées 68
Les étiquettes légères 69
Étiqueter après coup 70
Partager les étiquettes 71
Extraire une étiquette 72
Les alias Git 72
Résumé 73

CHAPTER 3: Les branches avec Git 75
Les branches en bref 75
Créer une nouvelle branche 78
Basculer entre les branches 79
Branches et fusions : les bases 83
Branches 83
Fusions (*Merges*) 88
Conflits de fusions (*Merge conflicts*) 90
Gestion des branches 93
Travailler avec les branches 95
Branches au long cours 95
Les branches thématiques 96
Branches distantes 99
Pousser les branches 104
Suivre les branches 106
Tirer une branche (*Pulling*) 108
Suppression de branches distantes 109
Rebaser (*Rebasing*) 109

Les bases 109
Rebases plus intéressants 112
Les dangers du rebasage 115
Rebaser quand vous rebasez 118
Rebaser ou Fusionner 120
Résumé 121

CHAPTER 4: Git sur le serveur 123
Protocoles 124
Protocole local 124
Protocoles sur HTTP 125
Protocole SSH 128
Protocole Git 129
Installation de Git sur un serveur 130
Copie du dépôt nu sur un serveur 131
Petites installations 132
Génération des clés publiques SSH 133
Mise en place du serveur 134
Démon (*Daemon*) Git 137
HTTP intelligent 138
GitWeb 140
GitLab 143
Installation 143
Administration 144
Usage de base 147
Coopérer 147
Git hébergé 148
Résumé 149

CHAPTER 5: Git distribué 151
Développements distribués 151

Gestion Centralisée 151
Mode du gestionnaire d'intégration 153
Mode dictateur et ses lieutenants 154
Résumé 155
Contribution à un projet 155
Guides pour une validation 156
Cas d'une petite équipe privée 158
Équipe privée importante 165
Projet public dupliqué 171
Projet public via courriel 175
Résumé 179
Maintenance d'un projet 179
Travail dans des branches thématiques 180
Application des patchs à partir de courriel 180
Vérification des branches distantes 184
Déterminer les modifications introduites 185
Intégration des contributions 187
Étiquetage de vos publications 194
Génération d'un nom de révision 195
Préparation d'une publication 196
Shortlog 196
Résumé 197

CHAPTER 6: GitHub 199
Configuration et paramétrage d'un compte 199
Accès par SSH 201
Votre Avatar 202
Vos adresses électroniques 203
Authentification à deux facteurs 204
Contribution à un projet 205
Duplication des projets 205

Processus GitHub 206
Requêtes de tirage avancées 214
Markdown 220
Maintenance d'un projet 226
Création d'un nouveau dépôt 226
Ajout de collaborateurs 228
Gestion des requêtes de tirage 229
Mentions et notifications 235
Fichiers spéciaux 239
README 239
CONTRIBUTING 240
Administration du projet 241
Gestion d'un regroupement 242
Les bases d'un regroupement 242
Équipes 243
Journal d'audit 245
Écriture de scripts pour GitHub 246
Crochets (*Hooks*) 247
L'interface de programmation (*API*) GitHub 251
Utilisation Basique 252
Commenter un problème 253
Changer le statut d'une requête de tirage 255
Octokit 257
Résumé 257

CHAPTER 7: Utilitaires Git 259
Sélection des versions 259
Révisions ponctuelles 259
Empreinte SHA courte 259
Références de branches 261
Raccourcis RefLog 262

Références ancêtres 263
Plages de *commits* 265
Indexation interactive 268
Indexation et désindexation des fichiers 269
Indexations partielles 271
Remisage et nettoyage 273
Remiser votre travail 273
Remisage créatif 275
Défaire l'effet d'une remise 277
Créer une branche depuis une remise 277
Nettoyer son répertoire de travail 278
Signer votre travail 280
Introduction à GPG 280
Signer des étiquettes 281
Verifier des étiquettes 281
Signer des *commits* 282
Tout le monde doit signer 284
Recherche 284
Git grep 284
Recherche dans le journal Git 286
Réécrire l'historique 288
Modifier la dernière validation 288
Modifier plusieurs messages de validation 289
Réordonner les *commits* 292
Écraser un *commit* 292
Diviser un *commit* 293
L'option nucléaire : `filter-branch` 295
Reset démystifié 297
Les trois arbres 297
Le flux de travail 299

Le rôle de reset 305
Reset avec un chemin 310
Écraser les *commits* 313
Et checkout 316
Résumé 318
Fusion avancée 319
Conflits de fusion 320
Défaire des fusions 331
Autres types de fusions 335
Rerere 340
Déboguer avec Git 347
Fichier annoté 347
Recherche dichotomique 349
Sous-modules 351
Démarrer un sous-module 351
Cloner un projet avec des sous-modules 354
Travailler sur un projet comprenant des sous-modules 355
Trucs et astuces pour les sous-modules 367
Les problèmes avec les sous-modules 369
Empaquetage (*bundling*) 371
Replace 376
Stockage des identifiants 384
Sous le capot 385
Un cache d'identifiants personnalisé 388
Résumé 390

CHAPTER 8: Personnalisation de Git 391
Configuration de Git 391
Configuration de base d'un client 392
Couleurs dans Git 395
Outils externes de fusion et de différence 397

Formatage et espaces blancs 400
Configuration du serveur 403
Attributs Git 404
Fichiers binaires 404
Expansion des mots-clés 408
Export d'un dépôt 411
Stratégies de fusion 413
Crochets Git 414
Installation d'un crochet 414
Crochets côté client 414
Crochets côté serveur 417
Exemple de politique gérée par Git 418
Crochet côté serveur 418
Crochets côté client 425
Résumé 428

CHAPTER 9: Git et les autres systèmes 429
Git comme client 429
Git et Subversion 429
Git et Mercurial 442
Git et Perforce 451
Git et TFS 467
Migration vers Git 478
Subversion 478
Mercurial 481
Perforce 483
TFS 486
Un importateur personnalisé 487
Résumé 495

CHAPTER 10: Les tripes de Git 497

Plomberie et porcelaine 497
Les objets de Git 499
Les objets arbres 501
Les objets *commit* 505
Stockage des objets 508
Références Git 510
La branche HEAD 511
Étiquettes 513
Références distantes 514
Fichiers groupés 515
La *refspec* 518
Pousser des *refspecs* 520
Supprimer des références 521
Les protocoles de transfert 521
Le protocole stupide 521
Le protocole intelligent 524
Résumé sur les protocoles 527
Maintenance et récupération de données 528
Maintenance 528
Récupération de données 529
Suppression d'objets 532
Les variables d'environnement 536
Comportement général 536
Les emplacements du dépôt 537
Pathspecs 538
Création de *commits* 538
Travail sur le réseau 539
Visualisation des différences et Fusion 539
Débogage 540
Divers 542

Résumé 542

Git dans d'autres environnements 545

Embarquer Git dans vos applications 561

Commandes Git 573

Index 591

Démarrage rapide 1

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

À propos de la gestion de version

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

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

Les systèmes de gestion de version locaux

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

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

FIGURE 1-1

Gestion de version locale.

Un des systèmes les plus populaires était RCS, qui est encore distribué avec de nombreux systèmes d'exploitation aujourd'hui. Même le système d'exploitation populaire Mac OS X inclut le programme `rcs` lorsqu'on installe les outils de développement logiciel. Cet outil fonctionne en conservant des ensembles de patchs (c'est-à-dire la différence entre les fichiers) d'une version à l'autre dans

un format spécial sur disque ; il peut alors restituer l'état de n'importe quel fichier à n'importe quel instant en ajoutant toutes les différences.

Les systèmes de gestion de version centralisés

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

FIGURE 1-2

Gestion de version centralisée.

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

Cependant ce système a aussi de nombreux défauts. Le plus visible est le point unique de panne que le serveur centralisé représente. Si ce serveur est en

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

Les systèmes de gestion de version distribués

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

FIGURE 1-3

Gestion de version distribuée.

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

Une rapide histoire de Git

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

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

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

Depuis sa naissance en 2005, Git a évolué et mûri pour être facile à utiliser tout en conservant ses qualités initiales. Il est incroyablement rapide, il est très efficace pour de grands projets et il a un incroyable système de branches pour des développements non linéaires (voir **Chapter 3**).

Rudiments de Git

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

Des instantanés, pas des différences

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

FIGURE 1-4

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

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

FIGURE 1-5

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

C'est une distinction importante entre Git et quasiment tous les autres VCS. Git a reconsidéré quasiment tous les aspects de la gestion de version que la plupart des autres systèmes ont copiés des générations précédentes. Git ressemble beaucoup plus à un mini système de fichiers avec des outils incroyablement puissants construits dessus, plutôt qu'à un simple VCS. Nous explorerons les bénéfices qu'il y a à penser les données de cette manière quand nous aborderons la gestion de branches dans **Chapter 3**.

Presque toutes les opérations sont locales

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

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

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

Git gère l'intégrité

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

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

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

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

Généralement, Git ne fait qu'ajouter des données

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

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

Les trois états

Un peu de concentration maintenant. Il est primordial de se souvenir de ce qui suit si vous souhaitez que le reste de votre apprentissage s'effectue sans difficulté. Git gère trois états dans lesquels les fichiers peuvent résider : validé,

modifié et indexé. Validé signifie que les données sont stockées en sécurité dans votre base de données locale. Modifié signifie que vous avez modifié le fichier mais qu'il n'a pas encore été validé en base. Indexé signifie que vous avez marqué un fichier modifié dans sa version actuelle pour qu'il fasse partie du prochain instantané du projet.

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

FIGURE 1-6

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

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

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

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

L'utilisation standard de Git se passe comme suit :

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

Si une version particulière d'un fichier est dans le répertoire Git, il est considéré comme validé. S'il est modifié mais a été ajouté dans la zone d'index, il est indexé. S'il a été modifié depuis le dernier instantané mais n'a pas été indexé, il est modifié. Dans **Chapter 2**, vous en apprendrez plus sur ces états et comment vous pouvez en tirer parti ou complètement occulter la phase d'indexation.

La ligne de commande

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

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

Installation de Git

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

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

Installation sur Linux

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

```
$ yum install git
```

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

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

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

Installation sur Mac

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

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

FIGURE 1-7

Installateur de Git pour OS X.

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

Installation sur Windows

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

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

Installation depuis les sources

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

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

```
$ 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
```

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

Puis, compiler et installer :

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

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

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

Paramétrage à la première utilisation de Git

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

Git contient un outil appelé `git config` pour vous permettre de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'ap-

parence et du comportement de Git. Ces variables peuvent être stockées dans trois endroits différents :

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

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

Sur les systèmes Windows, Git recherche le fichier `.gitconfig` dans le répertoire `$HOME` (`%USERPROFILE%` dans l'environnement natif de Windows) qui est `C:\Documents and Settings\$USER` ou `C:\Users\$USER` la plupart du temps, selon la version (`$USER` devient `%USERNAME%` dans l'environnement de Windows). Il recherche tout de même `/etc/gitconfig`, bien qu'il soit relatif à la racine MSys, qui se trouve où vous aurez décidé d'installer Git sur votre système Windows.

Votre identité

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

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

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

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

Votre éditeur de texte

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

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

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

Vérifier vos paramètres

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

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

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

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

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

Obtenir de l'aide

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

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

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

```
$ git help config
```

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

Résumé

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

Les bases de Git 2

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

Démarrer un dépôt Git

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

Initialisation d'un dépôt Git dans un répertoire existant

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

```
$ git init
```

Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt — un squelette de dépôt Git. Pour l'instant, aucun fichier n'est encore versionné. (Cf. **Chapter 10** pour plus d'information sur les fichiers contenus dans le répertoire `.git` que vous venez de créer.)

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

Cloner un dépôt existant

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

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

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

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

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

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

Git dispose de différents protocoles de transfert que vous pouvez utiliser. L'exemple précédent utilise le protocole `https://`, mais vous pouvez aussi voir `git://` ou `utilisateur@serveur:/chemin.git`, qui utilise le protocole de transfert SSH. **"Installation de Git sur un serveur"** introduit toutes les options

disponibles pour mettre en place un serveur Git, ainsi que leurs avantages et inconvénients.

Enregistrer des modifications dans le dépôt

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

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

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

FIGURE 2-1

Le cycle de vie des états des fichiers.

Vérifier l'état des fichiers

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

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

Ce message signifie que votre copie de travail est propre, en d'autres termes, aucun fichier suivi n'a été modifié. Git ne voit pas non plus de fichiers non-suivis, sinon ils seraient listés ici. Enfin, la commande vous indique sur quelle branche vous êtes. Pour l'instant, c'est toujours "master", qui correspond à la valeur par défaut ; nous ne nous en soucierons pas maintenant. Dans **Chapter 3**, nous parlerons plus en détail des branches et des références.

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

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

        LISEZMOI

aucune modification ajoutée à la validation mais des fichiers non suivis sont prés
```

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

Placer de nouveaux fichiers sous suivi de version

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

```
$ git add LISEZMOI
```

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

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

        nouveau fichier : LISEZMOI
```

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

Indexer des fichiers modifiés

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

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

        nouveau fichier : LISEZMOI

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

```
        modifié :         CONTRIBUTING.md
```

Le fichier `CONTRIBUTING.md` apparaît sous la section nommée « Modifications qui ne seront pas validées » ce qui signifie que le fichier sous suivi de version a été modifié dans la copie de travail mais n'est pas encore indexé. Pour l'indexer, il faut lancer la commande `git add`. `git add` est une commande multi-usage — elle peut être utilisée pour placer un fichier sous suivi de version, pour indexer un fichier ou pour d'autres actions telles que marquer comme résolus des conflits de fusion de fichiers. Sa signification s'approche plus de « ajouter ce contenu pour la prochaine validation » que de « ajouter ce contenu au projet ». Lançons maintenant `git add` pour indexer le fichier `CONTRIBUTING.md`, et relançons la commande `git status` :

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

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

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

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

        nouveau fichier : LISEZMOI
        modifié :         CONTRIBUTING.md

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

```
        modifié :         CONTRIBUTING.md
```

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

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

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

Statut court

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

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

Les nouveaux fichiers qui ne sont pas suivis sont précédés de `??`, les fichiers nouveaux et indexés sont précédés de `A`, les fichiers modifiés de `M` et ainsi de suite. Il y a deux colonnes d'état - la gauche indique l'état de l'index et la droite l'état du dossier de travail. Donc l'exemple ci-dessus indique que le fichier `README` est modifié dans le répertoire de travail mais n'est pas encore indexé, tan-

dis que le fichier `lib/simplegit.rb` est modifié et indexé. Le fichier `Rakefile` a été modifié, indexé puis modifié à nouveau, de sorte qu'il a des modifications à la fois indexées et non-indexées.

Ignorer des fichiers

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

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

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

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

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

Les patrons standards de fichiers sont des expressions régulières simplifiées utilisées par les shells. Un astérisque (*) correspond à un ou plusieurs caractères ; `[abc]` correspond à un des trois caractères listés dans les crochets, donc

a ou b ou c ; un point d'interrogation (?) correspond à un unique caractère ; des crochets entourant des caractères séparés par un tiret (`[0-9]`) correspond à un caractère dans l'intervalle des deux caractères indiqués, donc ici de 0 à 9. Vous pouvez aussi utiliser deux astérisques pour indiquer une série de répertoires inclus ; `a/**/z` correspond donc à `a/z`, `a/b/z`, `a/b/c/z` et ainsi de suite.

Voici un autre exemple de fichier `.gitignore` :

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

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

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

# ignorer tous les fichiers dans le répertoire build
build/

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

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

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

Inspecter les modifications indexées et non indexées

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

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

        nouveau fichier : LISEZMOI

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

        modifié :         CONTRIBUTING.md
```

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

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

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

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

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

```
$ git diff --staged
diff --git a/LISEZMOI b/LISEZMOI
new file mode 100644
```

```
index 0000000..1e17b0c
--- /dev/null
+++ b/LISEZMOI
@@ -0,0 +1 @@
+Mon Projet
```

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

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

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

        nouveau fichier : CONTRIBUTING.md

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

        modifié :         CONTRIBUTING.md
```

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

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

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

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

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

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

GIT DIFF DANS UN OUTIL EXTERNE

Nous allons continuer à utiliser la commande `git diff` de différentes manières par la suite. Il existe une autre manière de visualiser les différences si vous préférez un outil graphique ou externe. Si vous lancez `git difftool` au lieu de `git diff`, vous pourrez visualiser les différences grâce à une application telle que Araxis, emerge, vimdiff ou autre. Lancez `git difftool --tool-help` pour connaître les applications disponibles sur votre système.

Valider vos modifications

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

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

```
$ git commit
```

Cette action lance votre éditeur par défaut (qui est paramétré par la variable d'environnement `$EDITOR` de votre shell — habituellement vim ou Emacs, mais vous pouvez le paramétrer spécifiquement pour Git en utilisant la commande `git config --global core.editor` comme nous l'avons vu au **Chapter 1**).

L'éditeur affiche le texte suivant :

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

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

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

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

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

Souvenez-vous que la validation enregistre l'instantané que vous avez préparé dans la zone d'index. Tout ce que vous n'avez pas indexé est toujours en état modifié ; vous pouvez réaliser une nouvelle validation pour l'ajouter à l'historique. À chaque validation, vous enregistrez un instantané du projet en forme

de jalon auquel vous pourrez revenir ou avec lequel comparer votre travail ultérieur.

Passer l'étape de mise en index

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

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

        modifié :         CONTRIBUTING.md

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

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

Effacer des fichiers

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

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

```
$ rm PROJECTS.md
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui ne seront pas validées :
  (utilisez "git add/rm <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de t

        supprimé :        PROJECTS.md

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

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

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

        supprimé :        PROJECTS.md
```

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

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

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

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

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

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

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

Cette commande élimine tous les fichiers se terminant par ~.

Déplacer des fichiers

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

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

```
$ git mv nom_origine nom_cible
```

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

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

        renommé :         LISEZMOI.txt -> LISEZMOI
```

Néanmoins, cela revient à lancer les commandes suivantes :

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

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

Visualiser l'historique des validations

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

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

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

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

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

    changed the version number

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

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
```

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

    first commit
```

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

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

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

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

    changed the version number

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

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

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
```

```
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

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

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

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

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

    changed the version number

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

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

    removed unnecessary test

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

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

    first commit

 README           |  6 ++++++
```

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

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

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

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

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

Table 2-1 liste les options de formatage les plus utiles.

TABLE 2-1. *Options utiles pour `git log --pretty=format`*

Option	Description du formatage
%H	Somme de contrôle du commit
%h	Somme de contrôle abrégée du commit
%T	Somme de contrôle de l'arborescence

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

Vous pourriez vous demander quelle est la différence entre *auteur* et *validateur*. L'*auteur* est la personne qui a réalisé initialement le travail, alors que le *validateur* est la personne qui a effectivement validé ce travail en gestion de version. Donc, si quelqu'un envoie un patch à un projet et un des membres du projet l'applique, les deux personnes reçoivent le crédit — l'écrivain en tant qu'auteur, et le membre du projet en tant que validateur. Nous traiterons plus avant de cette distinction dans le **Chapter 5**.

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

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

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

Les options ci-dessus ne sont que des options simples de format de sortie de `git log` — il y en a de nombreuses autres. **Table 2-2** donne une liste des options que nous avons traitées ainsi que d'autres options communément utilisées accompagnées de la manière dont elles modifient le résultat de la commande `log`.

TABLE 2-2. *Options usuelles de `git log`*

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

Limiter la longueur de l'historique

En complément des options de formatage de sortie, `git log` est pourvu de certaines options de limitation utiles — des options qui permettent de restreindre la liste à un sous-ensemble de *commits*. Vous avez déjà vu une de ces options — l'option `-2` qui ne montre que les deux derniers *commits*. En fait, on

peut utiliser `-<n>`, où `n` correspond au nombre de *commits* que l'on cherche à visualiser en partant des plus récents. En vérité, il est peu probable que vous utilisiez cette option, parce que Git injecte par défaut sa sortie dans un outil de pagination qui permet de la visualiser page à page.

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

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

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

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

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

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

```
$ git log -Snom_de_fonction
```

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

Le tableau **Table 2-3** récapitule les options que nous venons de voir ainsi que quelques autres pour référence.

TABLE 2-3. *Options pour limiter la sortie de `git log`*

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

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

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
```

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

Annuler des actions

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

donc un des rares cas d'utilisation de Git où des erreurs de manipulation peuvent entraîner des pertes définitives de données.

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

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

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

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

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

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

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

Désindexer un fichier déjà indexé

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

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

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

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

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

        renommé :         README.md -> README

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

        modifié :         CONTRIBUTING.md
```

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

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

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

Réinitialiser un fichier modifié

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

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

        modifié :         CONTRIBUTING.md
```

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

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

        renommé :         README.md -> README
```

Vous pouvez constater que les modifications ont été annulées.

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

Si vous souhaitez seulement écarter momentanément cette modification, nous verrons comment mettre de côté et créer des branches dans le chapitre **Chapter 3** ; ce sont de meilleures façons de procéder.

Souvenez-vous, tout ce qui a été *validé* dans Git peut quasiment toujours être récupéré. Y compris des *commits* sur des branches qui ont été effacées ou des *commits* qui ont été écrasés par une validation avec l'option `--amend` (se référer au chapitre **"Récupération de données"** pour la récupération de don-

nées). Cependant, tout ce que vous perdez avant de l'avoir validé n'a aucune chance d'être récupérable via Git.

Travailler avec des dépôts distants

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

Afficher les dépôts distants

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

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

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

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

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

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

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

Ajouter des dépôts distants

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

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

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

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

La branche `master` de Paul est accessible localement en tant que `pb/master` — vous pouvez la fusionner dans une de vos propres branches, ou vous pouvez extraire une branche localement si vous souhaitez l'inspecter. Nous traiterons plus en détail de la nature des branches et de leur utilisation au chapitre **Chapter 3**.

Récupérer et tirer depuis des dépôts distants

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

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

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

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

Si vous avez créé une branche pour suivre l'évolution d'une branche distante (cf. la section suivante et le chapitre **Chapter 3** pour plus d'information), vous

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

Pousser son travail sur un dépôt distant

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

```
$ git push origin master
```

Cette commande ne fonctionne que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Si vous et quelqu'un d'autre clonez un dépôt au même moment et que cette autre personne pousse ses modifications et qu'après vous tentez de pousser les vôtres, votre poussée sera rejetée à juste titre. Vous devrez tout d'abord tirer les modifications de l'autre personne et les fusionner avec les vôtres avant de pouvoir pousser. Référez-vous au chapitre **Chapter 3** pour de plus amples informations sur les techniques pour pousser vers un serveur distant.

Inspecter un dépôt distant

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

```
$ git remote show origin
* distante origin
  URL de rapatriement : https://github.com/schacon/ticgit
  URL push : https://github.com/schacon/ticgit
```

```
  Branche HEAD : master
  Branches distantes :
    master suivi
    ticgit suivi
  Branche locale configurée pour 'git pull' :
    master fusionne avec la distante master
  Référence locale configurée pour 'git push' :
    master pousse vers master (à jour)
```

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

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

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

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

Retirer et déplacer des branches distantes

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

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

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

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

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

Étiquetage

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

Lister vos étiquettes

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

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

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

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

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

Créer des étiquettes

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

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

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

Les étiquettes annotées

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

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

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

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

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

ma version 1.4

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

    changed the version number
```

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

Les étiquettes légères

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

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

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

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

    changed the version number
```

Étiqueter après coup

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

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

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

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

Le *commit* a été étiqueté :

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

$ git show v1.2
```

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

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

    updated rakefile
...
```

Partager les étiquettes

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

```
$ git push origin v1.5
Décompte des objets: 14, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (12/12), fait.
Écriture des objets: 100% (14/14), 2.05KiB | 0 bytes/s, fait.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

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

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

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

Extraire une étiquette

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

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

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

Les alias Git

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

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

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

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

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

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

Cela rend les deux commandes suivantes équivalentes :

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

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

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

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

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

    test for current head

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

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

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

Résumé

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

Les branches avec Git 3

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

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

Les branches en bref

Pour réellement comprendre la manière dont Git gère les branches, nous devons revenir en arrière et examiner de plus près comment Git stocke ses données. Si vous vous souvenez bien du chapitre **Chapter 1**, Git ne stocke pas ses données comme une série de modifications ou de différences successives mais plutôt comme une série d'instantanés (appelés *snapshots*).

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

commit normal et de multiples parents pour un *commit* qui résulte de la fusion d'une ou plusieurs branches.

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

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

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

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

FIGURE 3-1

Un commit et son arbre

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

FIGURE 3-2

Commits et leurs parents

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

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

FIGURE 3-3

Une branche et l'historique de ses commits

Créer une nouvelle branche

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

```
$ git branch testing
```

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

FIGURE 3-4

Deux branches pointant vers la même série de commits

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

FIGURE 3-5

HEAD pointant vers une branche

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

```
$ git log --oneline --decorate
f30ab (HEAD, master, test) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

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

Basculer entre les branches

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

```
$ git checkout testing
```

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

FIGURE 3-6

HEAD pointe vers la branche courante

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

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

FIGURE 3-7

La branche HEAD avance à chaque commit

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

```
$ git checkout master
```

FIGURE 3-8

HEAD est déplacé lors d'un checkout

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

CHANGER DE BRANCHE MODIFIE LES FICHIERS DANS VOTRE RÉPERTOIRE DE TRAVAIL

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

Réalisons quelques autres modifications et validons à nouveau :

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

Maintenant, l'historique du projet a divergé (voir **Figure 3-9**). Vous avez créé une branche et basculé dessus, y avez réalisé des modifications, puis vous avez rebasculé sur la branche principale et réalisé d'autres modifications. Ces deux modifications sont isolées dans des branches séparées : vous pouvez basculer

d'une branche à l'autre et les fusionner quand vous êtes prêt. Et vous avez fait tout ceci avec de simples commandes : `branch`, `checkout` et `commit`.

FIGURE 3-9

Divergence d'historique

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

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (test) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

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

C'est une différence de taille avec la manière dont la plupart des VCS gèrent les branches, qui implique de copier tous les fichiers du projet dans un second

répertoire. Cela peut durer plusieurs secondes ou même quelques minutes selon la taille du projet, alors que pour Git, le processus est toujours instantané. De plus, comme nous enregistrons les parents quand nous validons les modifications, la détermination de l'ancêtre commun approprié pour la fusion est réalisée automatiquement pour nous et est généralement une opération très facile. Ces fonctionnalités encouragent naturellement les développeurs à créer et utiliser souvent des branches.

Voyons pourquoi vous devriez en faire autant.

Branches et fusions : les bases

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

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

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

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

Branches

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

FIGURE 3-10

Historique de commits simple

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

```
$ git checkout -b prob53
Switched to a new branch "prob53"
```

Cette commande est un raccourci pour :

```
$ git branch prob53
$ git checkout prob53
```

FIGURE 3-11

Création d'un nouveau pointeur de branche

Vous travaillez sur votre site web et validez vos modifications. Ce faisant, la branche `prob53` avance parce que vous l'avez extraite (c'est-à-dire que votre pointeur HEAD pointe dessus) :

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

FIGURE 3-12

La branche prob53 a avancé avec votre travail

À ce moment-là, vous recevez un appel qui vous apprend qu'il y a un problème sur le site web qu'il faut résoudre immédiatement. Avec Git, vous n'avez pas à déployer en même temps votre correctif et les modifications déjà validées pour `prob53` et vous n'avez pas non plus à vous fatiguer à annuler ces modifications avant de pouvoir appliquer votre correctif sur ce qu'il y a en production. Tout ce que vous avez à faire, c'est simplement de rebasculer sur la branche `master`.

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

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

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

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

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

FIGURE 3-13

Branche de correctif basée sur `master`

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

```
$ git checkout master
$ git merge correctif
Updating f42c576..3a0874c
Fast-forward
```

```
index.html | 2 ++
1 file changed, 2 insertions(+)
```

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

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

FIGURE 3-14

Avancement du pointeur de `master` sur `correctif`

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

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

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

```
$ git checkout prob53
Switched to branch "prob53"
$ vim index.html
$ git commit -a -m 'Nouveau pied de page terminé [issue 53]'
[prob53 ad82d7a] Nouveau pied de page terminé [issue 53]
1 file changed, 1 insertion(+)
```

FIGURE 3-15

Le travail continue sur `prob53`

Il est utile de noter que le travail réalisé dans la branche `correctif` n'est pas contenu dans les fichiers de la branche `prob53`. Si vous avez besoin de les y rapatrier, vous pouvez fusionner la branche `master` dans la branche `prob53` en lançant la commande `git merge master`, ou vous pouvez retarder l'intégration de ces modifications jusqu'à ce que vous décidiez plus tard de rapatrier la branche `prob53` dans `master`.

Fusions (*Merges*)

Supposons que vous ayez décidé que le travail sur le problème #53 était terminé et prêt à être fusionné dans la branche `master`. Pour ce faire, vous allez fusionner votre branche `prob53` de la même manière que vous l'avez fait plus tôt pour la branche `correctif`. Tout ce que vous avez à faire est d'extraire la branche dans laquelle vous souhaitez fusionner et lancer la commande `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge prob53
Merge made by the 'recursive' strategy.
README |    1 +
1 file changed, 1 insertion(+)
```

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

FIGURE 3-16

Trois instantanés utilisés dans une fusion classique

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

FIGURE 3-17

Un commit de fusion

Il est à noter que Git détermine par lui-même le meilleur ancêtre commun à utiliser comme base de la fusion. Ce comportement est très différent de celui de CVS ou Subversion (avant la version 1.5), où le développeur en charge de la fusion doit trouver par lui-même la meilleure base. Cela rend la fusion beaucoup plus facile dans Git que dans les autres systèmes.

À présent que votre travail a été fusionné, vous n'avez plus besoin de la branche `prob53`. Vous pouvez fermer le ticket dans votre outil de suivi des tâches et supprimer la branche :

```
$ git branch -d prob53
```

Conflits de fusions (*Merge conflicts*)

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

```
$ git merge prob53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

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

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

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

    both modified:      index.html

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

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

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> prob53:index.html
```

Cela signifie que la version dans HEAD (votre branche `master`, parce que c'est celle que vous aviez extraite quand vous avez lancé votre commande de fusion) est la partie supérieure de ce bloc (tout ce qui se trouve au-dessus de la ligne `=======`), tandis que la version de votre branche `prob53` se trouve en dessous. Pour résoudre le conflit, vous devez choisir une partie ou l'autre ou bien fusionner leurs contenus vous-même. Par exemple, vous pourriez choisir de résoudre ce conflit en remplaçant tout le bloc par ceci :

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

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

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

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerg
Merging:
index.html

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

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

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

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

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

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

Changes to be committed:

    modified:   index.html
```

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

```
Merge branch 'prob53'

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


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

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

Gestion des branches

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

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

```
$ git branch
  prob53
* master
  test
```

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

```
$ git branch -v
  prob53   93b412c fix javascript issue
* master  7a98805 Merge branch 'prob53'
  test 782fd34 add scott to the author list in the readmes
```

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

```
$ git branch --merged
  prob53
* master
```

Comme vous avez déjà fusionné `prob53` un peu plus tôt, vous la voyez dans votre liste. Les branches de cette liste qui ne comportent pas le préfixe `*` peuvent généralement être effacées sans risque au moyen de `git branch -d` puisque vous avez déjà intégré leurs modifications dans une autre branche et ne risquez donc pas de perdre quoi que ce soit.

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

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

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

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

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

Travailler avec les branches

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

Branches au long cours

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

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

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

FIGURE 3-18

Vue linéaire de branches dans un processus de stabilité progressive

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

FIGURE 3-19

Vue en silo de branches dans un processus de stabilité progressive

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

Les branches thématiques

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

Vous avez déjà vu ces branches dans la section précédente avec les branches `prob53` et `correctif` que vous avez créées. Vous y avez réalisé quelques *commits* et vous les avez supprimées immédiatement après les avoir fusionnées dans votre branche principale. Cette technique vous permet de changer de contexte rapidement et complètement. Parce que votre travail est isolé dans des silos où toutes les modifications sont liées à une thématique donnée, il est beaucoup plus simple de réaliser des revues de code. Vous pouvez conserver vos modifications dans ces branches pendant des minutes, des jours ou des mois puis les fusionner quand elles sont prêtes, indépendamment de l'ordre dans lequel elles ont été créées ou traitées.

Prenons l'exemple suivant : alors que vous développez (sur `master`), vous créez une nouvelle branche pour un problème (`iss91`), travaillez un peu sur ce problème puis créez une seconde branche pour essayer de trouver une autre manière de le résoudre (`prob91v2`). Vous retournez ensuite sur la branche `master` pour y travailler pendant un moment puis finalement créez une dernière branche (`ideeidiote`) contenant une idée dont vous doutez de la pertinence. Votre historique de *commits* pourrait ressembler à ceci :

FIGURE 3-20

Branches thématiques multiples

Maintenant, supposons que vous décidiez que vous préférez la seconde solution pour le problème (`prob91v2`) et que vous ayez montré la branche `ideeidiote` à vos collègues qui vous ont dit qu'elle était géniale. Vous pouvez jeter la branche `iss91` originale (perdant ainsi les *commits* `C5` et `C6`) et fusionner les deux autres branches. Votre historique ressemble à présent à ceci :

FIGURE 3-21

Historique après la fusion de `ideeidiote` et `prob91v2`

Nous verrons au chapitre **Chapter 5**, d'autres méthodes et processus possibles pour vos projets Git. Nous vous invitons à prendre connaissance de ce chapitre avant de vous décider pour une méthode particulière de gestion de vos branches pour votre prochain projet.

Il est important de se souvenir que lors de la réalisation de toutes ces actions, ces branches sont complètement locales. Lorsque vous créez et fusionnez

des branches, ceci est réalisé uniquement dans votre dépôt Git local et aucune communication avec un serveur n'a lieu.

Branches distantes

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

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

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

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

`ORIGIN` N'EST PAS SPÉCIAL

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

FIGURE 3-22

Dépôts distant et local après un clone

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

FIGURE 3-23

Les travaux locaux et distants peuvent diverger

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

FIGURE 3-24

`git fetch` met à jour vos références distantes

Pour démontrer l'usage de multiples serveurs distants et le fonctionnement des branches distantes pour ces projets distants, supposons que vous avez un autre serveur Git interne qui n'est utilisé que par une équipe de développeurs. Ce serveur se trouve sur `git.equipe1.notresociete.com`. Vous pouvez l'ajouter aux références distantes de votre projet en lançant la commande `git remote add` comme nous l'avons décrit au chapitre **Chapter 2**. Nommez ce serveur distant `equipeun` qui sera le raccourci pour l'URL complète.

FIGURE 3-25

Ajout d'un nouveau server en tant que référence distante

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

FIGURE 3-26

Suivi d'une branche distante `equipeun/master`

Pousser les branches

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

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

```
$ git push origin correctionserveur
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
```

```
To https://github.com/schacon/simplegit
 * [new branch]      correctionserveur -> correctionserveur
```

Il s'agit en quelque sorte d'un raccourci. Git développe automatiquement le nom de branche `correctionserveur` en `refs/heads/correctionserveur:refs/heads/correctionserveur`, ce qui signifie "Prendre ma branche locale `correctionserveur` et la pousser pour mettre à jour la branche distante `correctionserveur`". Nous traiterons plus en détail la partie `refs/heads/` au chapitre **Chapter 10** mais généralement, vous pouvez l'oublier. Vous pouvez aussi lancer `git push origin correctionserveur:correctionserveur`, qui réalise la même chose — ce qui signifie « Prendre ma branche `correctionserveur` et en faire la branche `correctionserveur` distante ». Vous pouvez utiliser ce format pour pousser une branche locale vers une branche distante nommée différemment. Si vous ne souhaitez pas l'appeler `correctionserveur` sur le serveur distant, vous pouvez lancer à la place `git push origin correctionserveur:branchegeniale` pour pousser votre branche locale `correctionserveur` sur la branche `branchegeniale` sur le dépôt distant.

NE RENSEIGNEZ PAS VOTRE MOT DE PASSE À CHAQUE FOIS

Si vous utilisez une URL en HTTPS, le serveur Git vous demandera votre nom d'utilisateur et votre mot de passe pour vous authentifier. Par défaut, vous devez entrer ces informations sur votre terminal et le serveur pourra alors déterminer si vous être autorisé à pousser.

Si vous ne voulez pas entrer ces informations à chaque fois que vous poussez, vous pouvez mettre en place un "cache d'identification" (*credential cache*). Son fonctionnement le plus simple consiste à garder ces informations en mémoire pour quelques minutes mais vous pouvez configurer ce délai en lançant la commande `git config --global credential.helper cache`.

Pour davantage d'informations sur les différentes options de cache d'identification disponibles, vous pouvez vous référer au chapitre **"Stockage des identifiants"**.

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

```
$ git fetch origin
remote: Counting objects: 7, done.
```

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

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

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

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

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

Suivre les branches

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

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

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

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

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

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

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

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

RACCOURCI VERS *UPSTREAM*

Quand vous avez une branche de suivi configurée, vous pouvez y faire référence grâce au raccourci `@{upstream}` ou `@{u}`. Ainsi, si vous êtes sur la branche `master` et sa branche de suivi `origin/master`, vous pouvez utiliser quelque chose comme `git merge @{u}` au lieu de `git merge origin/master` si vous le souhaitez.

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

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
```

```
* correctionserveur f8674d9 [equipe1/correction-serveur-ok: ahead 3, behind 1] thi
  test   5ea463a trying something new
```

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

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

Tirer une branche (*Pulling*)

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

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

Suppression de branches distantes

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

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

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

Rebaser (*Rebasing*)

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

Les bases

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

FIGURE 3-27

Historique divergeant simple

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

FIGURE 3-28

Fusion pour intégrer des travaux aux historiques divergeants

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

Dans cet exemple, vous lanceriez les commandes suivantes :

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

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

FIGURE 3-29

Rebasage des modifications introduites par `C4` sur `C3`

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

FIGURE 3-30

Avance rapide de la branche `master`

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

toutes les modifications apparaissent en série même si elles ont eu lieu en parallèle.

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

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

Rebases plus intéressants

Vous pouvez aussi faire rejouer votre rebasage sur autre chose qu'une branche. Prenez un historique tel que **Figure 3-31** par exemple. Vous avez créé une branche thématique (`serveur`) pour ajouter des fonctionnalités côté serveur à votre projet et avez réalisé un *commit*. Ensuite, vous avez créé une branche pour ajouter des modifications côté client (`client`) et avez validé plusieurs fois. Finalement, vous avez rebasculé sur la branche `serveur` et avez réalisé quelques *commits* supplémentaires.

FIGURE 3-31

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

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

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

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

FIGURE 3-32

Rebaser deux branches thématiques l'une sur l'autre

Maintenant, vous pouvez faire une avance rapide sur votre branche `master` (cf. **Figure 3-33**):

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

FIGURE 3-33

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

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

```
$ git rebase master serveur
```

Cette commande rejoue les modifications de `serveur` sur le sommet de la branche `master`, comme indiqué dans **Figure 3-34**.

FIGURE 3-34

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

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

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

Vous pouvez effacer les branches `client` et `serveur` une fois que tout le travail est intégré et que vous n'en avez plus besoin, éliminant tout l'historique de ce processus, comme visible sur **Figure 3-35** :

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

FIGURE 3-35

Historique final des commits

Les dangers du rebasage

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

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

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

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

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

FIGURE 3-36

Cloner un dépôt et baser du travail dessus

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

FIGURE 3-37

Récupération de commits et fusion dans votre copie

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

FIGURE 3-38

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

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

FIGURE 3-39

Vous fusionnez le même travail une nouvelle fois dans un nouveau commit de fusion

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

Rebaser quand vous rebasez

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

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

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

Par exemple, dans le scénario précédent, si au lieu de fusionner quand nous étions à l'étape **Figure 3-38** nous exécutons la commande `git rebase equipe1/master`, Git va :

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

Ainsi, au lieu du résultat que nous avons observé au chapitre **Figure 3-39**, nous aurions pu finir avec quelque chose qui ressemblerait davantage à **Figure 3-40**.

FIGURE 3-40

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

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

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

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

Si vous considérez le fait de rebaser comme un moyen de nettoyer et réarranger des *commits* avant de les pousser et si vous vous en tenez à ne rebaser

que des *commits* qui n'ont jamais été publiés, tout ira bien. Si vous tentez de rebaser des *commits* déjà publiés sur lesquels les gens ont déjà basé leur travail, vous allez au devant de gros problèmes et votre équipe vous en tiendra rigueur.

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

Rebaser ou Fusionner

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

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

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

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

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

Résumé

Nous avons traité les bases des branches et des fusions dans Git. Vous devriez désormais être à l'aise pour créer et basculer sur de nouvelles branches, basculer entre branches et fusionner des branches locales. Vous devriez aussi être capable de partager vos branches en les poussant sur un serveur partagé, de travailler avec d'autres personnes sur des branches partagées et de re-baser vos branches avant de les partager. Nous aborderons ensuite tout ce que vous devez savoir pour faire tourner votre propre serveur d'hébergement de dépôts.

Git sur le serveur 4

À présent, vous devriez être capable de réaliser la plupart des tâches quotidiennes impliquant Git. Néanmoins, pour pouvoir collaborer avec d'autres personnes au moyen de Git, vous allez devoir disposer d'un dépôt distant Git. Bien que vous puissiez techniquement tirer et pousser des modifications depuis et vers des dépôts personnels, cette pratique est déconseillée parce qu'elle introduit très facilement une confusion avec votre travail actuel. De plus, vous souhaitez que vos collaborateurs puissent accéder à votre dépôt de sources, y compris si vous n'êtes pas connecté — disposer d'un dépôt accessible en permanence peut s'avérer utile. De ce fait, la méthode canonique pour collaborer consiste à instancier un dépôt intermédiaire auquel tout le monde a accès, que ce soit pour pousser ou tirer.

Un serveur Git est simple à lancer. Premièrement, vous devez choisir quels protocoles seront supportés. La première partie de ce chapitre traite des protocoles disponibles et de leurs avantages et inconvénients. La partie suivante explique certaines configurations typiques de ces protocoles et comment les mettre en œuvre. Enfin, nous traiterons de quelques types d'hébergement, si vous souhaitez héberger votre code sur un serveur tiers, sans avoir à installer et maintenir un serveur par vous-même.

Si vous ne voyez pas d'intérêt à gérer votre propre serveur, vous pouvez sauter directement à la dernière partie de ce chapitre pour détailler les options pour mettre en place un compte hébergé, avant de continuer au chapitre suivant dans lequel les problématiques de développement distribué sont abordées.

Un dépôt distant est généralement un *dépôt nu* (*bare repository*) : un dépôt Git qui n'a pas de copie de travail. Comme ce dépôt n'est utilisé que comme centralisateur de collaboration, il n'y a aucune raison d'extraire un instantané sur le disque ; seules les données Git sont nécessaires. Pour simplifier, un dépôt nu est le contenu du répertoire `.git` sans fioriture.

Protocoles

Git peut utiliser quatre protocoles réseau majeurs pour transporter des données : local, HTTP, *Secure Shell* (SSH) et Git. Nous allons voir leur nature et dans quelles circonstances ils peuvent (ou ne peuvent pas) être utilisés.

Protocole local

Le protocole de base est le protocole *local* pour lequel le dépôt distant est un autre répertoire dans le système de fichiers. Il est souvent utilisé si tous les membres de l'équipe ont accès à un répertoire partagé via NFS par exemple ou dans le cas moins probable où tous les développeurs travaillent sur le même ordinateur. Ce dernier cas n'est pas optimum car tous les dépôts seraient hébergés de fait sur le même ordinateur, rendant ainsi toute défaillance catastrophique.

Si vous disposez d'un système de fichiers partagé, vous pouvez cloner, pousser et tirer avec un dépôt local. Pour cloner un dépôt ou pour l'utiliser comme dépôt distant d'un projet existant, utilisez le chemin vers le dépôt comme URL. Par exemple, pour cloner un dépôt local, vous pouvez lancer ceci :

```
$ git clone /opt/git/project.git
```

Ou bien cela :

```
$ git clone file:///opt/git/project.git
```

Git opère légèrement différemment si vous spécifiez explicitement le protocole `file://` au début de l'URL. Si vous spécifiez simplement le chemin et si la destination se trouve sur le même système de fichiers, Git tente d'utiliser des liens physiques pour les fichiers communs. Si vous spécifiez le protocole `file://`, Git lance un processus d'accès à travers le réseau, ce qui est généralement moins efficace. La raison d'utiliser spécifiquement le préfixe `file://` est la volonté d'obtenir une copie propre du dépôt, sans aucune référence ou aucun objet supplémentaire qui pourraient résulter d'un import depuis un autre système de gestion de version ou d'une action similaire (voir chapitre **Chapter 10** pour les tâches de maintenance). Nous utiliserons les chemins normaux par la suite car c'est la méthode la plus efficace.

Pour ajouter un dépôt local à un projet Git existant, lancez ceci :

```
$ git remote add local_proj /opt/git/project.git
```

Ensuite, vous pouvez pousser vers et tirer depuis ce dépôt distant de la même manière que vous le feriez pour un dépôt accessible sur le réseau.

AVANTAGES

Les avantages des dépôts accessibles sur le système de fichiers sont qu'ils sont simples et qu'ils utilisent les permissions du système de fichiers. Si vous avez déjà un montage partagé auquel toute votre équipe a accès, déployer un dépôt est extrêmement facile. Vous placez la copie du dépôt nu à un endroit accessible de tous et positionnez correctement les droits de lecture/écriture de la même manière que pour tout autre partage. Nous aborderons la méthode pour exporter une copie de dépôt nu à cette fin dans la section suivante **"Installation de Git sur un serveur"**.

C'est un choix satisfaisant pour partager rapidement le travail. Si vous et votre coéquipier travaillez sur le même projet et qu'il souhaite partager son travail, lancer une commande telle que `git pull /home/john/project` est certainement plus simple que de passer par un serveur intermédiaire.

INCONVÉNIENTS

Les inconvénients de cette méthode sont qu'il est généralement plus difficile de rendre disponible un partage réseau depuis de nombreux endroits que de simplement gérer des accès réseau. Si vous souhaitez pousser depuis votre portable à la maison, vous devez monter le partage distant, ce qui peut s'avérer plus difficile et plus lent que d'y accéder directement via un protocole réseau.

Il faut aussi mentionner que ce n'est pas nécessairement l'option la plus rapide à l'utilisation si un partage réseau est utilisé. Un dépôt local n'est rapide que si l'accès aux fichiers est rapide. Un dépôt accessible sur un montage NFS est souvent plus lent qu'un dépôt accessible via SSH sur le même serveur qui ferait tourner Git avec un accès aux disques locaux.

Protocoles sur HTTP

Git peut communiquer sur HTTP de deux manières. Avant Git 1.6.6, il n'existait qu'une seule manière qui était très simple et généralement en lecture seule. Depuis la version 1.6.6, il existe un nouveau protocole plus intelligent qui nécessite que Git puisse négocier les transferts de données de manière similaire à ce qu'il fait pour SSH. Ces dernières années, le nouveau protocole HTTP a gagné en popularité du fait qu'il est plus simple à utiliser et plus efficace dans ses

communications. La nouvelle version est souvent appelée protocole HTTP « intelligent » et l'ancienne version protocole HTTP « idiot ». Nous allons voir tout d'abord le protocole HTTP « intelligent ».

HTTP INTELLIGENT

Le protocole HTTP « intelligent » se comporte de manière très similaire aux protocoles SSH ou Git mais fonctionne par-dessus les ports HTTP/S et peut utiliser différents mécanismes d'authentification, ce qui le rend souvent plus facile pour l'utilisateur que SSH, puisque l'on peut utiliser des méthodes telles que l'authentification par utilisateur/mot de passe plutôt que de devoir gérer des clés SSH.

C'est devenu probablement le moyen le plus populaire d'utiliser Git, car il peut être utilisé pour du service anonyme, comme le protocole `git://` aussi bien que pour pousser avec authentification et chiffrement, comme le protocole SSH. Au lieu de devoir gérer différentes URL pour ces usages, vous pouvez maintenant utiliser une URL unique pour les deux. Si vous essayez de pousser et que le dépôt requiert une authentification (ce qui est normal), le serveur peut demander un nom d'utilisateur et un mot de passe. De même pour les accès en lecture.

En fait, pour les services tels que GitHub, l'URL que vous utilisez pour visualiser le dépôt sur le web (par exemple `https://github.com/schacon/simplegit[]`) est la même URL utilisable pour le cloner et, si vous en avez les droits, y pousser.

HTTP IDIOT

Si le serveur ne répond pas avec un service Git HTTP intelligent, le client Git essayera de se rabattre sur le protocole HTTP « idiot ». Le protocole idiot consiste à servir le dépôt Git nu comme des fichiers normaux sur un serveur web. La beauté du protocole idiot réside dans sa simplicité de mise en place. Tout ce que vous avez à faire, c'est de copier les fichiers de votre dépôt nu sous la racine de documents HTTP et de positionner un crochet (**hook**) `post-update` spécifique, et c'est tout (voir **"Crochets Git"**). Dès ce moment, tous ceux qui peuvent accéder au serveur web sur lequel vous avez déposé votre dépôt peuvent le cloner. Pour permettre un accès en lecture seule à votre dépôt via HTTP, faîtes quelque chose comme :

```
$ cd /var/www/htdocs/
$ git clone --bare /chemin/vers/projet_git projetgit.git
$ cd projetgit.git
```

```
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Et voilà ! Le crochet `post-update` livré par défaut avec Git lance la commande appropriée (`git update-server-info`) pour faire fonctionner correctement le clonage et la récupération HTTP. Cette commande est lancée quand vous poussez sur ce dépôt (peut-être sur SSH). Ensuite, les autres personnes peuvent cloner via quelque chose comme :

```
$ git clone https://exemple.com/projetgit.git
```

Dans ce cas particulier, nous utilisons le chemin `/var/www/htdocs` qui est le plus commun pour une configuration Apache, mais vous pouvez utiliser n'importe quel serveur web statique – placez juste les dépôts nus dans son chemin. Les données Git sont servies comme de simples fichiers statiques (voir **Chapter 10** pour la manière exacte dont elles sont servies).

Généralement, vous choisirez soit de lancer un serveur HTTP intelligent avec des droits en lecture/écriture ou de fournir simplement les fichiers en lecture seule par le protocole idiot. Il est rare de mélanger les deux types de protocoles.

AVANTAGES

Nous nous concentrerons sur les avantages de la version intelligente du protocole sur HTTP.

La simplicité vient de l'utilisation d'une seule URL pour tous les types d'accès et de la demande d'authentification seulement en cas de besoin. Ces deux caractéristiques rendent les choses très faciles pour l'utilisateur final. La possibilité de s'authentifier avec un nom d'utilisateur et un mot de passe apporte un gros avantage par rapport à SSH puisque les utilisateurs n'ont plus à générer localement les clés SSH et à télécharger leur clé publique sur le serveur avant de pouvoir interagir avec lui. Pour les utilisateurs débutants ou pour des utilisateurs utilisant des systèmes où SSH est moins commun, c'est un avantage d'utilisabilité majeur. C'est aussi un protocole très rapide et efficace, similaire à SSH.

Vous pouvez aussi servir vos dépôts en lecture seule sur HTTPS, ce qui signifie que vous pouvez chiffrer les communications ; ou vous pouvez pousser jusqu'à faire utiliser des certificats SSL à vos clients.

Un autre avantage est que HTTP/S sont des protocoles si souvent utilisés que les pare-feux d'entreprise sont souvent paramétrés pour les laisser passer.

INCONVÉNIENTS

Configurer Git sur HTTP/S peut être un peu plus difficile que sur SSH sur certains serveurs. Mis à part cela, les autres protocoles ont peu d'avantages sur le protocole HTTP intelligent pour servir Git.

Si vous utilisez HTTP pour pousser de manière authentifiée, fournir vos information d'authentification est parfois plus compliqué qu'utiliser des clés sur SSH. Il existe cependant des outils de mise en cache d'informations d'authentification, comme Keychain sur OSX et Credential Manager sur Windows pour rendre cela indolore. Reportez-vous à **"Stockage des identifiants"** pour voir comment configurer la mise en cache des mots de passe HTTP sur votre système.

Protocole SSH

SSH est un protocole répandu de transport pour Git en auto-hébergement. Cela est dû au fait que l'accès SSH est déjà en place à de nombreux endroits et que si ce n'est pas le cas, cela reste très facile à faire. Cela est aussi dû au fait que SSH est un protocole authentifié ; et comme il est très répandu, il est généralement facile à mettre en œuvre et à utiliser.

Pour cloner un dépôt Git à travers SSH, spécifiez le préfixe `ssh://` dans l'URL comme ceci :

```
$ git clone ssh://utilisateur@serveur/projet.git
```

Vous pouvez utiliser aussi la syntaxe scp habituelle avec le protocole SSH :

```
$ git clone utilisateur@serveur:projet.git
```

Vous pouvez aussi ne pas spécifier de nom d'utilisateur et Git utilisera par défaut le nom de login.

AVANTAGES

Les avantages liés à l'utilisation de SSH sont nombreux. Premièrement, SSH est relativement simple à mettre en place, les *daemons* SSH sont facilement disponibles, les administrateurs réseau sont habitués à les gérer et de nombreuses distributions de systèmes d'exploitation en disposent ou proposent des outils pour les gérer. Ensuite, l'accès distant à travers SSH est sécurisé, toutes les données sont chiffrées et authentifiées. Enfin, comme les protocoles HTTP/S, Git et

local, SSH est efficace et permet de comprimer autant que possible les données avant de les transférer.

INCONVÉNIENTS

Le point négatif avec SSH est qu'il est impossible de proposer un accès anonyme au dépôt. Les accès sont régis par les permissions SSH, même pour un accès en lecture seule, ce qui s'oppose à une optique open source. Si vous souhaitez utiliser Git dans un environnement d'entreprise, SSH peut bien être le seul protocole nécessaire. Si vous souhaitez proposer de l'accès anonyme en lecture seule à vos projets, vous aurez besoin de SSH pour vous permettre de pousser mais un autre protocole sera nécessaire pour permettre à d'autres de tirer.

Protocole Git

Vient ensuite le protocole Git. Celui-ci est géré par un *daemon* spécial livré avec Git. Ce *daemon* (démon, processus en arrière-plan) écoute sur un port dédié (9418) et propose un service similaire au protocole SSH, mais sans aucune sécurisation. Pour qu'un dépôt soit publié via le protocole Git, le fichier `git-daemon-export-ok` doit exister mais mise à part cette condition sans laquelle le *daemon* refuse de publier un projet, il n'y a aucune sécurité. Soit le dépôt Git est disponible sans restriction en lecture, soit il n'est pas publié. Cela signifie qu'il ne permet pas de pousser des modifications. Vous pouvez activer la capacité à pousser mais étant donné l'absence d'authentification, n'importe qui sur Internet ayant trouvé l'URL du projet peut pousser sur le dépôt. Autant dire que ce mode est rarement recherché.

AVANTAGES

Le protocole Git est souvent le protocole avec la vitesse de transfert la plus rapide. Si vous devez servir un gros trafic pour un projet public ou un très gros projet qui ne nécessite pas d'authentification en lecture, il est très probable que vous devriez installer un *daemon* Git. Il utilise le même mécanisme de transfert de données que SSH, la surcharge du chiffrement et de l'authentification en moins.

INCONVÉNIENTS

Le défaut du protocole Git est le manque d'authentification. N'utiliser que le protocole Git pour accéder à un projet n'est généralement pas suffisant. Il faut le coupler avec un accès SSH ou HTTPS pour quelques développeurs qui auront

le droit de pousser (écrire) et le garder en accès `git://` pour la lecture seule. C'est aussi le protocole le plus difficile à mettre en place. Il doit être géré par son propre *daemon* qui est spécifique. Il nécessite la configuration d'un *daemon* `xinetd` ou apparenté, ce qui est loin d'être simple. Il nécessite aussi un accès à travers le pare-feu au port 9418 qui n'est pas un port ouvert en standard dans les pare-feux professionnels. Derrière les gros pare-feux professionnels, ce port obscur est tout simplement bloqué.

Installation de Git sur un serveur

Nous allons à présent traiter de la configuration d'un service Git gérant ces protocoles sur votre propre serveur.

> Les commandes et étapes décrites ci-après s'appliquent à des installations simplifiées sur un serveur à base de Linux, bien qu'il soit aussi possible de faire fonctionner ces services sur des serveurs Mac ou Windows. La mise en place effective d'un serveur en production au sein d'une infrastructure englobera vraisemblablement des différences dans les mesures de sécurité et les outils système, mais ceci devrait permettre de se faire une idée générale des besoins.

Pour réaliser l'installation initiale d'un serveur Git, il faut exporter un dépôt existant dans un nouveau dépôt nu — un dépôt qui ne contient pas de copie de répertoire de travail. C'est généralement simple à faire. Pour cloner votre dépôt en créant un nouveau dépôt nu, lancez la commande clone avec l'option `--bare`. Par convention, les répertoires de dépôt nu finissent en `.git`, de cette manière :

```
$ git clone --bare mon_project mon_projet.git
Clonage dans le dépôt nu 'mon_projet.git'...
fait.
```

Vous devriez maintenant avoir une copie des données de Git dans votre répertoire `mon_project.git`.

C'est grossièrement équivalent à :

```
$ cp -Rf mon_projet/.git mon_projet.git
```

Il y a quelques légères différences dans le fichier de configuration mais pour l'utilisation envisagée, c'est très proche. La commande extrait le répertoire Git sans répertoire de travail et crée un répertoire spécifique pour l'accueillir.

Copie du dépôt nu sur un serveur

À présent que vous avez une copie nue de votre dépôt, il ne reste plus qu'à la placer sur un serveur et à régler les protocoles. Supposons que vous avez mis en place un serveur nommé `git.exemple.com` auquel vous avez accès par SSH et que vous souhaitez stocker vos dépôts Git dans le répertoire `/opt/git`. En supposant que `/opt/git` existe, vous pouvez mettre en place votre dépôt en copiant le dépôt nu :

```
$ scp -r mon_projet.git utilisateur@git.exemple.com:/opt/git
```

À partir de maintenant, tous les autres utilisateurs disposant d'un accès SSH au serveur et ayant un accès en lecture seule au répertoire `/opt/git` peuvent cloner votre dépôt en lançant la commande :

```
$ git clone utilisateur@git.exemple.com:/opt/git/mon_projet.git
```

Si un utilisateur se connecte via SSH au serveur et a accès en écriture au répertoire `/opt/git/mon_projet.git`, il aura automatiquement accès pour pousser.

Git ajoutera automatiquement les droits de groupe en écriture à un dépôt si vous lancez la commande `git init` avec l'option `--shared`.

```
$ ssh utilisateur@git.exemple.com
$ cd /opt/git/mon_projet.git
$ git init --bare --shared
```

Vous voyez comme il est simple de prendre un dépôt Git, créer une version nue et la placer sur un serveur auquel vous et vos collaborateurs avez accès en SSH. Vous voilà prêts à collaborer sur le même projet.

Il faut noter que c'est littéralement tout ce dont vous avez besoin pour démarrer un serveur Git utile auquel plusieurs personnes ont accès : ajoutez simplement des comptes SSH sur un serveur, et collez un dépôt nu quelque part où tous les utilisateurs ont accès en lecture et écriture. Vous êtes prêts à travailler, vous n'avez besoin de rien d'autre.

Dans les chapitres à venir, nous traiterons de mises en place plus sophistiquées. Ces sujets incluront l'élimination du besoin de créer un compte système pour chaque utilisateur, l'accès public aux dépôts, la mise en place d'interfaces utilisateur web, etc. Néanmoins, gardez à l'esprit que pour collaborer avec quelques personnes sur un projet privé, tout ce qu'il faut, c'est un serveur SSH et un dépôt nu.

Petites installations

Si vous travaillez dans un petit groupe ou si vous n'êtes qu'en phase d'essai de Git au sein de votre société avec peu de développeurs, les choses peuvent rester simples. Un des aspects les plus compliqués de la mise en place d'un serveur Git est la gestion des utilisateurs. Si vous souhaitez que certains dépôts ne soient accessibles à certains utilisateurs qu'en lecture seule et en lecture/écriture pour d'autres, la gestion des accès et des permissions peut devenir difficile à régler.

ACCÈS SSH

Si vous disposez déjà d'un serveur auquel tous vos développeurs ont un accès SSH, il est généralement plus facile d'y mettre en place votre premier dépôt car vous n'aurez quasiment aucun réglage supplémentaire à faire (comme nous l'avons expliqué dans le chapitre précédent). Si vous souhaitez des permissions d'accès plus complexes, vous pouvez les mettre en place par le jeu des permissions standards sur le système de fichiers du système d'exploitation de votre serveur.

Si vous souhaitez placer vos dépôts sur un serveur qui ne dispose pas déjà de comptes pour chacun des membres de votre équipe qui aurait accès en écriture, alors vous devrez mettre en place un accès SSH pour eux. En supposant que pour vos dépôts, vous disposiez déjà d'un serveur SSH installé et auquel vous avez accès.

Il y a quelques moyens de donner un accès à tout le monde dans l'équipe. Le premier est de créer des comptes pour tout le monde, ce qui est logique mais peut s'avérer lourd. Vous ne souhaiteriez sûrement pas lancer `adduser` et entrer un mot de passe temporaire pour chaque utilisateur.

Une seconde méthode consiste à créer un seul utilisateur Git sur la machine, demander à chaque développeur nécessitant un accès en écriture de vous envoyer une clé publique SSH et d'ajouter la-dite clé au fichier `~/.ssh/authorized_keys` de votre utilisateur Git. À partir de là, tout le monde sera capable d'accéder à la machine via l'utilisateur Git. Cela n'affecte en rien les données de *commit* — les informations de l'utilisateur SSH par lequel on se connecte n'affectent pas les données de *commit* enregistrées.

Une dernière méthode consiste à faire une authentification SSH auprès d'un serveur LDAP ou tout autre système d'authentification centralisé que vous utiliseriez déjà. Tant que chaque utilisateur peut accéder à un shell sur la machine, n'importe quel schéma d'authentification SSH devrait fonctionner.

Génération des clés publiques SSH

Cela dit, de nombreux serveurs Git utilisent une authentification par clés publiques SSH. Pour fournir une clé publique, chaque utilisateur de votre système doit la générer s'il n'en a pas déjà. Le processus est similaire sur tous les systèmes d'exploitation. Premièrement, l'utilisateur doit vérifier qu'il n'en a pas déjà une. Par défaut, les clés SSH d'un utilisateur sont stockées dans le répertoire `~/.ssh` du compte. Vous pouvez facilement vérifier si vous avez déjà une clé en listant le contenu de ce répertoire :

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

Recherchez une paire de fichiers appelés *quelquechose* et *quelquechose*`.pub` où le *quelquechose* en question est généralement `id_dsa` ou `id_rsa`. Le fichier en `.pub` est la clé publique tandis que l'autre est la clé privée. Si vous ne voyez pas ces fichiers (ou n'avez même pas de répertoire `.ssh`), vous pouvez les créer en lançant un programme appelé `ssh-keygen` fourni par le paquet SSH sur les systèmes Linux/Mac et MSysGit pour Windows :

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Premièrement, le programme demande confirmation de l'endroit où vous souhaitez sauvegarder la clé (`.ssh/id_rsa`) puis il demande deux fois d'entrer

un mot de passe qui peut être laissé vide si vous ne souhaitez pas devoir le taper quand vous utilisez la clé.

Maintenant, chaque utilisateur ayant suivi ces indications doit envoyer la clé publique à la personne en charge de l'administration du serveur Git (en supposant que vous utilisez un serveur SSH réglé pour l'utilisation de clés publiques). Ils doivent copier le contenu du fichier `.pub` et l'envoyer par courriel. Les clés publiques ressemblent à ceci :

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Pour un tutoriel plus approfondi sur la création de clé SSH sur différents systèmes d'exploitation, référez-vous au guide GitHub sur les clés SSH à ***https://help.github.com/articles/generating-ssh-keys***.

Mise en place du serveur

Parcourons les étapes de la mise en place d'un accès SSH côté serveur. Dans cet exemple, vous utiliserez la méthode des `authorized_keys` pour authentifier vos utilisateurs. Nous supposerons également que vous utilisez une distribution Linux standard telle qu'Ubuntu. Premièrement, créez un utilisateur *git* et un répertoire `.ssh` pour cet utilisateur.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Ensuite, vous devez ajouter la clé publique d'un développeur au fichier `authorized_keys` de l'utilisateur Git. Supposons que vous avez reçu quelques clés par courriel et les avez sauvées dans des fichiers temporaires. Pour rappel, une clé publique ressemble à ceci :

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
```

```
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Il suffit de les ajouter au fichier `authorized_keys` de l'utilisateur `git` dans son répertoire `.ssh` :

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Maintenant, vous pouvez créer un dépôt vide nu en lançant la commande `git init` avec l'option `--bare`, ce qui initialise un dépôt sans répertoire de travail :

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

Alors, John, Josie ou Jessica peuvent pousser la première version de leur projet vers ce dépôt en l'ajoutant en tant que dépôt distant et en lui poussant une branche. Notons que quelqu'un doit se connecter par shell au serveur et créer un dépôt nu pour chaque ajout de projet. Supposons que le nom du serveur soit `gitserveur`. Si vous l'hébergez en interne et avez réglé le DNS pour faire pointer `gitserveur` sur ce serveur, alors vous pouvez utiliser les commandes suivantes telles quelles (en supposant que `monprojet` est un projet existant et comprenant des fichiers) :

```
# Sur l'ordinateur de John
$ cd monproject
$ git init
$ git add .
$ git commit -m 'première validation'
$ git remote add origin git@gitserveur:/opt/git/projet.git
$ git push origin master
```

À présent, les autres utilisateurs peuvent cloner le dépôt et y pousser leurs modifications aussi simplement :

```
$ git clone git@gitserveur:/opt/git/projet.git
$ cd projet
$ vim LISEZMOI
$ git commit -am 'correction du fichier LISEZMOI'
$ git push origin master
```

De cette manière, vous pouvez rapidement mettre en place un serveur Git en lecture/écriture pour une poignée de développeurs.

Il faut aussi noter que pour l'instant tous ces utilisateurs peuvent aussi se connecter au serveur et obtenir un shell en tant qu'utilisateur « git ». Si vous souhaitez restreindre ces droits, il faudra changer le shell pour quelque chose d'autre dans le fichier `passwd`.

Vous pouvez simplement restreindre l'utilisateur *git* à des actions Git avec un shell limité appelé `git-shell` qui est fourni avec Git. Si vous configurez ce shell comme shell de login de l'utilisateur *git*, l'utilisateur *git* ne peut pas avoir de shell normal sur ce serveur. Pour utiliser cette fonction, spécifiez `git-shell` en lieu et place de bash ou csh pour shell de l'utilisateur. Pour faire cela, vous devez d'abord ajouter `git-shell` à `/etc/shells` s'il n'y est pas déjà :

```
$ cat /etc/shells   # voir si `git-shell` est déjà déclaré.  Sinon...
$ which git-shell   # s'assurer que git-shell est installé sur le système
$ sudo vim /etc/shells  # et ajouter le chemin complet vers git-shell
```

Maintenant, vous pouvez éditer le shell de l'utilisateur en utilisant `chsh <utilisateur>` :

```
$ sudo chsh git  # saisir le chemin vers git-shell, souvent : /usr/bin/git-shell
```

À présent, l'utilisateur *git* ne peut plus utiliser la connexion SSH que pour pousser et tirer sur des dépôts Git, il ne peut plus ouvrir un shell. Si vous essayez, vous verrez un rejet de login :

```
$ ssh git@gitserveur
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserveur closed.
```

Maintenant, les commandes réseau Git continueront de fonctionner correctement mais les utilisateurs ne pourront plus obtenir de shell. Comme la sortie l'indique, vous pouvez aussi configurer un répertoire dans le répertoire personnel de l'utilisateur « git » qui va personnaliser légèrement le `git-shell`. Par exemple, vous pouvez restreindre les commandes Git que le serveur accepte ou vous pouvez personnaliser le message que les utilisateurs verront s'ils essaient de se connecter en SSH comme ci-dessus. Lancer `git help shell` pour plus d'informations sur la personnalisation du shell.

Démon (*Daemon*) Git

Dans la suite, nous allons configurer un *daemon* qui servira des dépôts sur le protocole « Git ». C'est un choix répandu pour permettre un accès rapide sans authentification à vos données Git. Souvenez-vous que du fait de l'absence d'authentification, tout ce qui est servi sur ce protocole est public au sein de son réseau.

Mis en place sur un serveur à l'extérieur de votre pare-feu, il ne devrait être utilisé que pour des projets qui sont destinés à être visibles publiquement par le monde entier. Si le serveur est derrière le pare-feu, il peut être utilisé pour des projets avec accès en lecture seule pour un grand nombre d'utilisateurs ou des ordinateurs (intégration continue ou serveur de compilation) pour lesquels vous ne souhaitez pas avoir à gérer des clés SSH.

En tout cas, le protocole Git est relativement facile à mettre en place. Grossièrement, il suffit de lancer la commande suivante en tant que *daemon* :

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` autorise le serveur à redémarrer sans devoir attendre que les anciennes connexions expirent, l'option `--base-path` autorise les utilisateurs à cloner des projets sans devoir spécifier le chemin complet, et le chemin en fin de ligne indique au *daemon* Git l'endroit où chercher des dépôts à exporter. Si vous utilisez un pare-feu, il sera nécessaire de rediriger le port 9418 sur la machine hébergeant le serveur.

Transformer ce processus en *daemon* peut s'effectuer de différentes manières qui dépendent du système d'exploitation sur lequel il est lancé. Sur une machine Ubuntu, c'est un script Upstart. Donc dans le fichier :

```
/etc/event.d/local-git-daemon
```

mettez le script suivant :

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

Par sécurité, ce *daemon* devrait être lancé par un utilisateur n'ayant que des droits de lecture seule sur les dépôts — simplement en créant un nouvel utilisateur « git-ro » qui servira à lancer le *daemon*. Par simplicité, nous le lancerons avec le même utilisateur « git » qui est utilisé par `git-shell`.

Au redémarrage de la machine, votre *daemon* Git démarrera automatiquement et redémarrera s'il meurt. Pour le lancer sans avoir à redémarrer, vous pouvez lancer ceci :

```
initctl start local-git-daemon
```

Sur d'autres systèmes, le choix reste large, allant de `xinetd` à un script de système `sysvinit` ou à tout autre moyen — tant que le programme est démonisé et surveillé.

Ensuite, il faut spécifier à Git quels dépôts sont autorisés en accès non authentifié au moyen du serveur. Dans chaque dépôt concerné, il suffit de créer un fichier appelé `git-daemon-export-ok`.

```
$ cd /chemin/au/projet.git
$ touch git-daemon-export-ok
```

La présence de ce fichier indique à Git que ce projet peut être servi sans authentification.

HTTP intelligent

Nous avons à présent un accès authentifié par SSH et un accès non authentifié par `git://`, mais il existe aussi un protocole qui peut faire les deux à la fois. La configuration d'un HTTP intelligent revient simplement à activer sur le serveur un script CGI livré avec Git qui s'appelle `git-http-backend`. Ce CGI va lire le

chemin et les entêtes envoyés par un `git fetch` ou un `git push` à une URL donnée et déterminer si le client peut communiquer sur HTTP (ce qui est vrai pour tout client depuis la version 1.6.6). Si le CGI détecte que le client est intelligent, il va commencer à communiquer par protocole intelligent, sinon il repassera au comportement du protocole idiot (ce qui le rend de ce fait compatible avec les vieux clients).

Détaillons une installation de base. Nous la réaliserons sur un serveur web Apache comme serveur CGI. Si Apache n'est pas installé sur votre PC, vous pouvez y remédier avec une commande :

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

Cela a aussi pour effet d'activer les modules `mod_cgi`, `mod_alias`, et `mod_env` qui sont nécessaires au fonctionnement du serveur.

Ensuite, nous devons ajouter quelques lignes à la configuration d'Apache pour qu'il lance `git-http-backend` comme gestionnaire de tous les chemins du serveur web sous `/git`.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Si vous ne définissez pas la variable d'environnement `GIT_HTTP_EXPORT_ALL`, Git ne servira aux utilisateurs non authentifiés que les dépôts comprenant le fichier `git-daemon-export-ok`, de la même manière que le *daemon* Git.

Puis, nous allons indiquer à Apache qu'il doit accepter les requêtes sur ce chemin avec quelque chose comme :

```
<Directory "/usr/lib/git-core*">
   Options ExecCGI Indexes
   Order allow,deny
   Allow from all
   Require all granted
</Directory>
```

Enfin, il faut forcer l'authentification pour l'écriture, probablement avec un bloc `Auth` comme celui-ci :

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
</LocationMatch>
```

Il faudra donc créer un fichier `.htaccess` contenant les mots de passe de tous les utilisateurs valides. Voici un exemple d'ajout d'un utilisateur `schacon` au fichier :

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

Il existe des milliers de façons d'authentifier des utilisateurs avec Apache, il suffira d'en choisir une et de la mettre en place. L'exemple ci-dessus n'est que le plus simple. Vous désirerez sûrement gérer tout ceci sous SSL pour que vos données soient chiffrées.

Nous ne souhaitons pas nous appesantir spécifiquement sur la configuration d'Apache, car on peut utiliser un serveur différent ou avoir besoin d'une authentification différente. L'idée générale reste que Git est livré avec un CGI appelé `git-http-backend` qui, après authentification, va gérer toute la négociation pour envoyer et recevoir les données sur HTTP. Il ne gère pas l'authentification par lui-même, mais peut être facilement contrôlé à la couche serveur web qui l'invoque. Cela peut être réalisé avec n'importe quel serveur web gérant le CGI, donc celui que vous connaissez le mieux.

Pour plus d'informations sur la configuration de l'authentification dans Apache, référez-vous à la documentation d'Apache : *http://httpd.apache.org/docs/current/howto/auth.html*

GitWeb

Après avoir réglé les accès de base en lecture/écriture et en lecture seule pour vos projets, vous souhaiterez peut-être mettre en place une interface web simple de visualisation. Git fournit un script CGI appelé GitWeb qui est souvent utilisé à cette fin.

FIGURE 4-1

L'interface web de visualisation Gitweb.

Si vous souhaitez vérifier à quoi GitWeb ressemblerait pour votre projet, Git fournit une commande pour démarrer une instance temporaire de serveur si vous avez un serveur léger tel que `lighttpd` ou `webrick` sur votre système. Sur les machines Linux, `lighttpd` est souvent pré-installé et vous devriez pouvoir le démarrer en tapant `git instaweb` dans votre répertoire de travail. Si vous utilisez un Mac, Ruby est installé de base avec Léopard, donc `webrick` est une meilleure option. Pour démarrer `instaweb` avec un gestionnaire autre que `lighttpd`, vous pouvez le lancer avec l'option `--httpd`.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Cette commande démarre un serveur HTTP sur le port 1234 et lance automatiquement un navigateur Internet qui ouvre la page d'accueil. C'est vraiment très simple. Pour arrêter le serveur, il suffit de lancer la même commande, mais avec l'option `--stop` :

```
$ git instaweb --httpd=webrick --stop
```

Si vous souhaitez fournir l'interface web en permanence sur le serveur pour votre équipe ou pour un projet opensource que vous hébergez, il sera nécessaire d'installer le script CGI pour qu'il soit appelé par votre serveur web. Quelques distributions Linux ont un package `gitweb` qu'il suffira d'installer via `apt` ou `yum`, ce qui est une possibilité. Nous détaillerons tout de même rapidement l'installation manuelle de GitWeb. Premièrement, le code source de Git qui fournit GitWeb est nécessaire pour pouvoir générer un script CGI personnalisé :

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Notez que vous devez indiquer où trouver les dépôts Git au moyen de la variable `GITWEB_PROJECTROOT`. Maintenant, il faut paramétrer dans Apache l'utilisation de CGI pour ce script, en spécifiant un nouveau VirtualHost :

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Une fois de plus, GitWeb peut être géré par tout serveur web capable de prendre en charge CGI ou Perl. La mise en place ne devrait pas être plus difficile avec un autre serveur. Après redémarrage du serveur, vous devriez être capable de visiter `http://gitserveur/` pour visualiser vos dépôts en ligne.

GitLab

GitWeb reste tout de même simpliste. Si vous cherchez un serveur Git plus moderne et complet, il existe quelques solutions libres pertinentes. Comme GitLab est un des plus populaires, nous allons prendre son installation et son utilisation comme exemple. Cette solution est plus complexe que l'option GitWeb et demandera indubitablement plus de maintenance, mais elle est aussi plus complète.

Installation

GitLab est une application web reposant sur une base de données, ce qui rend son installation un peu plus lourde que certains autres serveurs Git. Celle-ci est heureusement très bien documentée et supportée.

GitLab peut s'installer de différentes manières. Pour obtenir rapidement quelque chose qui tourne, vous pouvez télécharger une image de machine virtuelle ou un installateur rapide depuis ***https://bitnami.com/stack/gitlab***, puis configurer plus finement selon vos besoins. Une touche particulière incluse par Bitnami concerne l'écran d'identification (accessible via alt-→) qui vous indique l'adresse IP, l'utilisateur et le mot de passe par défaut de l'instance GitLab installée.

FIGURE 4-2

L'écran d'identification de la machine virtuelle du GitLab de Bitnami.

Pour toute autre méthode, suivez les instructions du readme du *GitLab Community Edition*, qui est consultable à ***https://gitlab.com/gitlab-org/gitlab-ce/***

tree/master. Vous y trouverez une aide pour installer GitLab en utilisant une recette Chef, une machine virtuelle sur Digital Ocean, ou encore via RPM ou DEB (qui, au moment de la rédaction du présent livre sont en bêta). Il existe aussi des guides « non-officiels » pour faire fonctionner GitLab avec des systèmes d'exploitation ou de base données non standards, un script d'installation totalement manuel et d'autres guides couvrant d'autres sujets.

Administration

L'interface d'administration de GitLab passe par le web. Pointez simplement votre navigateur sur le nom d'hôte ou l'adresse IP où GitLab est hébergé et identifiez-vous comme administrateur. L'utilisateur par défaut est `admin@local.host` et le mot de passe par défaut est `5iveL!fe` (qu'il vous sera demandé de changer dès la première connexion). Une fois identifié, cliquez sur l'icône « Admin area » dans le menu en haut à droite.

FIGURE 4-3

L'entrée « Admin area » dans le menu GitLab.

UTILISATEURS

Les utilisateurs dans GitLab sont des comptes qui correspondent à des personnes. Les comptes utilisateurs ne sont pas très complexes ; ce sont principalement des collections d'informations personnelles rattachées à chaque information d'identification. Chaque compte utilisateur fournit un **espace de nommage**, qui est un rassemblement logique des projets appartenant à cet utilisateur. Si l'utilisateur `jane` a un projet appelé `projet`, l'URL du projet est ***http://serveur/jane/projet***.

FIGURE 4-4

L'écran d'administration des utilisateurs GitLab.

Il existe deux manières de supprimer un utilisateur. Bloquer (`Blocking`) un utilisateur l'empêche de s'identifier sur l'instance Gitlab, mais toutes les données sous l'espace de nom de cet utilisateur sont préservées, et les commits signés avec l'adresse courriel de cet utilisateur renverront à son profil.

Détruire (`Destroying`) un utilisateur, par contre, l'efface complètement de la base de données et du système de fichiers. Tous les projets et les données situées dans son espace de nom sont effacés et tous les groupes qui lui appartiennent sont aussi effacés. Il s'agit clairement d'une action plus destructive et permanente, et son usage est assez rare.

GROUPES

Un groupe GitLab est un assemblage de projets, accompagné des informations de droits d'accès à ces projets. Chaque groupe a un espace de nom de projet (de la même manière que les utilisateurs), donc si le groupe `formation` a un projet `matériel`, son URL sera ***http://serveur/formation/matériel***.

FIGURE 4-5

L'écran d'administration des groupes GitLab.

Chaque groupe est associé à des utilisateurs, dont chacun dispose d'un niveau de permissions sur les projets du groupe et sur le groupe lui-même. Ces niveaux s'échelonnent de *invité* : `Guest` (tickets et discussions seulement) à *propriétaire* : `Owner` (contrôle complet du groupe, ses membres et ses projets). Les types de permissions sont trop nombreux pour être énumérés ici, mais GitLab fournit un lien très utile sur son écran d'administration.

PROJETS

Un projet GitLab correspond grossièrement à un dépôt Git unique. Tous les projets appartiennent à un espace de nom unique, que ce soit un utilisateur ou un groupe. Si le projet appartient à un utilisateur, le propriétaire du projet contrôle directement les droits d'accès au projet ; si le projet appartient à un groupe, le niveau de permission de l'utilisateur pour le groupe est aussi pris en compte.

Tous les projets ont un niveau de visibilité qui permet de contrôler qui a accès en lecture aux pages et au dépôt de ce projet. Si un projet est privé (*Private*), l'accès au projet doit être explicitement accordé par le propriétaire du projet à chaque utilisateur. Un projet interne (*Internal*) est visible par tout utilisateur identifié, et un projet public (*Public*) est un projet visible par tout le monde. Notez que ces droits contrôlent aussi bien les accès pour git `fetch` que les accès à l'interface utilisateur web du projet.

CROCHETS (*HOOKS*)

GitLab inclut le support pour les crochets, tant au niveau projet que système. Pour ces deux niveaux, le serveur GitLab lance des requêtes HTTP POST contenant un JSON de description lorsque certains événements précis arrivent. C'est

une excellent moyen de connecter vos dépôts Git et votre instance GitLab avec le reste de vos automatisations de développement, telles que serveurs d'intégration continue, forum de discussion et outils de déploiement.

Usage de base

La première chose à faire avec GitLab est de créer un nouveau projet. Pour cela, il suffit de cliquer sur l'icône + sur la barre d'outils. On vous demande le nom du projet, à quel espace de nom il appartient, et son niveau de visibilité. La plupart des configurations demandées ici ne sont pas permanentes et peuvent être réajustées plus tard grâce à l'interface de paramétrage. Cliquez sur `Create Project` pour achever la création.

Une fois le projet créé, on peut le connecter à un dépôt Git local. Chaque projet est accessible sur HTTPS ou SSH, qui peuvent donc être utilisés pour un dépôt distant. Les URLs sont visibles en haut de la page du projet. Pour un dépôt local existant, cette commande crée un dépôt distant nommé `gitlab` pointant vers l'hébergement distant :

```
$ git remote add gitlab https://serveur/espace_de_nom/projet.git
```

Si vous n'avez pas de copie locale du dépôt, vous pouvez simplement taper ceci :

```
$ git clone https://serveur/espace_de_nom/projet.git
```

L'interface utilisateur web donne accès à différentes vues utiles du dépôt lui-même. La page d'accueil de chaque projet montre l'activité récente et des liens alignés en haut vous mènent aux fichiers du projet et au journal des *commits*.

Coopérer

Le moyen le plus simple de coopérer sur un projet GitLab consiste à donner à un autre utilisateur un accès direct en écriture sur le dépôt Git. Vous pouvez ajouter un utilisateur à un projet en sélectionnant la section `Members` des paramètres du projet et en associant le nouvel utilisateur à un niveau d'accès (les différents niveaux d'accès sont abordés dans **"Groupes"**). En donnant un niveau d'accès `Developer` ou plus à un utilisateur, cet utilisateur peut pousser des *commits* et des branches directement sur le dépôt sans restriction.

Un autre moyen plus découplé de collaborer est d'utiliser des requêtes de tirage (*pull request*). Cette fonction permet à n'importe quel utilisateur qui peut voir le projet d'y contribuer de manière contrôlée. Les utilisateurs avec un accès direct peuvent simplement créer une branche, pousser des *commits* dessus et ouvrir une requête de tirage depuis leur branche vers `master` ou toute autre branche. Les utilisateurs qui n'ont pas la permission de pousser sur un dépôt peuvent en faire un *fork* (créer leur propre copie), pousser des *commits* sur cette copie et ouvrir une requête de tirage depuis leur *fork* vers le projet principal. Ce modèle permet au propriétaire de garder le contrôle total sur ce qui entre dans le dépôt et quand, tout en autorisant les contributions des utilisateurs non fiables.

Les requêtes de fusion (*merge requests*) et les problèmes (*issues*) sont les principaux moyens pour mener des discussions au long cours dans GitLab. Chaque requête de fusion permet une discussion ligne par ligne sur les modifications proposées (qui permettent un sorte de revue de code légère), ainsi qu'un fil de discussion général. Requêtes de fusion et problèmes peuvent être assignés à des utilisateurs ou assemblés en jalons (*milestones*).

Cette section se concentre principalement sur les parties de GitLab dédiées à Git, mais c'est un système assez mature qui fournit beaucoup d'autres fonctions qui peuvent aider votre équipe à coopérer. Parmi celles-ci figurent les wikis, les murs de discussion et des outils de maintenance du système. Un des bénéfices de GitLab est que, une fois le serveur paramétré et en marche, vous n'aurez pas besoin de bricoler un fichier de configuration ou d'accéder au serveur via SSH ; la plupart des tâches générales ou d'administration peuvent se réaliser à travers l'interface web.

Git hébergé

Si vous ne vous ne voulez pas vous investir dans la mise en place de votre propre serveur Git, il reste quelques options pour héberger vos projets Git sur un site externe dédié à l'hébergement. Cette méthode offre de nombreux avantages : un site en hébergement est généralement rapide à créer et facilite le démarrage de projets, et n'implique pas de maintenance et de surveillance de serveur. Même si vous montez et faites fonctionner votre serveur en interne, vous souhaiterez sûrement utiliser un site d'hébergement public pour votre code open source — cela rend généralement plus facile l'accès et l'aide par la communauté.

Aujourd'hui, vous avez à disposition un nombre impressionnant d'options d'hébergement, chacune avec différents avantages et inconvénients. Pour une liste à jour, référez-vous à la page GitHosting sur le wiki principal de Git : ***https://git.wiki.kernel.org/index.php/GitHosting***.

Nous traiterons de l'utilisation de GitHub en détail dans **Chapter 6** du fait que c'est le plus gros hébergement de Git sur Internet et que vous pourriez avoir besoin d'y interagir pour des projets hébergés à un moment, mais il existe aussi d'autres plates-formes d'hébergement si vous ne souhaitez pas mettre en place votre propre serveur Git.

Résumé

Vous disposez de plusieurs moyens de mettre en place un dépôt Git distant pour pouvoir collaborer avec d'autres et partager votre travail.

Gérer votre propre serveur vous donne une grande maîtrise et vous permet de l'installer derrière un pare-feu, mais un tel serveur nécessite généralement une certaine quantité de travail pour l'installation et la maintenance. Si vous placez vos données sur un serveur hébergé, c'est très simple à installer et maintenir. Cependant vous devez pouvoir héberger votre code sur des serveurs tiers et certaines politiques d'organisation ne le permettent pas.

Choisir la meilleure solution ou combinaison de solutions pour votre cas ou celui de votre société ne devrait pas poser de problème.

Git distribué 5

Avec un dépôt distant Git mis en place pour permettre à tous les développeurs de partager leur code, et la connaissance des commandes de base de Git pour une gestion locale, abordons les méthodes de gestion distribuée que Git nous offre.

Dans ce chapitre, vous découvrirez comment travailler dans un environnement distribué avec Git en tant que contributeur ou comme intégrateur. Cela recouvre la manière de contribuer efficacement à un projet et de rendre la vie plus facile au mainteneur du projet ainsi qu'à vous-même, mais aussi en tant que mainteneur, de gérer un projet avec de nombreux contributeurs.

Développements distribués

À la différence des systèmes de gestion de version centralisés (CVCS), la nature distribuée de Git permet une bien plus grande flexibilité dans la manière dont les développeurs collaborent sur un projet. Dans les systèmes centralisés, tout développeur est un nœud travaillant de manière plus ou moins égale sur un concentrateur central. Dans Git par contre, tout développeur est potentiellement un nœud et un concentrateur, c'est-à-dire que chaque développeur peut à la fois contribuer du code vers les autres dépôts et maintenir un dépôt public sur lequel d'autres vont baser leur travail et auquel ils vont contribuer. Cette capacité ouvre une perspective de modes de développement pour votre projet ou votre équipe dont certains archétypes tirant parti de cette flexibilité seront traités dans les sections qui suivent. Les avantages et inconvénients éventuels de chaque mode seront traités. Vous pouvez choisir d'en utiliser un seul ou de mélanger les fonctions de chacun.

Gestion Centralisée

Dans les systèmes centralisés, il n'y a généralement qu'un seul modèle de collaboration, la gestion centralisée. Un concentrateur ou dépôt central accepte le

code et tout le monde doit synchroniser son travail avec. Les développeurs sont des nœuds, des consommateurs du concentrateur, seul endroit où ils se synchronisent.

FIGURE 5-1

Gestion centralisée.

Cela signifie que si deux développeurs clonent depuis le concentrateur et qu'ils introduisent tous les deux des modifications, le premier à pousser ses modifications le fera sans encombre. Le second développeur doit fusionner les modifications du premier dans son dépôt local avant de pousser ses modifications pour ne pas écraser les modifications du premier. Ce concept reste aussi vrai avec Git qu'il l'est avec Subversion (ou tout autre CVCS) et le modèle fonctionne parfaitement dans Git.

Si vous êtes déjà habitué à une gestion centralisée dans votre société ou votre équipe, vous pouvez simplement continuer à utiliser cette méthode avec Git. Mettez en place un dépôt unique et donnez à tous l'accès en poussée. Git empêchera les utilisateurs d'écraser le travail des autres. Supposons que John et Jessica commencent en même temps une tâche. John la termine et pousse ses modifications sur le serveur. Puis Jessica essaie de pousser ses modifications, mais le serveur les rejette. Il lui indique qu'elle tente de pousser des modifications sans avance rapide et qu'elle ne pourra le faire tant qu'elle n'aura pas récupéré et fusionné les nouvelles modifications depuis le serveur. Cette méthode est très intéressante pour de nombreuses personnes car c'est un paradigme avec lequel beaucoup sont familiarisés et à l'aise.

Ce modèle n'est pas limité aux petites équipes. Avec le modèle de branchement de Git, des centaines de développeurs peuvent travailler harmonieusement sur un unique projet au travers de dizaines de branches simultanées.

Mode du gestionnaire d'intégration

Comme Git permet une multiplicité de dépôts distants, il est possible d'envisager un mode de fonctionnement où chaque développeur a un accès en écriture à son propre dépôt public et en lecture à tous ceux des autres. Ce scénario inclut souvent un dépôt canonique qui représente le projet « officiel ». Pour commencer à contribuer au projet, vous créez votre propre clone public du projet et poussez vos modifications dessus. Après, il suffit d'envoyer une demande au mainteneur de projet pour qu'il tire vos modifications dans le dépôt canonique. Il peut ajouter votre dépôt comme dépôt distant, tester vos modifications localement, les fusionner dans sa branche et les pousser vers le dépôt public. Le processus se passe comme ceci (voir **Figure 5-2**) :

1. Le mainteneur du projet pousse vers son dépôt public.
2. Un contributeur clone ce dépôt et introduit des modifications.
3. Le contributeur pousse son travail sur son dépôt public.
4. Le contributeur envoie au mainteneur un e-mail de demande pour tirer ses modifications depuis son dépôt.
5. Le mainteneur ajoute le dépôt du contributeur comme dépôt distant et fusionne les modifications localement.
6. Le mainteneur pousse les modifications fusionnées sur le dépôt principal.

FIGURE 5-2

Le mode du gestionnaire d'intégration.

C'est une gestion très commune sur des sites « échangeurs » tels que GitHub ou GitLab où il est aisé de dupliquer un projet et de pousser ses modifications pour les rendre publiques. Un avantage distinctif de cette approche est qu'il devient possible de continuer à travailler et que le mainteneur du dépôt principal peut tirer les modifications à tout moment. Les contributeurs n'ont pas à attendre le bon vouloir du mainteneur pour incorporer leurs modifications. Chaque acteur peut travailler à son rythme.

Mode dictateur et ses lieutenants

C'est une variante de la gestion multi-dépôt. En général, ce mode est utilisé sur des projets immenses comprenant des centaines de collaborateurs. Un exemple célèbre est le noyau Linux. Des gestionnaires d'intégration gèrent certaines parties du projet. Ce sont les lieutenants. Tous les lieutenants ont un unique gestionnaire d'intégration, le dictateur bienveillant. Le dépôt du dictateur sert de dépôt de référence à partir duquel tous les collaborateurs doivent tirer. Le processus se déroule comme suit (voir **Figure 5-3**) :

1. Les simples développeurs travaillent sur la branche thématique et rebasent leur travail sur master. La branche `master` est celle du dictateur.
2. Les lieutenants fusionnent les branches thématiques des développeurs dans leur propre branche `master`.
3. Le dictateur fusionne les branches master de ses lieutenants dans sa propre branche `master`.
4. Le dictateur pousse sa branche `master` sur le dépôt de référence pour que les développeurs se rebasent dessus.

FIGURE 5-3

Le processus du dictateur bienveillant.

Ce schéma de processus n'est pas très utilisé mais s'avère utile dans des projets très gros ou pour lesquels un ordre hiérarchique existe, car il permet au chef de projet (le dictateur) de déléguer une grande partie du travail et de collecter de grands sous-ensembles de codes à différents points avant de les intégrer.

Résumé

Voilà donc quelques-uns des flux de travail les plus utilisés avec un système distribué tel que Git, mais on voit que de nombreuses variations sont possibles pour mieux correspondre à un mode de gestion réel. À présent que vous avez pu déterminer le mode de gestion qui s'adapte à votre cas, nous allons traiter des exemples spécifiques détaillant comment remplir les rôles principaux constituant chaque mode. Dans le chapitre suivant, nous traiterons de quelques modèles d'activité pour la contribution à un projet.

Contribution à un projet

La principale difficulté à décrire ce processus réside dans l'extraordinaire quantité de variations dans sa réalisation. Comme Git est très flexible, les gens peuvent collaborer de différentes façons et ils le font, et il devient problématique de décrire de manière unique comment devrait se réaliser la contribution à un projet. Chaque projet est légèrement différent. Les variables incluent la taille du corps des contributeurs, le choix du flux de gestion, les accès en validation et la méthode de contribution externe.

La première variable est la taille du corps de contributeurs. Combien de personnes contribuent activement du code sur ce projet et à quelle vitesse ? Dans de nombreux cas, vous aurez deux à trois développeurs avec quelques validations par jour, voire moins pour des projets endormis. Pour des sociétés ou des projets particulièrement grands, le nombre de développeurs peut être de plusieurs milliers, avec des centaines ou des milliers de patchs ajoutés chaque jour. Ce cas est important car avec de plus en plus de développeurs, les problèmes de fusion et d'application de patch deviennent de plus en plus courants. Les modifications soumises par un développeur peuvent être obsolètes ou impossibles à appliquer à cause de changements qui ont eu lieu dans l'intervalle de leur développement, de leur approbation ou de leur application. Comment dans ces conditions conserver son code en permanence synchronisé et ses patchs valides ?

La variable suivante est le mode de gestion utilisé pour le projet. Est-il centralisé avec chaque développeur ayant un accès égal en écriture sur la ligne de développement principale ? Le projet présente-t-il un mainteneur ou un gestionnaire d'intégration qui vérifie tous les patchs ? Tous les patchs doivent-ils subir une revue de pair et une approbation ? Faites-vous partie du processus ? Un système à lieutenants est-il en place et doit-on leur soumettre les modifications en premier ?

La variable suivante est la gestion des accès en écriture. Le mode de gestion nécessaire à la contribution au projet est très différent selon que vous avez ou

non accès au dépôt en écriture. Si vous n'avez pas accès en écriture, quelle est la méthode préférée pour la soumission de modifications ? Y a-t-il seulement une politique en place ? Quelle est la quantité de modifications fournie à chaque fois ? Quelle est la périodicité de contribution ?

Toutes ces questions affectent la manière de contribuer efficacement à un projet et les modes de gestion disponibles ou préférables. Je vais traiter ces sujets dans une série de cas d'utilisation allant des plus simples aux plus complexes. Vous devriez pouvoir construire vos propres modes de gestion à partir de ces exemples.

Guides pour une validation

Avant de passer en revue les cas d'utilisation spécifiques, voici un point rapide sur les messages de validation. La définition et l'utilisation d'une bonne ligne de conduite sur les messages de validation facilitent grandement l'utilisation de Git et la collaboration entre développeurs. Le projet Git fournit un document qui décrit un certain nombre de bonnes pratiques pour créer des *commits* qui serviront à fournir des patchs — le document est accessible dans les sources de Git, dans le fichier `Documentation/SubmittingPatches`.

Premièrement, il ne faut pas soumettre de patchs comportant des erreurs d'espace (caractères espace inutiles en fin de ligne ou entrelacement d'espaces et de tabulations). Git fournit un moyen simple de le vérifier — avant de valider, lancez la commande `git diff --check` qui identifiera et listera les erreurs d'espace.

FIGURE 5-4

Sortie de `git diff --check`.

En lançant cette commande avant chaque validation, vous pouvez vérifier que vous ne commettez pas d'erreurs d'espace qui pourraient ennuyer les autres développeurs.

Ensuite, assurez-vous de faire de chaque validation une modification logiquement atomique. Si possible, rendez chaque modification digeste — ne codez pas pendant un week-end entier sur cinq sujets différents pour enfin les soumettre tous dans une énorme validation le lundi suivant. Même si vous ne validez pas du week-end, utilisez la zone d'index le lundi pour découper votre travail en au moins une validation par problème, avec un message utile par validation. Si certaines modifications touchent au même fichier, essayez d'utiliser `git add --patch` pour indexer partiellement des fichiers (cette fonctionnalité est traitée au chapitre **"Indexation interactive"**). L'instantané final sera identique, que vous utilisiez une validation unique ou cinq petites validations, à condition que toutes les modifications soient intégrées à un moment, donc n'hésitez pas à rendre la vie plus simple à vos compagnons développeurs lorsqu'ils auront à vérifier vos modifications. Cette approche simplifie aussi le retrait ou l'inversion ultérieurs d'une modification en cas de besoin. Le chapitre **"Réécrire l'historique"** décrit justement quelques trucs et astuces de Git pour réécrire l'historique et indexer interactivement les fichiers — utilisez ces outils pour fabriquer un historique propre et compréhensible.

Le dernier point à soigner est le message de validation. S'habituer à écrire des messages de validation de qualité facilite grandement l'emploi et la collaboration avec Git. En règle générale, les messages doivent débuter par une ligne unique d'au plus 50 caractères décrivant concisément la modification, suivie d'une ligne vide, suivie d'une explication plus détaillée. Le projet Git exige que l'explication détaillée inclue la motivation de la modification en contrastant le nouveau comportement par rapport à l'ancien — c'est une bonne règle de rédaction. Une bonne règle consiste aussi à utiliser le présent de l'impératif ou des verbes substantivés dans le message. En d'autres termes, utilisez des ordres. Au lieu d'écrire « J'ai ajouté des tests pour » ou « En train d'ajouter des tests pour », utilisez juste « Ajoute des tests pour » ou « Ajout de tests pour ».

Voici ci-dessous un modèle écrit par Tim Pope :

```
Court résumé des modifications (50 caractères ou moins)

Explication plus détaillée, si nécessaire. Retour à la ligne vers 72
caractères. Dans certains contextes, la première ligne est traitée
comme le sujet d'un courriel et le reste comme le corps. La ligne
vide qui sépare le titre du corps est importante (à moins d'omettre
totalement le corps). Des outils tels que rebase peuvent être gênés
si vous les laissez collés.

Paragraphes supplémentaires après des lignes vides.
```

```
- Les listes à puce sont aussi acceptées

- Typiquement, un tiret ou un astérisque précédés d'un espace unique
  séparés par des lignes vides mais les conventions peuvent varier
```

Si tous vos messages de validation ressemblent à ceci, les choses seront beaucoup plus simples pour vous et les développeurs avec qui vous travaillez. Le projet Git montre des messages de *commit* bien formatés — lancez donc `git log --no-merges` dessus pour voir à quoi ressemble un historique de *commits* avec des messages bien formatés.

Dans les exemples suivants et à travers tout ce livre, par souci de simplification, je ne formaterai pas les messages aussi proprement. J'utiliserai plutôt l'option `-m` de `git commit`. Faites ce que je dis, pas ce que je fais.

Cas d'une petite équipe privée

Le cas le plus probable que vous rencontrerez est celui du projet privé avec un ou deux autres développeurs. Par privé, j'entends code source fermé non accessible au public en lecture. Vous et les autres développeurs aurez accès en poussée au dépôt.

Dans cet environnement, vous pouvez suivre une méthode similaire à ce que vous feriez en utilisant Subversion ou tout autre système centralisé. Vous bénéficiez toujours d'avantages tels que la validation hors-ligne et la gestion de branche et de fusion grandement simplifiée mais les étapes restent similaires. La différence principale reste que les fusions ont lieu du côté client plutôt que sur le serveur au moment de valider. Voyons à quoi pourrait ressembler la collaboration de deux développeurs sur un dépôt partagé. Le premier développeur, John, clone le dépôt, fait une modification et valide localement. Dans les exemples qui suivent, les messages de protocole sont remplacés par `...` pour les raccourcir.

```
# Ordinateur de John
$ git clone john@githost:simplegit.git
Clonage dans 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Eliminer une valeur par défaut invalide'
[master 738ee87] Eliminer une valeur par défaut invalide
 1 files changed, 1 insertions(+), 1 deletions(-)
```

La deuxième développeuse, Jessica, fait la même chose. Elle clone le dépôt et valide une modification :

```
# Ordinateur de Jessica
$ git clone jessica@githost:simplegit.git
Clonage dans 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Ajouter une tache reset'
[master fbff5bc] Ajouter une tache reset
 1 files changed, 1 insertions(+), 0 deletions(-)
```

À présent, Jessica pousse son travail sur le serveur :

```
# Ordinateur de Jessica
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John tente aussi de pousser ses modifications :

```
# Ordinateur de John
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: impossible de pousser des références vers 'john@githost:simplegit.git'
astuce: Les mises à jour ont été rejetées car la pointe de la branche courante est derrière
astuce: son homologue distant. Intégrez les changements distants (par exemple 'git pull ...'
astuce: avant de pousser à nouveau.
astuce: Voir la 'Note à propos des avances rapides' dans 'git push --help' pour plus d'infor
```

John n'a pas le droit de pousser parce que Jessica a déjà poussé dans l'intervalle. Il est très important de comprendre ceci si vous avez déjà utilisé Subversion, parce qu'il faut remarquer que les deux développeurs n'ont pas modifié le même fichier. Quand des fichiers différents ont été modifiés, Subversion réalise cette fusion automatiquement sur le serveur alors que Git nécessite une fusion des modifications locale. John doit récupérer les modifications de Jessica et les fusionner avant d'être autorisé à pousser :

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

À présent, le dépôt local de John ressemble à la figure 5-4.

FIGURE 5-5

Historique divergent de John.

John a une référence aux modifications que Jessica a poussées, mais il doit les fusionner dans sa propre branche avant d'être autorisé à pousser :

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Cette fusion se passe sans problème — l'historique de *commits* de John ressemble à présent à ceci :

FIGURE 5-6

Le dépôt de John après la fusion d'`origin/master`.

Maintenant, John peut tester son code pour s'assurer qu'il fonctionne encore correctement et peut pousser son travail nouvellement fusionné sur le serveur :

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

À la fin, l'historique des *commits* de John ressemble à ceci :

FIGURE 5-7

L'historique de John après avoir poussé sur le serveur origin.

Dans l'intervalle, Jessica a travaillé sur une branche thématique. Elle a créé une branche thématique nommée `prob54` et réalisé trois validations sur cette branche. Elle n'a pas encore récupéré les modifications de John, ce qui donne un historique semblable à ceci :

FIGURE 5-8

La branche thématique de Jessica.

Jessica souhaite se synchroniser sur le travail de John. Elle récupère donc ses modifications :

```
# Ordinateur de Jessica
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Cette commande tire le travail que John avait poussé dans l'intervalle. L'historique de Jessica ressemble maintenant à ceci :

FIGURE 5-9

L'historique de Jessica après avoir récupéré les modifications de John.

Jessica pense que sa branche thématique est prête mais elle souhaite savoir si elle doit fusionner son travail avant de pouvoir pousser. Elle lance `git log` pour s'en assurer :

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700
```

```
    Eliminer une valeur par defaut invalide
```

La syntaxe `prob54..origin/master` est un filtre du journal qui ordonne à Git de ne montrer que la liste des *commits* qui sont sur la seconde branche (dans ce cas `origin/master`) qui ne sont pas sur la première (dans ce cas `prob54`). Nous aborderons cette syntaxe en détail dans **“Plages de *commits*”**.

Pour l’instant, nous pouvons voir que dans le résultat qu’il n’y a qu’un seul *commit* créé par John que Jessica n’a pas fusionné. Si elle fusionne `origin/master`, ce sera le seul commit qui modifiera son travail local.

Maintenant, Jessica peut fusionner sa branche thématique dans sa branche `master`, fusionner le travail de John (`origin/master`)dans sa branche `master`, puis pousser le résultat sur le serveur. Premièrement, elle rebascule sur sa branche `master` pour intégrer son travail :

```
$ git checkout master
Basculement sur la branche 'master'
Votre branche est en retard sur 'origin/master' de 2 commits, et peut être mise à jour en av
```

Elle peut fusionner soit `origin/master` soit `prob54` en premier — les deux sont en avance, mais l’ordre n’importe pas. L’instantané final devrait être identique quel que soit l’ordre de fusion qu’elle choisit. Seul l’historique sera légèrement différent. Elle choisit de fusionner en premier `prob54` :

```
$ git merge issue54
Mise à jour fbff5bc..4af4298
Avance rapide
 LISEZMOI         |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Aucun problème n’apparaît. Comme vous pouvez le voir, c’est une simple avance rapide. Maintenant, Jessica fusionne le travail de John (`origin/master`) :

```
$ git merge origin/master
Fusion automatique de lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Tout a fusionné proprement et l'historique de Jessica ressemble à ceci :

FIGURE 5-10

L'historique de Jessica après avoir fusionné les modifications de John.

Maintenant `origin/master` est accessible depuis la branche `master` de Jessica, donc elle devrait être capable de pousser (en considérant que John n'a pas encore poussé dans l'intervalle) :

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Chaque développeur a validé quelques fois et fusionné les travaux de l'autre avec succès.

FIGURE 5-11

L'historique de Jessica après avoir poussé toutes ses modifications sur le serveur.

C'est un des schémas les plus simples. Vous travaillez pendant quelque temps, généralement sur une branche thématique, et fusionnez dans votre branche `master` quand elle est prête à être intégrée. Quand vous souhaitez partager votre travail, vous récupérez `origin/master` et la fusionnez si elle a changé, puis finalement vous poussez le résultat sur la branche `master` du serveur. La séquence correspond à ceci :

FIGURE 5-12

Séquence générale des événements pour une utilisation simple multi-développeur de Git.

Équipe privée importante

(contribuer, équipe privée gérée Dans le scénario suivant, nous aborderons les rôles de contributeur dans un groupe privé plus grand. Vous apprendrez comment travailler dans un environnement où des petits groupes collaborent sur des fonctionnalités, puis les contributions de chaque équipe sont intégrées par une autre entité.

Supposons que John et Jessica travaillent ensemble sur une première fonctionnalité, tandis que Jessica et Josie travaillent sur une autre. Dans ce cas, l'entreprise utilise un mode d'opération de type « gestionnaire d'intégration » où le travail des groupes est intégré par certains ingénieurs, et la branche `master` du dépôt principal ne peut être mise à jour que par ces ingénieurs. Dans ce scénario, tout le travail est validé dans des branches orientées équipe, et tiré plus tard par les intégrateurs.

Suivons le cheminement de Jessica tandis qu'elle travaille sur les deux nouvelles fonctionnalités, collaborant en parallèle avec deux développeurs différents dans cet environnement. En supposant qu'elle ait cloné son dépôt, elle décide de travailler sur la `fonctionA` en premier. Elle crée une nouvelle branche pour cette fonction et travaille un peu dessus :

```
# Ordinateur de Jessica
$ git checkout -b fonctionA
Basculement sur la nouvelle branche 'fonctionA'
$ vim lib/simplegit.rb
$ git commit -am 'Ajouter une limite à la fonction de log'
[fonctionA 3300904] Ajouter une limite à la fonction de log
 1 files changed, 1 insertions(+), 1 deletions(-)
```

À ce moment, elle a besoin de partager son travail avec John, donc elle pousse les *commits* de sa branche `fonctionA` sur le serveur. Jessica n'a pas le droit de pousser sur la branche `master` — seuls les intégrateurs l'ont — et elle doit donc pousser sur une autre branche pour collaborer avec John :

```
$ git push -u origin fonctionA
...
To jessica@githost:simplegit.git
 * [nouvelle branche]      fonctionA -> fonctionA
```

Jessica envoie un courriel à John pour lui indiquer qu'elle a poussé son travail dans la branche appelée `fonctionA` et qu'il peut l'inspecter. Pendant qu'elle attend le retour de John, Jessica décide de commencer à travailler sur la `fonctionB` avec Josie. Pour commencer, elle crée une nouvelle branche thématique, à partir de la base `master` du serveur :

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b fonctionB origin/master
Basculement sur la nouvelle branche 'fonctionB'
```

À présent, Jessica réalise quelques validations sur la branche `fonctionB` :

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Le dépôt de Jessica ressemble à la figure suivante :

FIGURE 5-13

L'historique initial de Jessica.

Elle est prête à pousser son travail, mais elle reçoit un mail de Josie indiquant qu'une branche avec un premier travail a déjà été poussé sur le serveur en tant que `fonctionBee`. Jessica doit d'abord fusionner ces modifications avec les siennes avant de pouvoir pousser sur le serveur. Elle peut récupérer les modifications de Josie avec `git fetch` :

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [nouvelle branche]      fonctionBee -> origin/fonctionBee
```

Jessica peut à présent fusionner ceci dans le travail qu'elle a réalisé grâce à `git merge` :

```
$ git merge origin/fonctionBee
Fusion automatique de lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Mais il y a un petit problème — elle doit pousser son travail fusionné dans sa branche `fonctionB` sur la branche `fonctionBee` du serveur. Elle peut le faire en spécifiant la branche locale suivie de deux points (:) suivi de la branche distante à la commande `git push` :

```
$ git push -u origin fonctionB:fonctionBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  fonctionB -> fonctionBee
```

Cela s'appelle une *refspec*. Référez-vous à **"La *refspec*"** pour une explication plus détaillée des refspecs Git et des possibilités qu'elles offrent. Notez l'option `-u`. C'est un raccourci pour `--set-upstream`, qui configure les branches pour faciliter les poussées et les tirages plus tard.

Ensuite, John envoie un courriel à Jessica pour lui indiquer qu'il a poussé des modifications sur la branche `fonctionA` et lui demander de les vérifier. Elle lance `git fetch` pour tirer toutes ces modifications :

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  fonctionA   -> origin/fonctionA
```

Elle peut voir ce qui a été modifié avec `git log` :

```
$ git log fonctionA..origin/fonctionA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    largeur du log passee de 25 a 30
```

Finalement, elle fusionne le travail de John dans sa propre branche `fonctionA` :

```
$ git checkout fonctionA
Basculement sur la branche 'fonctionA'
$ git merge origin/fonctionA
Updating 3300904..aad881d
Avance rapide
 lib/simplegit.rb |    10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica veut régler quelques détails. Elle valide donc encore et pousse ses changements sur le serveur :

```
$ git commit -am 'details regles'
[fonctionA ed774b3] details regles
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..ed774b3  fonctionA -> fonctionA
```

L'historique des *commits* de Jessica ressemble à présent à ceci :

FIGURE 5-14

L'historique de Jessica après la validation dans la branche thématique.

Jessica, Josie et John informent les intégrateurs que les branches `fonctionA` et `fonctionB` du serveur sont prêtes pour une intégration dans la branche principale. Après cette intégration dans la branche principale, une synchronisa-

tion apportera les *commits* de fusion, ce qui donnera un historique comme celui-ci :

FIGURE 5-15

L'historique de Jessica après la fusion de ses deux branches thématiques.

De nombreux groupes basculent vers Git du fait de cette capacité à gérer plusieurs équipes travaillant en parallèle, fusionnant plusieurs lignes de développement très tard dans le processus de livraison. La capacité donnée à plusieurs sous-groupes d'équipes de collaborer au moyen de branches distantes sans nécessairement impacter le reste de l'équipe est un grand bénéfice apporté par Git. La séquence de travail qui vous a été décrite ressemble à la figure suivante :

FIGURE 5-16

Une séquence simple de gestion orientée équipe.

Projet public dupliqué

Contribuer à un projet public est assez différent. Il faut présenter le travail au mainteneur d'une autre manière parce que vous n'avez pas la possibilité de mettre à jour directement des branches du projet. Ce premier exemple décrit un mode de contribution via des serveurs Git qui proposent facilement la duplication de dépôt. De nombreux sites proposent cette méthode (dont GitHub, BitBucket, Google Code, repo.or.cz), et de nombreux mainteneurs s'attendent à ce style de contribution. Le chapitre suivant traite des projets qui préfèrent accepter les contributions sous forme de patch via courriel.

Premièrement, vous souhaiterez probablement cloner le dépôt principal, créer une nouvelle branche thématique pour le patch ou la série de patchs que

seront votre contribution, et commencer à travailler. La séquence ressemble globalement à ceci :

```
$ git clone (url)
$ cd projet
$ git checkout -b fonctionA
# (travail)
$ git commit
# (travail)
$ git commit
```

Vous pouvez utiliser `rebase -i` pour réduire votre travail à une seule validation ou pour réarranger les modifications dans des *commits* qui rendront les patchs plus faciles à relire pour le mainteneur — référez-vous à **"Réécrire l'historique"** pour plus d'information sur comment rebaser de manière interactive.

Lorsque votre branche de travail est prête et que vous êtes prêt à la fournir au mainteneur, rendez-vous sur la page du projet et cliquez sur le bouton « Fork » pour créer votre propre projet dupliqué sur lequel vous aurez les droits en écriture. Vous devez alors ajouter l'URL de ce nouveau dépôt en tant que second dépôt distant, dans notre cas nommé `macopie` :

```
$ git remote add macopie (url)
```

Vous devez pousser votre travail sur ce dépôt distant. C'est beaucoup plus facile de pousser la branche sur laquelle vous travaillez sur une branche distante que de fusionner et de pousser le résultat sur le serveur. La raison principale en est que si le travail n'est pas accepté ou s'il est picoré, vous n'aurez pas à faire marche arrière sur votre branche `master`. Si le mainteneur fusionne, rebase ou picore votre travail, vous le saurez en tirant depuis son dépôt :

```
$ git push -u macopie fonctionA
```

Une fois votre travail poussé sur votre dépôt copie, vous devez notifier le mainteneur. Ce processus est souvent appelé une demande de tirage (*pull request*) et vous pouvez la générer soit via le site web — GitHub propose son propre mécanisme qui sera traité au chapitre **Chapter 6** — soit lancer la commande `git request-pull` et envoyer manuellement par courriel le résultat au mainteneur de projet.

La commande `request-pull` prend en paramètres la branche de base dans laquelle vous souhaitez que votre branche thématique soit fusionnée et l'URL du dépôt Git depuis lequel vous souhaitez qu'elle soit tirée, et génère un résumé des modifications que vous demandez à faire tirer. Par exemple, si Jessica envoie à John une demande de tirage et qu'elle a fait deux validations dans la branche thématique qu'elle vient de pousser, elle peut lancer ceci :

```
$ git request-pull origin/master macopie
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        ajout d'une nouvelle fonction

are available in the git repository at:

  git://githost/simplegit.git fonctionA

Jessica Smith (2):
      Ajout d'une limite à la fonction de log
      change la largeur du log de 25 a 30

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Le résultat peut être envoyé au mainteneur — cela lui indique d'où la modification a été branchée, le résumé des validations et d'où tirer ce travail.

Pour un projet dont vous n'êtes pas le mainteneur, il est généralement plus aisé de toujours laisser la branche `master` suivre `origin/master` et de réaliser vos travaux sur des branches thématiques que vous pourrez facilement effacer si elles sont rejetées. Garder les thèmes de travaux isolés sur des branches thématiques facilite aussi leur rebasage si le sommet du dépôt principal a avancé dans l'intervalle et que vos modifications ne s'appliquent plus proprement. Par exemple, si vous souhaitez soumettre un second sujet de travail au projet, ne continuez pas à travailler sur la branche thématique que vous venez de pousser mais démarrez en plutôt une depuis la branche `master` du dépôt principal :

```
$ git checkout -b fonctionB origin/master
# (travail)
$ git commit
$ git push macopie fonctionB
# (email au maintainer)
$ git fetch origin
```

À présent, chaque sujet est contenu dans son propre silo — similaire à une file de patchs — que vous pouvez réécrire, rebaser et modifier sans que les sujets n'interfèrent ou ne dépendent les uns des autres, comme ceci :

FIGURE 5-17

Historique initial des commits avec les modifications de fonctionB.

Supposons que le mainteneur du projet a tiré une poignée d'autres patchs et essayé par la suite votre première branche, mais celle-ci ne s'applique plus proprement. Dans ce cas, vous pouvez rebaser cette branche au sommet de `origin/master`, résoudre les conflits pour le mainteneur et soumettre de nouveau vos modifications :

```
$ git checkout fonctionA
$ git rebase origin/master
$ git push -f macopie fonctionA
```

Cette action réécrit votre historique pour qu'il ressemble à **Figure 5-18**.

FIGURE 5-18

Historique des validations après le travail sur fonctionA.

Comme vous avez rebasé votre branche, vous devez spécifier l'option `-f` à votre commande pour pousser, pour forcer le remplacement de la branche

`fonctionA` sur le serveur par la suite de *commits* qui n'en est pas descendante. Une solution alternative serait de pousser ce nouveau travail dans une branche différente du serveur (appelée par exemple `fonctionAv2`).

Examinons un autre scénario possible : le mainteneur a revu les modifications dans votre seconde branche et apprécie le concept, mais il souhaiterait que vous changiez des détails d'implémentation. Vous en profiterez pour rebaser ce travail sur le sommet actuel de la branche `master` du projet. Vous démarrez une nouvelle branche à partir de la branche `origin/master` courante, y collez les modifications de `fonctionB` en résolvant les conflits, changez l'implémentation et poussez le tout en tant que nouvelle branche :

```
$ git checkout -b fonctionBv2 origin/master
$ git merge --no-commit --squash fonctionB
# (changement d'implémentation)
$ git commit
$ git push macopie fonctionBv2
```

L'option `--squash` prend tout le travail de la branche à fusionner et le colle dans un *commit* sans fusion au sommet de la branche extraite. L'option `--no-commit` indique à Git de ne pas enregistrer automatiquement une validation. Cela permet de reporter toutes les modifications d'une autre branche, puis de réaliser d'autres modifications avant de réaliser une nouvelle validation.

À présent, vous pouvez envoyer au mainteneur un message indiquant que vous avez réalisé les modifications demandées et qu'il peut trouver cette nouvelle mouture sur votre branche `fonctionBv2`.

FIGURE 5-19

Historique des validations après le travail sur fonctionBv2.

Projet public via courriel

De nombreux grands projets ont des procédures établies pour accepter des patchs — il faut vérifier les règles spécifiques à chaque projet qui peuvent vari-

er. Comme il existe quelques gros projets établis qui acceptent les patchs via une liste de diffusion de développement, nous allons éclairer cette méthode d'un exemple.

La méthode est similaire au cas précédent — vous créez une branche thématique par série de patchs sur laquelle vous travaillez. La différence réside dans la manière de les soumettre au projet. Au lieu de dupliquer le projet et de pousser vos soumissions sur votre dépôt, il faut générer des versions courriel de chaque série de *commits* et les envoyer à la liste de diffusion de développement.

```
$ git checkout -b topicA
# (travail)
$ git commit
# (travail)
$ git commit
```

Vous avez à présent deux *commits* que vous souhaitez envoyer à la liste de diffusion. Vous utilisez `git format-patch` pour générer des fichiers au format mbox que vous pourrez envoyer à la liste. Cette commande transforme chaque *commit* en un message courriel dont le sujet est la première ligne du message de validation et le corps contient le reste du message plus le patch correspondant. Un point intéressant de cette commande est qu'appliquer le patch à partir d'un courriel formaté avec `format-patch` préserve toute l'information de validation.

```
$ git format-patch -M origin/master
0001-Ajout-d-une-limite-la-fonction-de-log.patch
0002-change-la-largeur-du-log-de-25-a-30.patch
```

La commande `format-patch` affiche les noms de fichiers de patch créés. L'option `-M` indique à Git de suivre les renommages. Le contenu des fichiers ressemble à ceci :

```
$ cat 0001-Ajout-d-une-limite-la-fonction-de-log.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Ajout d'un limite à la fonction de log

Limite la fonctionnalité de log aux 20 premières lignes

---
```

```
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

Vous pouvez maintenant éditer ces fichiers de patch pour ajouter plus d'informations à destination de la liste de diffusion mais que vous ne souhaitez pas voir apparaître dans le message de validation. Si vous ajoutez du texte entre la ligne `---` et le début du patch (la ligne `diff --git`), les développeurs peuvent le lire mais l'application du patch ne le prend pas en compte.

Pour envoyer par courriel ces fichiers, vous pouvez soit copier leur contenu dans votre application de courriel, soit l'envoyer via une ligne de commande. Le copier-coller cause souvent des problèmes de formatage, spécialement avec les applications « intelligentes » qui ne préservent pas les retours à la ligne et les types d'espace. Heureusement, Git fournit un outil pour envoyer correctement les patchs formatés via IMAP, la méthode la plus facile. Je démontrerai comment envoyer un patch via Gmail qui s'avère être la boîte mail que j'utilise ; vous pourrez trouver des instructions détaillées pour de nombreuses applications de mail à la fin du fichier susmentionné `Documentation/Submitting-Patches` du code source de Git.

Premièrement, il est nécessaire de paramétrer la section `imap` de votre fichier `~/.gitconfig`. Vous pouvez positionner ces valeurs séparément avec une série de commandes `git config`, ou vous pouvez les ajouter manuellement. À la fin, le fichier de configuration doit ressembler à ceci :

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = utilisateur@gmail.com
  pass = m0td3p4ss3
```

```
  port = 993
  sslverify = false
```

Si votre serveur IMAP n'utilise pas SSL, les deux dernières lignes ne sont probablement pas nécessaires et le paramètre `host` commencera par `imap://` au lieu de `imaps://`. Quand c'est fait, vous pouvez utiliser la commande `git imap-send` pour placer la série de patchs dans le répertoire *Drafts* du serveur IMAP spécifié :

```
$ cat *.patch | git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

À ce stade, vous devriez être capable d'aller dans votre dossier « Brouillons », remplacer le champ « Destinataire » par la liste de diffusion à laquelle vous envoyez le patch, peut-être mettre en copie le mainteneur ou la personne responsable de cette section, et l'envoyer.

Vous pouvez aussi envoyer les patchs via un serveur SMTP. Comme précédemment, vous pouvez définir chaque valeur séparément avec une serie de commandes `git config`, ou vous pouvez les ajouter manuellement dans la section « sendmail » dans votre fichier `~/.gitconfig` :

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

Après que ceci soit fait, vous pouvez utiliser `git send-email` pour envoyer vos patchs :

```
$ git send-email *.patch
0001-Ajout-d-une-limite-la-fonction-de-log.patch
0002-change-la-largeur-du-log-de-25-a-30.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Ensuite, Git crache un certain nombre d'informations qui ressemblent à ceci pour chaque patch à envoyer :

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

À présent, vous devriez pouvoir vous rendre dans le répertoire *Drafts*, changer le champ destinataire pour celui de la liste de diffusion, y ajouter optionnellement en copie le mainteneur du projet ou le responsable et l'envoyer.

Résumé

Ce chapitre a traité quelques-unes des méthodes communes de gestion de types différents de projets Git que vous pourrez rencontrer et a introduit un certain nombre de nouveaux outils pour vous aider à gérer ces processus. Dans la section suivante, nous allons voir comment travailler de l'autre côté de la barrière : en tant que mainteneur de projet Git. Vous apprendrez comment travailler comme dictateur bienveillant ou gestionnaire d'intégration.

Maintenance d'un projet

En plus de savoir comment contribuer efficacement à un projet, vous aurez probablement besoin de savoir comment en maintenir un. Cela peut consister à accepter et appliquer les patchs générés via `format-patch` et envoyés par courriel, ou à intégrer des modifications dans des branches distantes de dépôts distants. Que vous mainteniez le dépôt de référence ou que vous souhaitiez aider en vérifiant et approuvant les patchs, vous devez savoir comment accepter les contributions d'une manière limpide pour vos contributeurs et soutenable à long terme pour vous.

Travail dans des branches thématiques

Quand vous vous apprêtez à intégrer des contributions, une bonne idée consiste à les essayer d'abord dans une branche thématique, une branche temporaire spécifiquement créée pour essayer cette nouveauté. De cette manière, il est plus facile de rectifier un patch à part et de le laisser s'il ne fonctionne pas jusqu'à ce que vous disposiez de temps pour y travailler. Si vous créez une simple branche nommée d'après le thème de la modification que vous allez essayer, telle que `ruby_client` ou quelque chose d'aussi descriptif, vous pouvez vous en souvenir simplement plus tard. Le mainteneur du projet Git a l'habitude d'utiliser des espaces de nommage pour ses branches, tels que `sc/ruby_client`, où `sc` représente les initiales de la personne qui a fourni le travail. Comme vous devez vous en souvenir, on crée une branche à partir de `master` de la manière suivante :

```
$ git branch sc/ruby_client master
```

Ou bien, si vous voulez aussi basculer immédiatement dessus, vous pouvez utiliser l'option `checkout -b` :

```
$ git checkout -b sc/ruby_client master
```

Vous voilà maintenant prêt à ajouter les modifications sur cette branche thématique et à déterminer si c'est prêt à être fusionné dans les branches au long cours.

Application des patchs à partir de courriel

courriel, appliquer des patchs depuis) Si vous recevez un patch par courriel et que vous devez l'intégrer dans votre projet, vous devez l'appliquer dans une branche thématique pour l'évaluer. Il existe deux moyens d'appliquer un patch reçu par courriel : `git apply` et `git am`.

APPLICATION D'UN PATCH AVEC APPLY

Si vous avez reçu le patch de quelqu'un qui l'a généré avec la commande `git diff` ou `diff` Unix, vous pouvez l'appliquer avec la commande `git apply`. Si le patch a été sauvé comme fichier `/tmp/patch-ruby-client.patch`, vous pouvez l'appliquer comme ceci :

```
$ git apply /tmp/patch-ruby-client.patch
```

Les fichiers dans votre copie de travail sont modifiés. C'est quasiment identique à la commande `patch -p1` qui applique directement les patchs mais en plus paranoïaque et moins tolérant sur les concordances approximatives. Les ajouts, effacements et renommages de fichiers sont aussi gérés s'ils sont décrits dans le format `git diff`, ce que `patch` ne supporte pas. Enfin, `git apply` fonctionne en mode « applique tout ou refuse tout » dans lequel toutes les modifications proposées sont appliquées si elles le peuvent, sinon rien n'est modifié, là où `patch` peut n'appliquer que partiellement les patchs, laissant le répertoire de travail dans un état intermédiaire. `git apply` est par-dessus tout plus paranoïaque que `patch`. Il ne créera pas une validation à votre place : après l'avoir lancé, vous devrez indexer et valider les modifications manuellement.

Vous pouvez aussi utiliser `git apply` pour voir si un patch s'applique proprement avant de réellement l'appliquer — vous pouvez lancer `git apply --check` avec le patch :

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

S'il n'y pas de message, le patch devrait s'appliquer proprement. Cette commande se termine avec un statut non-nul si la vérification échoue et vous pouvez donc l'utiliser dans des scripts.

APPLICATION D'UN PATCH AVEC `AM`

Si le contributeur est un utilisateur de Git qui a été assez gentil d'utiliser la commande `format-patch` pour générer ses patchs, votre travail sera facilité car le patch contient alors déjà l'information d'auteur et le message de validation. Si possible, encouragez vos contributeurs à utiliser `format-patch` au lieu de `patch` pour générer les patchs qu'ils vous adressent. Vous ne devriez avoir à n'utiliser `git apply` que pour les vrais patchs.

Pour appliquer un patch généré par `format-patch`, vous utilisez `git am`. Techniquement, `git am` s'attend à lire un fichier au format mbox, qui est un format texte simple permettant de stocker un ou plusieurs courriels dans un unique fichier texte. Il ressemble à ceci :

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

C'est le début de ce que la commande `format-patch` affiche, comme vous avez vu dans la section précédente. C'est aussi un format courriel mbox parfaitement valide. Si quelqu'un vous a envoyé par courriel un patch correctement formaté en utilisant `git send-mail` et que vous le téléchargez en format mbox, vous pouvez pointer `git am` sur ce fichier mbox et il commencera à appliquer tous les patchs contenus. Si vous utilisez un client courriel qui sait sauver plusieurs messages au format mbox, vous pouvez sauver la totalité de la série de patchs dans un fichier et utiliser `git am` pour les appliquer tous en une fois.

Néanmoins, si quelqu'un a déposé un fichier de patch généré via `format-patch` sur un système de suivi de faits techniques ou quelque chose de similaire, vous pouvez toujours sauvegarder le fichier localement et le passer à `git am` pour l'appliquer :

```
$ git am 0001-limit-log-function.patch
Application : add limit to log function
```

Vous remarquez qu'il s'est appliqué proprement et a créé une nouvelle validation pour vous. L'information d'auteur est extraite des en-têtes `From` et `Date` tandis que le message de validation est repris du champ `Subject` et du corps (avant le patch) du message. Par exemple, si le patch est appliqué depuis le fichier mbox ci-dessus, la validation générée ressemblerait à ceci :

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function

   Limit log functionality to the first 20
```

L'information `Commit` indique la personne qui a appliqué le patch et la date d'application. L'information `Author` indique la personne qui a créé le patch et la date de création.

Il reste la possibilité que le patch ne s'applique pas proprement. Peut-être votre branche principale a-t'elle déjà trop divergé de la branche sur laquelle le patch a été construit, ou peut-être que le patch dépend d'un autre patch qui n'a pas encore été appliqué. Dans ce cas, le processus de `git am` échouera et vous demandera ce que vous souhaitez faire :

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Application : seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Le patch a échoué à 0001.
Lorsque vous aurez résolu ce problème, lancez "git am --continue".
Si vous préférez sauter ce patch, lancez "git am --skip" à la place.
Pour restaurer la branche d'origine et stopper le patchage, lancez
"git am --abort".
```

Cette commande introduit des marqueurs de conflit dans tous les fichiers qui ont généré un problème, de la même manière qu'un conflit de fusion ou de rebasage. Vous pouvez résoudre les problèmes de manière identique — éditez le fichier pour résoudre les conflits, indexez le nouveau fichier, puis lancez `git am --resolved` ou `git am --continue` pour continuer avec le patch suivant :

```
$ (correction du fichier)
$ git add ticgit.gemspec
$ git am --continue
Applying: seeing if this helps the gem
```

Si vous souhaitez que Git essaie de résoudre les conflits avec plus d'intelligence, vous pouvez passer l'option `-3` qui demande à Git de tenter une fusion à trois sources. Cette option n'est pas active par défaut parce qu'elle ne fonctionne pas si le *commit* sur lequel le patch indique être basé n'existe pas dans votre dépôt. Si par contre, le patch est basé sur un *commit* public, l'option `-3` est généralement beaucoup plus fine pour appliquer des patchs conflictuels :

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

```
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

Dans ce cas, je cherchais à appliquer un patch qui avait déjà été intégré. Sans l'option `-3`, cela aurait ressemblé à un conflit.

Si vous appliquez des patchs à partir d'un fichier mbox, vous pouvez aussi lancer la commande `am` en mode interactif qui s'arrête à chaque patch trouvé et vous demande si vous souhaitez l'appliquer :

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

C'est agréable si vous avez un certain nombre de patchs sauvegardés parce que vous pouvez voir les patchs pour vous rafraîchir la mémoire et ne pas les appliquer s'ils ont déjà été intégrés.

Quand tous les patchs pour votre sujet ont été appliqués et validés dans votre branche, vous pouvez choisir si et comment vous souhaitez les intégrer dans une branche au long cours.

Vérification des branches distantes

Si votre contribution a été fournie par un utilisateur de Git qui a mis en place son propre dépôt public sur lequel il a poussé ses modifications et vous a envoyé l'URL du dépôt et le nom de la branche distante, vous pouvez les ajouter en tant que dépôt distant et réaliser les fusions localement.

Par exemple, si Jessica vous envoie un courriel indiquant qu'elle a une nouvelle fonctionnalité géniale dans la branche `ruby-client` de son dépôt, vous pouvez la tester en ajoutant le dépôt distant et en tirant la branche localement :

```
$ git remote add jessica git://github.com/jessica/monproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Si elle vous envoie un autre mail indiquant une autre branche contenant une autre fonctionnalité géniale, vous pouvez la récupérer et la tester simplement à partir de votre référence distante.

C'est d'autant plus utile si vous travaillez en continu avec une personne. Si quelqu'un n'a qu'un seul patch à contribuer de temps en temps, l'accepter via courriel peut s'avérer moins consommateur en temps de préparation du serveur public, d'ajout et retrait de branches distantes juste pour tirer quelques patchs. Vous ne souhaiteriez sûrement pas devoir gérer des centaines de dépôts distants pour intégrer à chaque fois un ou deux patchs. Néanmoins, des scripts et des services hébergés peuvent rendre cette tâche moins ardue. Cela dépend largement de votre manière de développer et de celle de vos contributeurs.

Cette approche a aussi l'avantage de vous fournir l'historique des validations. Même si vous pouvez rencontrer des problèmes de fusion légitimes, vous avez l'information dans votre historique de la base ayant servi pour les modifications contribuées. La fusion à trois sources est choisie par défaut plutôt que d'avoir à spécifier l'option `-3` en espérant que le patch a été généré à partir d'un instantané public auquel vous auriez accès.

Si vous ne travaillez pas en continu avec une personne mais souhaitez tout de même tirer les modifications de cette manière, vous pouvez fournir l'URL du dépôt distant à la commande `git pull`. Cela permet de réaliser un tirage unique sans sauver l'URL comme référence distante :

```
$ git pull https://github.com/pourunefois/projet
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

Déterminer les modifications introduites

Vous avez maintenant une branche thématique qui contient les contributions. À partir de là, vous pouvez déterminer ce que vous souhaitez en faire. Cette section revisite quelques commandes qui vont vous permettre de faire une revue de ce que vous allez exactement introduire si vous fusionnez dans la branche principale.

Faire une revue de tous les *commits* dans cette branche s'avère souvent d'une grande aide. Vous pouvez exclure les *commits* de la branche `master` en ajoutant l'option `--not` devant le nom de la branche. C'est équivalent au format `master..contrib` utilisé plus haut. Par exemple, si votre contributeur vous envoie deux patchs et que vous créez une branche appelée `contrib` et y appliquez ces patchs, vous pouvez lancer ceci :

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
```

```
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

Pour visualiser les modifications que chaque **commit** introduit, souvenez-vous que vous pouvez passer l'option `-p` à `git log` et elle ajoutera le diff introduit à chaque *commit*.

Pour visualiser un diff complet de ce qui arriverait si vous fusionniez cette branche thématique avec une autre branche, vous pouvez utiliser un truc bizarre pour obtenir les résultats corrects. Vous pourriez penser à lancer ceci :

```
$ git diff master
```

Cette commande affiche un diff mais elle peut être trompeuse. Si votre branche `master` a avancé depuis que vous avez créé la branche thématique, vous obtiendrez des résultats apparemment étranges. Cela arrive parce que Git compare directement l'instantané de la dernière validation sur la branche thématique et celui de la dernière validation sur la branche `master`. Par exemple, si vous avez ajouté une ligne dans un fichier sur la branche `master`, une comparaison directe donnera l'impression que la branche thématique va retirer cette ligne.

Si `master` est un ancêtre directe de la branche thématique, ce n'est pas un problème. Si les deux historiques ont divergé, le diff donnera l'impression que vous ajoutez toutes les nouveautés de la branche thématique et retirez tout ce qui a été fait depuis dans la branche `master`.

Ce que vous souhaitez voir en fait, ce sont les modifications ajoutées sur la branche thématique — le travail que vous introduirez si vous fusionnez cette branche dans `master`. Vous obtenez ce résultat en demandant à Git de comparer le dernier instantané de la branche thématique avec son ancêtre commun à la branche `master` le plus récent.

Techniquement, c'est réalisable en déterminant exactement l'ancêtre commun et en lançant la commande `diff` dessus :

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Néanmoins, comme ce n'est pas très commode, Git fournit un raccourci pour réaliser la même chose : la syntaxe à trois points. Dans le contexte de la commande `diff`, vous pouvez placer trois points après une autre branche pour réaliser un `diff` entre le dernier instantané de la branche sur laquelle vous vous trouvez et son ancêtre commun avec une autre branche :

```
$ git diff master...contrib
```

Cette commande ne vous montre que les modifications que votre branche thématique a introduites depuis son ancêtre commun avec master. C'est une syntaxe très simple à retenir.

Intégration des contributions

Lorsque tout le travail de votre branche thématique est prêt à être intégré dans la branche principale, il reste à savoir comment le faire. De plus, il faut connaître le mode de gestion que vous souhaitez pour votre projet. Vous avez de nombreux choix et je vais en traiter quelques-uns.

MODES DE FUSION

Un mode simple fusionne votre travail dans la branche `master`. Dans ce scénario, vous avez une branche `master` qui contient le code stable. Quand vous avez des modifications prêtes dans une branche thématique, vous la fusionnez dans votre branche `master` puis effacez la branche thématique, et ainsi de suite. Si vous avez un dépôt contenant deux branches nommées `ruby_client` et `php_client` qui ressemble à **Figure 5-20** et que vous fusionnez `ruby_client` en premier, suivi de `php_client`, alors votre historique ressemblera à la fin à **Figure 5-21**.

FIGURE 5-20

Historique avec quelques branches thématiques.

FIGURE 5-21

Après fusion des branches thématiques.

C'est probablement le mode le plus simple mais cela peut s'avérer problématique si vous avez à gérer des dépôts ou des projets plus gros pour lesquels vous devez être circonspect sur ce que vous acceptez.

Si vous avez plus de développeurs ou un projet plus important, vous souhaiterez probablement utiliser un cycle de fusion à deux étapes. Dans ce scénario, vous avez deux branches au long cours, `master` et `develop`, dans lequel vous déterminez que `master` est mis à jour seulement lors d'une version vraiment stable et tout le nouveau code est intégré dans la branche `develop`. Vous poussez régulièrement ces deux branches sur le dépôt public. Chaque fois que vous avez une nouvelle branche thématique à fusionner (**Figure 5-22**), vous la fusionnez dans `develop` (**Figure 5-23**). Puis, lorsque vous étiquetez une version majeure, vous mettez `master` à niveau avec l'état stable de `develop` en avance rapide (**Figure 5-24**).

FIGURE 5-22

Avant la fusion d'une branche thématique.

FIGURE 5-23

Après la fusion d'une branche thématique.

FIGURE 5-24

Après une publication d'une branche thématique.

Ainsi, lorsque l'on clone le dépôt de votre projet, on peut soit extraire la branche `master` pour construire la dernière version stable et mettre à jour fac-

ilement ou on peut extraire la branche `develop` qui représente le nec plus ultra du développement.

Vous pouvez aussi continuer ce concept avec une branche d'intégration où tout le travail est fusionné. Alors, quand la base de code sur cette branche est stable et que les tests passent, vous la fusionnez dans la branche `develop`. Quand cela s'est avéré stable pendant un certain temps, vous mettez à jour la branche `master` en avance rapide.

GESTIONS AVEC NOMBREUSES FUSIONS

Le projet Git dispose de quatre branches au long cours : `master`, `next`, `pu` (*proposed updates* : propositions) pour les nouveaux travaux et `maint` pour les backports de maintenance. Quand une nouvelle contribution est proposée, elle est collectée dans des branches thématiques dans le dépôt du mainteneur d'une manière similaire à ce que j'ai décrit (**Figure 5-25**). À ce point, les fonctionnalités sont évaluées pour déterminer si elles sont stables et prêtes à être consommées ou si elles nécessitent un peaufinage. Si elles sont stables, elles sont fusionnées dans `next` et cette branche est poussée sur le serveur public pour que tout le monde puisse essayer les fonctionnalités intégrées ensemble.

FIGURE 5-25

Série complexe de branches thématiques contribuées en parallèle.

Si les fonctionnalités nécessitent encore du travail, elles sont fusionnées plutôt dans `pu`. Quand elles sont considérées comme totalement stables, elles sont re-fusionnées dans `master` et sont alors reconstruites à partir des fonctionnalités qui résidaient dans `next` mais n'ont pu intégrer `master`. Cela signifie que

`master` évolue quasiment toujours en mode avance rapide, tandis que `next` est rebasé assez souvent et pu est rebasé encore plus souvent :

FIGURE 5-26

Fusion des branches thématiques dans les branches à long terme.

Quand une branche thématique a finalement été fusionnée dans `master`, elle est effacée du dépôt. Le projet Git a aussi une branche `maint` qui est créée à partir de la dernière version pour fournir des patchs correctifs en cas de besoin de version de maintenance. Ainsi, quand vous clonez le dépôt de Git, vous avez quatre branches disponibles pour évaluer le projet à différentes étapes de développement, selon le niveau de développement que vous souhaitez utiliser ou pour lequel vous souhaitez contribuer. Le mainteneur a une gestion structurée qui lui permet d'évaluer et sélectionner les nouvelles contributions.

GESTION PAR REBASAGE ET SÉLECTION DE *COMMIT*

D'autres mainteneurs préfèrent rebaser ou sélectionner les contributions sur le sommet de la branche `master`, plutôt que les fusionner, de manière à conserver un historique à peu près linéaire. Lorsque plusieurs modifications sont présentes dans une branche thématique et que vous souhaitez les intégrer, vous vous placez sur cette branche et vous lancer la commande `rebase` pour reconstruire les modifications à partir du sommet courant de la branche `master` (ou `develop`, ou autre). Si cela fonctionne correctement, vous pouvez faire une avance rapide sur votre branche `master` et vous obtenez finalement un historique de projet linéaire.

L'autre moyen de déplacer des modifications introduites dans une branche vers une autre consiste à les sélectionner ou les picorer (`cherry-pick`). Un picorage dans Git ressemble à un rebasage appliqué à un *commit* unique. Cela consiste à prendre le patch qui a été introduit lors d'une validation et à essayer de l'appliquer sur la branche sur laquelle on se trouve. C'est très utile si on a un certain nombre de *commits* sur une branche thématique et que l'on veut n'en intégrer qu'un seul, ou si on n'a qu'un *commit* sur une branche thématique et qu'on préfère le sélectionner plutôt que de lancer `rebase`. Par exemple, supposons que vous ayez un projet ressemblant à ceci :

FIGURE 5-27

Historique d'exemple avant une sélection.

Si vous souhaitez tirer le *commit* `e43a6` dans votre branche `master`, vous pouvez lancer :

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

La même modification que celle introduite en `e43a6` est tirée mais vous obtenez une nouvelle valeur de SHA-1 car les dates d'application sont différentes. À présent, votre historique ressemble à ceci :

FIGURE 5-28

Historique après sélection d'un commit dans une branche thématique.

Maintenant, vous pouvez effacer votre branche thématique et abandonner les *commits* que vous n'avez pas tirés dans `master`.

RERERE

Si vous fusionnez et rebasez beaucoup ou si vous maintenez une branche au long cours, la fonctionnalité appelée « rerere » peut s'avérer utile.

Rerere signifie « *ré* utiliser les *ré* solutions en *re* gistrées » (" *reuse recorded resolution* ") - c'est un moyen de raccourcir les résolutions manuelles de conflit. Quand rerere est actif, Git va conserver un jeu de couples d'images pré et post fusion des fichiers ayant présenté des conflits, puis s'il s'aperçoit qu'un conflit ressemble à une de ces résolutions, il va utiliser la même stratégie sans rien vous demander.

Cette fonctionnalité se traite en deux phases : une étape de configuration et une commande. L'étape de configuration est `rerere.enabled` qui active la fonction et qu'il est facile de placer en config globale :

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

Ensuite, quand vous fusionnez en résolvant des conflits, la résolution sera enregistrée dans le cache pour un usage futur.

Si besoin, vous pouvez interagir avec le cache rerere au moyen de la commande `git rerere`.

Quand elle est invoquée telle quelle, Git vérifie sa base de données de résolutions et essaie de trouver une correspondance avec les conflits en cours et les résout (bien que ce soit automatique si `rerere.enabled` est à `true`). Il existe

aussi des sous-commandes permettant de voir ce qui sera enregistré, d'effacer du cache une résolution spécifique ou d'effacer entièrement le cache. rerere est traité plus en détail dans **"Rerere"**.

Étiquetage de vos publications

Quand vous décidez de créer une publication de votre projet, vous souhaiterez probablement étiqueter le projet pour pouvoir recréer cette version dans le futur. Vous pouvez créer une nouvelle étiquette (*tag*) telle que décrite dans **Chapter 2**. Si vous décidez de signer l'étiquette en tant que mainteneur, la commande ressemblera à ceci :

```
$ git tag -s v1.5 -m 'mon etiquette v1.5 signée'
Une phrase secrète est nécessaire pour déverrouiller la clef secrète de
l'utilisateur : "Scott Chacon <schacon@gmail.com>"
clé DSA de 1024 bits, identifiant F721C45A, créée le 2009-02-09
```

Si vous signez vos étiquettes, vous rencontrerez le problème de la distribution de votre clé publique PGP permettant de vérifier la signature. Le mainteneur du projet Git a résolu le problème en incluant la clé publique comme blob dans le dépôt et en ajoutant une étiquette qui pointe directement sur ce contenu. Pour faire de même, vous déterminez la clé de votre trousseau que vous voulez publier en lançant `gpg --list-keys` :

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Ensuite, vous pouvez importer la clé directement dans la base de données Git en l'exportant de votre trousseau et en la redirigeant dans `git hash-object` qui écrit un nouveau blob avec son contenu dans Git et vous donne en sortie le SHA-1 du blob :

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

À présent, vous avez le contenu de votre clé dans Git et vous pouvez créer une étiquette qui pointe directement dessus en spécifiant la valeur SHA-1 que la commande `hash-object` vous a fournie :

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Si vous lancez `git push --tags`, l'étiquette `maintainer-pgp-pub` sera partagée publiquement. Un tiers pourra vérifier une étiquette après import direct de votre clé publique PGP, en extrayant le blob de la base de donnée et en l'important dans GPG :

```
$ git show maintainer-pgp-pub | gpg --import
```

Il pourra alors utiliser cette clé pour vérifier vos étiquettes signées. Si de plus, vous incluez des instructions d'utilisation pour la vérification de signature dans le message d'étiquetage, l'utilisateur aura accès à ces informations en lançant la commande `git show <étiquette>`.

Génération d'un nom de révision

Comme Git ne fournit pas par nature de nombres croissants tels que « r123 » à chaque validation, la commande `git describe` permet de générer un nom humainement lisible pour chaque *commit*. Git concatène le nom de l'étiquette la plus proche, le nombre de validations depuis cette étiquette et un code SHA-1 partiel du *commit* que l'on cherche à définir :

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

De cette manière, vous pouvez exporter un instantané ou le construire et le nommer de manière intelligible. En fait, si Git est construit à partir du source cloné depuis le dépôt Git, `git --version` vous donne exactement cette valeur. Si vous demandez la description d'un instantané qui a été étiqueté, le nom de l'étiquette est retourné.

La commande `git describe` repose sur les étiquettes annotées (étiquettes créées avec les options `-a` ou `-s`). Les étiquettes de publication doivent donc être créées de cette manière si vous souhaitez utiliser `git describe` pour garantir que les *commits* seront décrits correctement. Vous pouvez aussi utiliser ces noms comme cible lors d'une extraction ou d'une commande `show`, bien

qu'ils reposent sur le SHA-1 abrégé et pourraient ne pas rester valides indéfiniment. Par exemple, le noyau Linux a sauté dernièrement de 8 à 10 caractères pour assurer l'unicité des objets SHA-1 et les anciens noms `git describe` sont par conséquent devenus invalides.

Préparation d'une publication

Maintenant, vous voulez publier une version. Une des étapes consiste à créer une archive du dernier instantané de votre code pour les malheureux qui n'utilisent pas Git. La commande dédiée à cette action est `git archive` :

```
$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Lorsqu'on ouvre l'archive, on obtient le dernier instantané du projet sous un répertoire `projet`. On peut aussi créer une archive au format zip de manière similaire en passant l'option `--format=zip` à la commande `git archive` :

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Voilà deux belles archives tar.gz et zip de votre projet prêtes à être téléchargées sur un site web ou envoyées par courriel.

Shortlog

Il est temps d'envoyer une annonce à la liste de diffusion des nouveautés de votre projet. Une manière simple d'obtenir rapidement une sorte de liste des modifications depuis votre dernière version ou courriel est d'utiliser la commande `git shortlog`. Elle résume toutes les validations dans l'intervalle que vous lui spécifiez. Par exemple, ce qui suit vous donne un résumé de toutes les validations depuis votre dernière version si celle-ci se nomme v1.0.1 :

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
```

```
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

Vous obtenez ainsi un résumé clair de toutes les validations depuis v1.0.1, regroupées par auteur, prêt à être envoyé sur la liste de diffusion.

Résumé

Vous devriez à présent vous sentir à l'aise pour contribuer à un projet avec Git, mais aussi pour maintenir votre propre projet et intégrer les contributions externes. Félicitations, vous êtes un développeur Git efficace ! Au prochain chapitre, vous découvrirez des outils plus puissants pour gérer des situations complexes, qui feront de vous un maître de Git.

GitHub 6

GitHub est l'un des plus grands hébergeurs de dépôts Git et constitue le point central pour la collaboration de millions de développeurs et de projets. Une grande partie des dépôts Git publics sont hébergés sur GitHub, et de nombreux projets open-source l'utilisent pour l'hébergement Git, le suivi des erreurs, la revue de code et d'autres choses. Donc, bien que ce ne soit pas un sous-ensemble direct du projet open source Git, il est très probable que vous souhaiterez ou aurez besoin d'interagir avec GitHub à un moment ou à un autre dans votre utilisation professionnelle de Git.

Ce chapitre présente la façon d'utiliser GitHub de manière efficace. Nous traiterons la création et la gestion d'un compte utilisateur, la création et l'utilisation de dépôts Git, les processus courants pour contribuer aux projets ou pour accepter des contributions, l'interface de programmation de GitHub ainsi qu'un grand nombre d'astuces pour vous simplifier la vie.

Si vous n'êtes pas intéressé par l'utilisation de GitHub pour héberger vos projets personnels ou pour collaborer à d'autres projets hébergés sur GitHub, vous pouvez sans problème passer directement à **Chapter 7**.

MODIFICATION DE L'INTERFACE

À l'instar de nombreux sites Web actifs, GitHub changera tôt ou tard les éléments de l'interface utilisateur par rapport aux captures d'écran présentées ici. Heureusement, l'idée générale de l'objectif des actions reste valable, mais si vous souhaitez voir des versions plus à jour de ces captures, les versions en ligne de ce livre peuvent présenter des captures plus récentes.

Configuration et paramétrage d'un compte

La première chose à faire consiste à créer un compte utilisateur gratuit. Allez tout simplement sur ***https://github.com***, choisissez un nom d'utilisateur qui

n'est pas déjà pris et saisissez une adresse électronique et un mot de passe, puis cliquez sur le gros bouton vert « Sign up for GitHub » (S'inscrire sur GitHub).

FIGURE 6-1

Le formulaire d'inscription de GitHub.

La deuxième chose que vous verrez est la page des tarifs pour des projets améliorés mais il vaut mieux ignorer cela pour l'instant. GitHub vous envoie un courriel pour vérifier l'adresse fournie. Suivez les instructions mentionnées, c'est très important (comme nous allons le voir plus tard).

> Vous avez accès à toutes les fonctionnalités de GitHub avec un compte gratuit, à la condition que tous vos projets soient entièrement publics (tout le monde peut y accéder en lecture). Les projets payant de GitHub comprennent un nombre défini de projets privés mais nous ne parlerons pas de cela dans ce livre.

En cliquant sur le logo Octocat (logo en forme de chat) dans le coin supérieur gauche de l'écran, vous accéderez à votre tableau de bord. Vous êtes maintenant prêt à utiliser GitHub.

Accès par SSH

Pour l'instant, vous avez la possibilité de vous connecter à des dépôts Git en utilisant le protocole `https://` et de vous identifier au moyen de votre nom d'utilisateur et de votre mot de passe. Cependant, pour simplement cloner des projets publics, il n'est même pas nécessaire de créer un compte - le compte que nous venons de créer devient utile pour commencer à dupliquer (*fork*) un projet ou pour pousser sur ces dépôts plus tard.

Si vous préférez utiliser des serveurs distants en SSH, vous aurez besoin de renseigner votre clé publique. Si vous n'en possédez pas déjà une, référez-vous à **"Génération des clés publiques SSH"**. Accédez aux paramètres de votre compte en utilisant le lien en haut à droite de la fenêtre :

FIGURE 6-2

Lien vers « Account settings » (paramètres du compte).

Sélectionnez ensuite la section « SSH keys » (clés SSH) sur le côté gauche.

FIGURE 6-3

Lien vers « SSH keys » (clés SSH).

Ensuite, cliquez sur le bouton « Add an SSH key » (ajouter une clé SSH), donnez un nom à votre clé, copiez le contenu du fichier de clé publique `~/.ssh/id_rsa.pub` (ou autre si vous l'avez appelé différemment) dans la zone de texte et cliquez sur « Add key » (ajouter la clé).

Assurez-vous de choisir un nom facile à retenir pour votre clé SSH. Vous pouvez donner un nom à chacune de vos clés (par ex. : « mon portable » ou « compte travail ») de façon à la retrouver facilement si vous devez la révoquer plus tard.

Votre Avatar

Ensuite, si vous le souhaitez, vous pouvez remplacer l'avatar généré pour vous par une image de votre choix. Sélectionnez la section « Profile » (profil) (au dessus de la section « SSH Keys ») et cliquez sur « Upload new picture » (télécharger une nouvelle image).

FIGURE 6-4

Lien vers « Profile » (profil).

Après avoir sélectionné une image sur votre disque dur, il vous est possible de la recadrer.

FIGURE 6-5

Recadrage de l'avatar

À présent, toutes vos interventions sur le site seront agrémentées de votre avatar au côté de votre nom d'utilisateur.

S'il se trouve que vous avez déposé un avatar sur le service populaire Gravatar (souvent utilisé pour les comptes Wordpress), cet avatar sera utilisé par défaut et vous n'avez pas à exécuter cette étape.

Vos adresses électroniques

Github utilise les adresses électroniques pour faire correspondre les *commits* Git aux utilisateurs. Si vous utilisez plusieurs adresses électroniques dans vos *commits* et que vous souhaitez que GitHub les relie correctement, vous devez ajouter toutes les adresses que vous avez utilisées dans la section « Emails » (adresses électroniques) de la section d'administration.

FIGURE 6-6

Ajout d'adresses électroniques

Sur **Figure 6-6** nous pouvons voir certains états possibles. L'adresse du haut est vérifiée et définie comme adresse principale, c'est-à-dire que ce sera l'adresse utilisée pour vous envoyer toutes les notifications. La seconde adresse est vérifiée et peut donc aussi être définie comme adresse principale si on l'échange avec la première. La dernière adresse est non vérifiée, ce qui signifie que vous ne pouvez pas en faire votre adresse principale. Si GitHub détecte une de ces adresses dans des messages de validation dans n'importe quel dépôt du site, il les reliera à votre compte utilisateur.

Authentification à deux facteurs

Enfin, pour plus de sécurité, vous devriez assurément paramétrer une authentification à deux facteurs ou « 2FA » (*2 Factor Authentication*). L'authentification à deux facteurs est un mécanisme d'authentification qui est devenu très populaire récemment pour réduire les risques de corruption de votre compte si votre mot de passe est dérobé. Une fois activée, GitHub vous demandera deux méthodes différentes d'authentification, de sorte que si l'une devait être compromise, un attaquant ne pourrait tout de même pas accéder à votre compte.

Vous pouvez trouver les réglages de l'authentification à deux facteurs dans la section « Security » (Sécurité) de la section d'administration.

FIGURE 6-7

2FA dans la section « Security » (Sécurité)

Si vous cliquez sur le bouton « Set up two-factor authentication » (paramétrage de l'authentification à deux facteurs), vous serez redirigé vers une page de configuration sur laquelle vous pourrez choisir d'utiliser une application de téléphone mobile pour générer votre code secondaire (un « mot de passe à usage unique basé sur la date ») ou bien de vous faire envoyer un code GitHub par SMS chaque fois que vous avez besoin de vous identifier.

Après avoir choisi votre méthode préférée et suivi les instructions pour activer 2FA, votre compte sera un peu plus sécurisé et vous devrez fournir un code supplémentaire en plus de votre mot de passe quand vous vous identifierez sur GitHub.

Contribution à un projet

Après avoir configuré votre compte, examinons comment contribuer à un projet existant.

Duplication des projets

Si vous souhaitez contribuer à un projet existant sur lequel vous n'avez pas le droit de pousser, vous pouvez dupliquer (*fork*) ce projet. Cela signifie que GitHub va faire pour vous une copie personnelle du projet. Elle se situe dans votre espace de nom et vous pouvez pousser dessus.

> Historiquement, le terme « *fork* » transmet une idée négative, qui s'apparente à l'idée que quelqu'un mène un projet open-source vers une direction différente, créant un projet concurrent de l'original et divisant les forces de contributions. Au sein de GitHub, un « *fork* » constitue une simple copie d'un projet au sein de votre espace de nom personnel, ce qui vous permet d'y apporter publiquement des modifications, c'est donc tout simplement un moyen de contribuer de manière plus ouverte.

Ainsi, les gestionnaires de projets n'ont pas à se soucier de devoir ajouter des utilisateurs comme collaborateurs pour leur accorder un accès en poussée. Les personnes peuvent dupliquer un projet eux-mêmes, pousser sur leur copie personnelle et fournir leur contribution au dépôt originel en créant une requête de tirage (*Pull Request*), concept qui sera abordé par la suite. Ceci ouvre un fil de discussion avec possibilité de revue de code, pour que le propriétaire et le contributeur puissent discuter et modifier le code proposé jusqu'à ce que le propriétaire soit satisfait du résultat et le fusionne dans son dépôt.

Pour dupliquer un projet, visitez la page du projet et cliquez sur le bouton « *Fork* » en haut à droite de la page.

FIGURE 6-8

Le bouton « Fork ».

Quelques secondes plus tard, vous serez redirigé vers la page de votre nouveau projet, contenant votre copie modifiable du code.

Processus GitHub

GitHub est construit autour d'une certaine organisation de la collaboration, centrée autour des requêtes de tirage (*Pull Request*).

Ce processus de travail fonctionne aussi bien avec une petite équipe soudée collaborant sur un dépôt unique partagé qu'avec une société éclatée à travers le monde ou un réseau d'inconnus contribuant sur un projet au moyen de dizaines de projets dupliqués. Il est centré sur le processus de travail par branches thématiques (voir **"Les branches thématiques"** traité dans le **Chapter 3**).

Le principe général est le suivant :

1. création d'une branche thématique à partir de la branche `master`,
2. validation de quelques améliorations (*commit*),

3. poussée de la branche thématique sur votre projet GitHub (*push*),
4. ouverture d'une requête de tirage sur GitHub (*Pull Request*),
5. discussion et éventuellement possibilité de nouvelles validations (*commit*).
6. Le propriétaire du projet fusionne (*merge*) ou ferme (*close*) la requête de tirage.

C'est essentiellement le processus de gestion par gestionnaire d'intégration traité dans **"Mode du gestionnaire d'intégration"**, mais au lieu d'utiliser des courriels pour communiquer et faire une revue des modifications, les équipes utilisent les outils Web de GitHub.

Détaillons un exemple illustrant une proposition de modification à un projet open-source hébergé sur GitHub.

CRÉATION D'UNE REQUÊTE DE TIRAGE

Tony recherche un programme à faire tourner sur son micro-contrôleur Arduino et a trouvé un programme génial sur GitHub à ***https://github.com/schacon/blink***.

FIGURE 6-9

Le projet auquel nous souhaitons contribuer.

Le seul problème est que le clignotement est trop rapide, nous pensons qu'il serait mieux d'attendre 3 secondes au lieu d'une entre chaque changement d'état. Améliorons donc le programme et soumettons cette amélioration au projet initial.

Premièrement, nous cliquons sur le bouton « Fork » comme mentionné ci-dessus pour obtenir une copie du projet. Notre nom d'utilisateur ici est « tonychacon » donc notre copie de ce projet est à `https://github.com/tonychacon/blink` et c'est ici que nous pouvons la modifier. Nous pouvons aussi la cloner localement, créer une branche thématique, modifier le code et pousser finalement cette modification sur GitHub.

```
$ git clone https://github.com/tonychacon/blink ❶
Clonage dans 'blink'...

$ cd blink
$ git checkout -b slow-blink ❷
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ❸

$ git diff --word-diff ❹
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'three seconds is better' ❺
[master 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ❻
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

❶ Clone notre copie du projet localement

❷ Crée un branche thématique avec un nom descriptif

❸ Modifie le code

❹ Vérifie si la modification est bonne

❺ Valide les modifications dans la branche thématique

❻ Pousse notre branche thématique sur notre dépôt dupliqué GitHub

Maintenant, si nous allons sur notre projet dupliqué sur GitHub, nous pouvons voir que GitHub a remarqué que nous avons poussé une nouvelle branche thématique et affiche un gros bouton vert pour vérifier nos modifications et ouvrir une requête de tirage sur le projet original.

Vous pouvez aussi vous rendre à la page « Branches » à `https://github.com/<utilisateur>/<projet>/branches` pour trouver votre branche et ouvrir une requête de tirage à partir de là.

FIGURE 6-10

Le bouton Pull Request (requête de tirage)

Si nous cliquons sur le bouton vert, une fenêtre nous permet de créer un titre et une description de la modification que nous souhaitons faire intégrer pour que le propriétaire du projet trouve une bonne raison de la prendre en

considération. C'est généralement une bonne idée de passer un peu de temps à écrire une description aussi argumentée que possible pour que le propriétaire sache pourquoi la modification est proposée et en quoi elle apporterait une amélioration au projet.

Nous voyons aussi une liste de soumissions (*commits*) dans notre branche thématique qui sont « en avance » (*ahead*) par rapport à la branche `master` (ici, un seul uniquement) et une visualisation unifiée de toutes les modifications (*unified diff*) qui seraient intégrées en cas de fusion.

FIGURE 6-11

Page de création d'une requête de tirage

Quand vous cliquez sur le bouton « Create pull request » sur cet écran, le propriétaire du projet que vous avez dupliqué reçoit une notification lui indiquant que quelqu'un suggère une modification et qui renvoie à la page contenant toutes les informations correspondantes.

> Bien que les requêtes de tirage soient souvent utilisées de cette façon pour des projets publics quand un contributeur propose une modification complète, elles sont aussi souvent utilisées dans les projets internes *au début* d'un cycle de développement. Comme on peut continuer à pousser sur la branche thématique même **après** l'ouverture de la requête de tirage, on ouvre une requête de tirage très tôt et cela permet de travailler en équipe dans un contexte, plutôt que de l'ouvrir à la toute fin du processus.

ITÉRATIONS SUR UNE REQUÊTE DE TIRAGE

À présent, le propriétaire du projet peut regarder les modifications suggérées et les fusionner ou les rejeter ou encore les commenter. Supposons qu'il apprécie l'idée mais préférerait un temps d'extinction de la lumière légèrement plus long que le temps d'allumage.

Alors que cette conversation a lieu par échange de courriel dans les flux de travail présentés dans **Chapter 5**, ici elle a lieu en ligne sur GitHub. Le propriétaire du projet peut faire une revue des différences en vue unifiées et laisser un commentaire en cliquant sur une des lignes.

FIGURE 6-12

Commentaire sur une ligne spécifique de code de la requête de tirage

Une fois que le mainteneur a commenté, la personne qui a ouvert la requête de tirage (et en fait toute personne surveillant le dépôt) recevra une notification. Nous verrons comment personnaliser ce comportement plus tard, mais si la notification par courriel est activée, Tony recevra un courriel comme celui-ci :

FIGURE 6-13

Commentaires notifiés par courriel

Re: [blink] Three seconds is better (#2)

Scott Chacon <notifications@github.com> 10:55 AM (18 minutes ago)
to schacon/blink, me

In blink.ino:

```
>    digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
> -  delay(1000);               // wait for a second
> +  delay(3000);               // wait for a second
```

I believe it would be better if the light was off for 4 seconds and on for just 3.

—
Reply to this email directly or view it on GitHub.

N'importe qui peut aussi laisser un commentaire global sur la requête de tirage. Sur **Figure 6-14**, nous pouvons voir un exemple où le propriétaire du projet commente une ligne de code puis laisse un commentaire général dans la section de discussion. Vous pouvez voir que les commentaires de code sont aussi publiés dans la conversation.

FIGURE 6-14

Page de discussion d'une requête de tirage

Three seconds is better #2

Edit New Issue

Open tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1 +2 −2

tonychacon commented 6 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.

http://studies.example.com/optimal-led-delays.html

three seconds is better db44c53

schacon commented on the diff just now

blink.ino View full changes

((6 lines not shown))

```
22  22    digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23        - delay(1000);             // wait for a second
    23    + delay(3000);             // wait for a second
```

schacon added a note just now Owner

I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

schacon commented just now Owner

If you make that change, I'll be happy to merge this.

Labels
None yet

Milestone
No milestone

Assignee
No one—assign yourself

Notifications
Unsubscribe
You're receiving notifications because you commented.

2 participants

Lock pull request

Maintenant, le contributeur sait ce qu'il doit faire pour que ses modifications soient intégrées. Heureusement, ici c'est une chose facile à faire. Alors que par

courriel, il faudrait retravailler les séries de commit et les soumettre à nouveau à la liste de diffusion, avec GitHub il suffit de soumettre les correctifs sur la branche thématique et de la repousser.

Le propriétaire du projet sera notifié à nouveau des modifications du contributeur et pourra voir que les problèmes ont été réglés quand il visitera la page de la requête de tirage. En fait, comme la ligne de code initialement commentée a été modifiée entre temps, GitHub le remarque et fait disparaître la différence obsolète.

FIGURE 6-15

Requête de tirage finale

Un point intéressant à noter est que si vous cliquez sur l'onglet « Files Changed » (fichiers modifiés), vous obtenez la différence sous forme unifiée — c'est-à-dire la différence totalement agrégée qui serait introduite dans votre branche principale si cette branche thématique était fusionnée. En équivalent

`git diff`, cela montre automatiquement la même chose que la commande `git diff master...<branche>` pour la branche sur laquelle vous avez ouvert la requête de tirage. Référez-vous à **"Déterminer les modifications introduites"** pour plus d'informations sur ce type de différence.

L'autre point à noter est que GitHub vérifie si la requête de tirage peut être fusionnée proprement et fournit un bouton pour réaliser la fusion sur le serveur. Ce bouton n'apparaît que si vous avez accès en écriture au dépôt et si une fusion peut s'effectuer simplement. Si vous cliquez dessus, GitHub réalise une fusion sans avance rapide (non-fast-forward), ce qui signifie que même si la fusion *pouvait* se faire en avance rapide (fast-forward), il va tout de même créer une soumission de fusion (merge *commit*).

Si vous préférez, vous pouvez simplement tirer la branche et la fusionner localement. Si vous fusionnez cette branche dans `master` et poussez le tout sur GitHub, la requête de tirage sera fermée automatiquement.

C'est le processus de travail de base que la plupart des projets GitHub utilisent. Des branches thématiques sont créées, des requêtes de tirage sont ouvertes dessus, une discussion s'engage, du travail additionnel peut être ajouté sur la branche et à la fin, la requête est soit fermée, soit fusionnée.

PAS SEULEMENT AVEC DES DÉPÔTS DUPLIQUÉS

Il est important de noter que vous pouvez aussi ouvrir une requête de tirage entre deux branches du même dépôt. Si vous travaillez sur une fonctionnalité avec quelqu'un et que vous avez tous deux accès en écriture au projet, vous pouvez pousser une branche thématique sur le dépôt et ouvrir une requête de tirage dessus vers la branche `master` de ce même projet pour démarrer une revue de code et une discussion. Aucune duplication n'est nécessaire.

Requêtes de tirage avancées

Après avoir présenté les bases de la contribution à un projet sur GitHub, voyons quelques trucs et astuces concernant les requêtes de tirage afin d'améliorer votre efficacité .

REQUÊTES DE TIRAGE COMME PATCHS

Il est important de comprendre que pour de nombreux projets, les requêtes de tirage ne sont pas vues comme des files d'attente de patchs parfaits qui doivent s'appliquer correctement dans l'ordre, comme le conçoivent la plupart des projets basés sur des listes de diffusion qui fonctionnent par série de patchs envoyés par courriel. La plupart des projets GitHub voient les branches de requête

de tirage comme des conversations itératives autour d'une modification proposée, aboutissant à une différence unifiée qui est appliquée par fusion.

C'est une distinction importante, car généralement la modification est soumise à revue avant que le code ne soit considéré comme parfait, ce qui est bien plus rare avec les contributions par série de patchs envoyées sur une liste de diffusion. Cela permet une conversation précoce avec les mainteneurs de sorte que l'on atteint une solution correcte par un travail plus collectif. Quand du code est proposé par requête de tirage et que les mainteneurs ou la communauté suggèrent une modification, la série de patchs n'est généralement pas régénérée mais la différence est poussée comme nouvelle soumission (*commit*) à la branche, permettant ainsi d'avancer dans la discussion, tout en conservant intact le contexte du travail passé.

Par exemple, si vous regardez à nouveau la figure **Figure 6-15**, vous noterez que le contributeur n'a pas rebasé sa soumission et envoyé une nouvelle requête de tirage. Au lieu de cela, il a ajouté de nouvelles soumissions (*commit*) et les a poussé dans la branche existante. De cette manière, si on examine cette requête de tirage dans le futur, on peut aisément retrouver la totalité du contexte qui a amené aux décisions prises. L'utilisation du bouton « *Merge* » sur le site crée à dessein un « commit de fusion » (*merge*) qui référence la requête de tirage pour qu'il reste facile de revenir sur celle-ci et d'y rechercher la discussion originale si nécessaire.

SE MAINTENIR À JOUR AVEC LE DÉVELOPPEMENT AMONT

Si votre requête de tirage devient obsolète ou ne peut plus être fusionnée proprement, vous voudrez la corriger pour que le mainteneur puisse la fusionner facilement. GitHub testera cela pour vous et vous indique à la fin de la requête de tirage si la fusion automatique est possible ou non.

FIGURE 6-16

La requête de tirage ne peut pas être fusionnée proprement

Si vous voyez quelque chose comme sur la figure **Figure 6-16**, vous voudrez corriger votre branche pour qu'elle ait un statut vert et que le mainteneur n'ait pas à fournir de travail supplémentaire.

Vous avez deux options. Vous pouvez soit rebaser votre branche sur le sommet de la branche cible (normalement, la branche `master` du dépôt que vous avez dupliqué), soit fusionner la branche cible dans votre branche.

La plupart des développeurs sur GitHub choisirons cette dernière option, pour la même raison que celle citée à la section précédente. Ce qui importe est l'historique et la fusion finale, donc le rebasage n'apporte pas beaucoup plus qu'un historique légèrement plus propre avec en prime une plus grande difficulté d'application et l'introduction possible d'erreurs.

Si vous voulez fusionnez la branche cible pour rendre votre requête de tirage fusionnable, vous ajouterez le dépôt original comme nouveau dépôt distant, récupérerez la branche cible que vous fusionnerez dans votre branche thématique, corrigerez les erreurs et finalement pousserez la branche thématique sur la même branche thématique pour laquelle vous avez ouvert la requête de tirage.

Par exemple, considérons que dans l'exemple « tonychacon » que nous avons utilisé, l'auteur original a fait des modifications qui créent un conflit dans la requête de tirage. Examinons ces étapes.

```
$ git remote add upstream https://github.com/schacon/blink ❶

$ git fetch upstream ❷
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master ❸
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ❹
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ❺
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

❶ Ajoute le dépôt original comme dépôt distant sous le nom « upstream ».

❷ Récupère les derniers travaux depuis ce dépôt distant.

❸ Fusionne la branche principale dans la branche thématique.

❹ Corrige le conflit créé.

❺ Pousse sur la même branche thématique.

Quand vous faites cela, la requête de tirage est automatiquement mise à jour et un nouveau contrôle est effectué pour vérifier la possibilité de fusion.

FIGURE 6-17

La requête de tirage se fusionne proprement maintenant

Une des grandes forces de Git est que vous pouvez faire ceci régulièrement. Si vous avez un projet à très long terme, vous pouvez facilement fusionner depuis la branche cible de nombreuses fois et n'avoir à gérer que les conflits apparus depuis la dernière fusion, rendant ainsi le processus réalisable.

Si vous souhaitez absolument rebaser la branche pour la nettoyer, vous pouvez toujours le faire, mais il vaut mieux ne pas pousser en forçant sur la branche sur laquelle la requête de tirage est déjà ouverte. Si d'autres personnes l'ont déjà tirée et ont travaillé dessus, vous vous exposez aux problèmes décrits dans **"Les dangers du rebasage"**. À la place, poussez cette branche rebasée vers une nouvelle branche sur GitHub et ouvrez une nouvelle requête de tirage qui référence l'ancienne requête, puis fermez l'originale.

RÉFÉRENCES

Votre prochaine question pourrait être : « Comment faire pour référencer l'ancienne requête de tirage ? ». En fait, il y a de très très nombreuses manières de

faire référence à d'autres choses dans GitHub depuis à peu près toutes les zones textuelles.

Commençons par la manière de faire référence à une autre requête de tirage ou à une anomalie (Issue). Toutes les requêtes de tirage et toutes les anomalies sont identifiées par des numéros qui sont uniques au sein d'un projet. Par exemple, vous ne pouvez avoir une requête de tirage numéro 3 et une anomalie numéro 3. Si vous voulez faire référence à n'importe quelle requête de tirage ou anomalie depuis l'une ou l'autre du même projet, il vous suffit d'insérer `#<numéro>` dans n'importe quel commentaire ou n'importe quelle description. Vous pouvez aussi référencer une requête ou une anomalie d'un autre dépôt dupliqué du dépôt actuel en utilisant la syntaxe `<utilisateur>#<numéro>`, ou même un autre dépôt indépendant avec la syntaxe `<utilisateur>/<dépôt>#<numéro>`.

Voyons cela sur un exemple. Disons que nous avons rebasé la branche de l'exemple précédent, créé une nouvelle requête de tirage et nous souhaitons maintenant faire référence à l'ancienne requête de tirage depuis la nouvelle. Nous souhaitons aussi faire référence à une anomalie dans un dépôt dupliqué de celui-ci et à une anomalie soumise dans un projet complètement différent. Nous pouvons saisir une description comme sur la figure **Figure 6-18**.

FIGURE 6-18

Références croisées dans une requête de tirage.

Quand nous soumettons cette requête de tirage, nous voyons tout ceci mis en forme comme sur la figure **Figure 6-19**.

FIGURE 6-19

Références croisées mises en forme dans une requête de tirage.

Notez bien que l'URL GitHub complète que nous avons indiquée a été raccourcie pour ne contenir que l'information nécessaire.

À présent, si Tony retourne sur la requête de tirage originale et la ferme, nous pouvons voir que du fait de son référencement dans la nouvelle, GitHub a créé automatiquement un événement de suivi dans le journal de la nouvelle requête de tirage. Cela signifie qu'une personne qui visitera cette requête de tirage et verra qu'elle est fermée, pourra facilement se rendre sur celle qui l'a remplacée. Le lien ressemblera à quelque chose comme sur la figure **Figure 6-20**.

FIGURE 6-20

Références croisée dans une requête de tirage fermée.

En plus des numéros d'anomalies, vous pouvez aussi faire référence à une soumission (commit) spécifique par son SHA-1. Vous devez spécifier la totalité des 40 caractères du SHA-1, mais si GitHub rencontre cette chaîne, il créera un

lien direct vers la soumission. Vous pouvez aussi faire référence à des soumissions dans des dépôts dupliqués ou d'autres dépôts de la même manière que nous l'avons fait pour les anomalies.

Markdown

Faire des liens vers les autres anomalies n'est que le début des choses intéressantes que vous pouvez faire dans presque toutes les boîtes de saisie dans GitHub. Dans les descriptions d'anomalies et de requêtes de tirage, les commentaires, les commentaires de code et plus, vous pouvez utiliser ce qu'on appelle le « Markdown, saveur GitHub » (*GitHub Flavored Markdown*). Markdown, c'est comme écrire du texte simple mais celui-ci est rendu plus richement.

Référez-vous à l'exemple sur la figure **Figure 6-21** pour savoir comment les commentaires ou le texte peuvent être écrits puis rendus en utilisant Markdown.

FIGURE 6-21

Un exemple de Markdown écrit et rendu.

MARKDOWN, SAVEUR GITHUB

La saveur GitHub de Markdown permet de réaliser encore plus de choses au-delà de la syntaxe Markdown basique. Celles-ci peuvent être vraiment utiles pour la création de requêtes de tirage, de commentaires d'anomalies ou de descriptions.

Listes de tâches

La première spécificité vraiment utile du Markdown de GitHub, particulièrement dans le cadre de requêtes de tirage, est la création de listes de tâches. Une liste de tâches est une liste de cases à cocher pour chaque action à accomplir. Dans une anomalie ou une requête de tirage, cela indique les choses qui doit être faites avant de pouvoir considérer l'élément comme fermé.

Vous pouvez créer une liste de tâches comme ceci :

```
- [X] Écrire le code
- [ ] Écrire tous les tests
- [ ] Documenter le code
```

Si nous incluons ceci dans la description de notre requête de tirage ou de notre anomalie, nous le verrons rendu comme sur la figure **Figure 6-22**

FIGURE 6-22

Rendu d'une liste de tâches dans un commentaire Markdown.

C'est très utilisé dans les requêtes de tirage pour indiquer tout ce que vous souhaitez voir accompli sur la branche avant que la requête de tirage ne soit prête à être fusionnée. Le truc vraiment cool est que vous pouvez simplement cliquer sur les cases à cocher pour mettre à jour le commentaire — il est inutile de modifier directement le Markdown pour cocher les cases.

De plus, GitHub surveille la présence de listes de tâches dans vos anomalies et vos requêtes de tirage et les affiche comme métadonnées sur les pages qui en donnent la liste. Par exemple, si vous avez une requête de tirage contenant des tâches et que vous regardez la page de résumé de toutes les requêtes de tirage, vous pouvez y voir l'état d'avancement. Cela aide les gens à découper les requêtes de tirage en sous-tâches et aide les autres personnes à suivre le progrès sur la branche. Vous pouvez voir un exemple de cette fonctionnalité sur la figure **Figure 6-23**.

FIGURE 6-23

Résumé de listes de tâches dans la liste des requêtes de tirage.

C'est incroyablement utile quand vous ouvrez tôt une requête de tirage et les utilisez pour suivre votre progrès au cours du développement de la fonctionnalité.

Extraits de code

Vous pouvez aussi ajouter des extraits de code dans les commentaires. C'est particulièrement utile si vous souhaitez montrer quelque chose que vous *pourriez* essayer de faire avant de les développer réellement dans votre branche sous la forme d'une soumission. C'est aussi souvent utilisé pour ajouter un exemple de code de ce qui ne fonctionne pas ou de ce que cette requête de tirage pourrait mettre en œuvre.

Pour ajouter un extrait de code, vous devez le délimiter par des guillemets simples inversés.

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

Si de plus vous ajoutez un nom de langage comme nous l'avons fait avec *java*, GitHub essaye de colorer syntaxiquement l'extrait. Dans le cas ci-dessus, cela donnerait le rendu sur la figure **Figure 6-24**.

FIGURE 6-24

Exemple de rendu d'un code délimité

Citation

Si vous répondez à une petite partie d'un long commentaire, vous pouvez citer la partie concernée de l'autre commentaire de manière sélective en faisant précéder chaque ligne par le caractère >. En réalité, c'est même tellement courant et utile qu'il existe un raccourci clavier pour cela. Si vous sélectionnez un texte dans un commentaire auquel vous voulez directement répondre et que vous appuyez sur la touche r, ce texte sera automatiquement cité pour vous dans votre boîte de commentaire.

Les citations ressemblent à quelque chose comme ça :

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Une fois rendu, le commentaire ressemble à quelque chose comme sur la figure **Figure 6-25**.

FIGURE 6-25

Exemple de rendu de citation.

Émoticône (Emoji)

Enfin, vous pouvez également utiliser des émoticônes dans vos commentaires. C'est en réalité utilisé assez largement dans les commentaires que vous pouvez voir pour de nombreuses anomalies et requêtes de tirage GitHub. Il existe même un assistant pour émoticônes dans GitHub. Lorsque vous saisissez un commentaire et que vous commencez à saisir le caractère `:`, un outil pour l'auto-complétion vous aide à trouver ce que vous recherchez.

FIGURE 6-26

Auto-complétion d'émoticônes en action.

Les émoticônes apparaissent sous la forme `:<nom>:` n'importe où dans le commentaire. Par exemple, vous pourriez écrire quelque chose comme cela :

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

Une fois rendu, cela ressemblerait à quelque chose comme sur la figure **Figure 6-27**.

FIGURE 6-27

Commentaire très chargé en émoticônes.

Bien que cela ne soit pas indispensable, cela ajoute une touche d'humour et d'émotion à un moyen de communication avec lequel il est difficile de transmettre des émotions.

Il y a en fait un assez grand nombre de services Web qui emploient maintenant des émoticônes. Un formidable aide mémoire de référence pour trouver des émoticônes qui expriment ce que vous souhaitez dire peut être trouvé ici :

http://www.emoji-cheat-sheet.com

Images

Ce n'est pas à proprement parler du Markdown, saveur GitHub, mais c'est incroyablement utile. En plus de l'ajout de liens images aux commentaires (dont il peut être difficile de trouver et d'intégrer les URL), GitHub vous permet de faire un glisser-déposer de vos images sur les zones de texte pour les intégrer.

FIGURE 6-28

Glisser-déposer d'images pour les télécharger et les intégrer.

Si vous regardez à nouveau l'image **Figure 6-18**, vous y verrez une petite indication "Parsed as Markdown" (Traitement Markdown) en haut de la zone de texte. En cliquant dessus, vous serez redirigé vers une page (en anglais) affichant un aide-mémoire de référence vous résumant tout ce que vous pouvez faire avec Markdown sur GitHub.

Maintenance d'un projet

Maintenant que vous êtes à l'aise sur les aspects contribution à un projet, regardons maintenant l'autre côté : la création, la maintenance et l'administration de vos propres projets.

Création d'un nouveau dépôt

Créons un nouveau dépôt pour permettre le partage du code de notre projet avec d'autres. Commencez par cliquer sur le bouton « New repository » (nouveau dépôt) sur le côté droit de votre tableau de bord ou sur le bouton + dans la barre d'outils du haut à côté de votre nom d'utilisateur comme sur la figure **Figure 6-30**.

FIGURE 6-29

La zone « Your repositories » (vos dépôts)

FIGURE 6-30

La liste déroulante « New repository » (nouveau dépôt)

Vous êtes redirigé vers le formulaire pour la création de nouveau dépôt :

FIGURE 6-31

Le formulaire « new repository » (nouveau dépôt).

Tout ce que vous avez à faire, c'est de fournir un nom de projet, les autres champs sont facultatifs. Pour l'instant, cliquez juste sur le bouton « Create Repository » (créer un dépôt) et paf, vous obtenez un nouveau dépôt sur GitHub nommé `<utilisateur>/<nom_du_projet>`.

Puisque vous n'avez pas encore de code, GitHub vous affiche des instructions sur la façon de créer un tout nouveau dépôt Git ou de se connecter à un projet Git existant. Nous ne détaillerons pas cela ici ; si vous avez besoin d'un rappel, vérifiez **Chapter 2**.

Maintenant que votre projet est hébergé sur GitHub, vous pouvez donner l'URL à toutes les personnes avec lesquelles vous voulez partager votre projet. Chaque projet est accessible via HTTP par `https://github.com/<utilisateur>/<nom_du_projet>` et via SSH par `git@github.com:<utilisateur>/<nom_du_projet>`. Git peut récupérer et pousser en utilisant les deux URL mais l'accès est contrôlé sur la base des paramètres d'authentification de l'utilisateur qui s'y connecte.

Il est souvent mieux de partager l'URL basé sur HTTP pour un projet public puisque l'utilisateur n'a pas besoin d'avoir un compte GitHub pour y accéder et pour le cloner. Les utilisateurs devront posséder un compte et avoir déposé une clé SSH pour accéder à votre projet si vous leur donnez l'URL SSH. L'URL HTTP est également exactement le même que celui que vous colleriez dans votre navigateur pour y afficher le projet.

Ajout de collaborateurs

Si vous travaillez avec d'autres personnes à qui vous voulez donner l'accès en poussée, vous devez les ajouter en tant que « collaborateurs ». Si Ben, Jeff et Louise possèdent tous un compte GitHub et que vous voulez qu'ils puissent pousser sur votre dépôt, vous pouvez les ajouter à votre projet. En faisant cela, vous leur donnez un accès en poussée ce qui signifie qu'ils possèdent un accès en lecture et en écriture au projet et au dépôt Git.

Cliquez sur le lien « Settings » (paramètres) en bas de la barre latérale de droite.

FIGURE 6-32

Le lien des paramètres (Settings) du dépôt.

Ensuite sélectionnez « Collaborators » dans le menu de gauche, saisissez un nom d'utilisateur dans la boîte et cliquez sur « Add collaborator » (ajouter un collaborateur). Vous pouvez répéter cette action autant de fois que vous le voulez pour permettre l'accès à toutes les personnes que vous souhaitez. Si vous devez révoquer leur accès, il suffit de cliquer sur le « X » à droite de leur nom.

FIGURE 6-33

Les collaborateurs du dépôt.

Gestion des requêtes de tirage

Maintenant que vous possédez un projet contenant un peu de code et peut-être même quelques collaborateurs qui possèdent un accès en poussée, voyons ce que vous devez faire lorsque vous recevez vous-même une requête de tirage.

Les requêtes de tirage peuvent provenir soit d'une branche d'un clone de votre dépôt ou d'une autre branche du même dépôt. La seule différence est que celles d'un clone proviennent souvent de personnes vers lesquelles vous ne pouvez pas pousser sur leurs branches et qui ne peuvent pas pousser vers les vôtres alors qu'avec des requêtes de tirage internes, les deux parties peuvent généralement accéder à la branche.

Pour ces exemples, supposons que vous êtes « tonychacon » et que vous avez créé un nouveau projet de code Arduino qui s'appelle « fade ».

NOTIFICATIONS PAR COURRIEL

Quelqu'un se connecte et fait une modification à votre programme et vous envoie une requête de tirage. Vous devriez recevoir un courriel vous informant de cette nouvelle requête de tirage et ressemblant à celui sur la figure **Figure 6-34**.

FIGURE 6-34

Notification par courriel d'une nouvelle requête de tirage.

[fade] Wait longer to see the dimming effect better (#1)

Scott Chacon <notifications@github.com> 10:05 AM (0 minutes ago)
to tonychacon/fade Unsubscribe

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

https://github.com/tonychacon/fade/pull/1

Commit Summary

- wait longer to see the dimming effect better

File Changes

- **M** fade.ino (2)

Patch Links:

- https://github.com/tonychacon/fade/pull/1.patch
- https://github.com/tonychacon/fade/pull/1.diff

—
Reply to this email directly or view it on GitHub.

Faisons quelques remarques à propos de ce courriel. Celui-ci vous fournit quelques statistiques : une liste de fichiers modifiés par la requête de tirage et le nombre de modifications. Il vous donne un lien vers la requête de tirage sur GitHub et il vous fournit également quelques URL que vous pouvez utiliser en ligne de commande.

Remarquez la ligne `git pull <url> patch-1`, il s'agit d'une manière simple de fusionner une branche distante sans avoir à ajouter un dépôt distant. Nous avons déjà vu rapidement cela dans **"Vérification des branches distantes"**. Si vous voulez, vous pouvez créer une branche thématique et basculer vers celle-ci puis lancer cette commande pour fusionner les modifications de cette requête de tirage.

Les autres URL intéressantes sont les URL `.diff` et `.patch`, qui, comme vous l'avez certainement deviné, vous fournissent des versions au format différence unifiée et patch de la requête de tirage. Vous pourriez techniquement fusionner le travail contenu dans la requête de tirage de la manière suivante :

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

COLLABORATION À UNE REQUÊTE DE TIRAGE

Comme déjà traité dans la section **"Processus GitHub"**, vous pouvez maintenant commencer une conversation avec la personne qui a ouvert la requête de tirage. Vous pouvez commenter certaines lignes de code, commenter des sou-

missions complètes ou commenter la requête de tirage elle-même en utilisant les outils Markdown, saveur GitHub un peu partout.

À chaque fois que quelqu'un d'autre commente la requête de tirage, vous recevrez des notifications par courriel afin d'être au courant de chaque activité. Celles-ci possèdent un lien vers la requête de tirage dans laquelle l'activité s'est produite et vous pouvez également répondre directement au courriel pour commenter le fil de discussion de la requête de tirage.

FIGURE 6-35

Les réponses aux courriels sont incorporées dans le fil de discussion.

Une fois que le code est dans un état satisfaisant et que vous voulez le fusionner, vous pouvez soit tirer le code et le fusionner localement, soit utiliser la syntaxe décrite précédemment `git pull <url> <branch>`, soit ajouter le clone comme dépôt distant, le récupérer et le fusionner.

Si la fusion est triviale, vous pouvez également cliquer sur le bouton « Merge » (fusionner) sur le site GitHub. Une fusion sans avance rapide (non-fast-forward) sera réalisée ce qui créera une soumission de fusion (*merge commit*) même si une fusion en avance rapide (fast-forward) était possible. Cela signifie que dans tous les cas, à chaque fois que vous cliquez sur le bouton « Merge », un commit de fusion est créé. Comme vous pouvez le voir sur **Figure 6-36**, GitHub vous donne toutes ces informations si vous cliquez sur le lien descriptif.

FIGURE 6-36

Bouton « Merge » et instructions pour la fusion manuelle d'une requête de tirage.

Si vous décidez que vous ne voulez pas fusionner, vous pouvez tout simplement fermer la requête de tirage et la personne qui l'a créée en sera informée.

RÉFÉRENCES AUX REQUÊTES DE TIRAGE

Si vous gérez **beaucoup** de requêtes de tirage et que vous ne voulez pas ajouter une série de dépôts distants ou faire des tirages isolés à chaque fois, GitHub vous permet une astuce. C'est toutefois une astuce avancée et nous irons un peu plus dans les détails à la section **"La *refspec*"** mais cela peut être assez utile dès maintenant.

GitHub traite en réalité les branches de requête de tirage d'un dépôt comme une sorte de pseudo-branches sur le serveur. Par défaut, vous ne les obtenez pas lorsque vous clonez mais elles sont présentes de façon cachée et vous pouvez y accéder assez facilement.

Pour le montrer, nous allons utiliser une commande bas niveau (souvent appelée commande de « plomberie » dont nous parlerons un peu plus dans la section **"Plomberie et porcelaine"**) qui s'appelle `ls-remote`. Cette commande n'est en général pas utilisée dans les opérations quotidiennes mais elle est utile pour afficher les références présentes sur le serveur.

Si nous lançons cette commande sur le dépôt « blink » que nous utilisions tout à l'heure, nous obtenons la liste de toutes les branches et étiquettes ainsi que d'autres références dans le dépôt.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d        HEAD
10d539600d86723087810ec636870a504f4fee4d        refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e        refs/pull/1/head
```

```
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3        refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1        refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d        refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a        refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c        refs/pull/4/merge
```

Bien sûr, si vous êtes dans votre dépôt et que vous lancez la commande `git ls-remote origin` (ou avec un autre dépôt distant), quelque chose de similaire s'affiche.

Si le dépôt se trouve sur GitHub et que des requêtes de tirage ont été ouvertes, vous obtiendrez leurs références préfixées par `refs/pull/`. Ce sont simplement des branches mais comme elles ne sont pas sous `refs/heads/`, vous ne les obtenez généralement pas lorsque vous clonez ou récupérez à partir d'un serveur — le processus de récupération les ignore normalement.

Il y a deux références par requête de tirage - l'une se termine par `/head` et pointe vers la même soumission que la dernière soumission dans la branche de requête de tirage. Donc si quelqu'un ouvre une requête de tirage sur notre dépôt, que leur branche s'appelle `bug-fix` et qu'elle pointe sur la soumission `a5a775`, alors dans **notre** dépôt nous n'aurons pas de branche `bug-fix` (puisqu'elle se trouve dans leur clone) mais nous *aurons* une référence `pull/<pr#>/head` qui pointe vers `a5a775`. Cela signifie que vous pouvez assez facilement tirer toute branche de requête de tirage d'un coup sans avoir à ajouter tout un tas de dépôts distants.

Vous pouvez désormais récupérer la référence directement.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

Cela dit à Git, « Connecte-toi au dépôt distant `origin` et télécharge la référence appelée `refs/pull/958/head` ». Git obéit joyeusement et télécharge tout ce dont vous avez besoin pour construire cette référence et positionne un pointeur vers la soumission souhaitée sous `.git/FETCH_HEAD`. Vous pouvez continuer en faisant `git merge FETCH_HEAD` dans une branche dans laquelle vous voulez la tester mais ce message de fusion (*merge commit*) semble un peu bizarre. De plus, si vous passez en revue **beaucoup** de requêtes de tirage, cela devient fastidieux.

Il existe également une façon de récupérer *toutes* les requêtes de tirage et de les maintenir à jour à chaque fois que vous vous connectez au dépôt distant. Ouvrez le fichier `.git/config` dans votre éditeur favori et cherchez le dépôt `origin`. Cela devrait ressembler à cela :

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

La ligne qui commence par `fetch =` est une spécification de références (refspec). C'est une façon de faire correspondre des noms sur un dépôt distant à des noms dans votre dossier `.git` local. Celle-ci en particulier dit à Git, « les choses sur le dépôt distant qui se trouvent sous `refs/heads` doivent aller dans mon dépôt local sous `refs/remotes/origin` ». Vous pouvez modifier cette section pour ajouter une autre spécification de références :

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Cette dernière ligne dit à Git, « Toutes les références du type `refs/pull/123/head` doivent être enregistrées localement comme `refs/remotes/origin/pr/123` ». Maintenant, si vous enregistrez ce fichier et faites une récupération (`git fetch`) :

```
$ git fetch
# …
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
# …
```

Maintenant toutes les requêtes de tirage distantes sont représentées localement par des références qui agissent un peu comme des branches de suivi : elles sont en lecture seule et elles se mettent à jour lorsque vous faites un tirage. Il est ainsi super facile d'essayer le code d'une requête de tirage localement :

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

Les Sherlock Holmes en herbe parmi vous auront remarqué le terme `head` à la fin de la partie distante de la spécification de références. Il y a également une référence `refs/pull/#/merge` du côté de GitHub qui représente la soumission qui serait obtenue si vous cliquiez sur le bouton « Fusionner » sur le site. Cela peut vous permettre de tester la fusion avant même de cliquer sur le bouton.

REQUÊTES DE TIRAGE SUR DES REQUÊTES DE TIRAGE

Non seulement vous pouvez ouvrir des requêtes de tirage qui ciblent la branche principale ou `master`, mais vous pouvez en fait ouvrir une requête de tirage ciblant n'importe quelle branche du réseau. En réalité, vous pouvez même cibler une autre requête de tirage.

Si vous remarquez une requête de tirage qui va dans la bonne direction et que vous avez une idée de modifications qui dépendent de celle-ci, ou vous n'êtes pas sûr que c'est une bonne idée, ou vous n'avez tout simplement pas accès en poussée vers la branche cible, vous pouvez ouvrir une requête de tirage directement sur elle.

Lorsque vous ouvrez une requête de tirage, une boîte en haut de la page vous indique vers quelle branche vous voulez pousser et à partir de quelle branche vous allez tirer. Si vous cliquez sur le bouton « Edit » (modifier) à droite de cette boîte, vous pouvez modifier non seulement les branches mais aussi le clone.

FIGURE 6-37

Modification manuelle du clone cible et de la branche de la requête de tirage.

Vous pouvez à cet instant très facilement indiquer de fusionner votre nouvelle branche sur une autre requête de tirage ou un autre clone du projet.

Mentions et notifications

GitHub dispose également d'un système de notifications intégré assez sympa qui peut devenir utile lorsque vous avez des questions et besoin du retour de certaines personnes ou d'équipes.

Dans tous les commentaires, si vous saisissez le caractère `@`, cela commence à proposer des noms et des noms d'utilisateur de personnes qui collaborent ou contribuent au projet.

FIGURE 6-38

Saisissez @ pour faire référence à quelqu'un.

Vous pouvez aussi faire référence à un utilisateur qui n'apparaît pas dans cette liste, mais souvent l'auto-complétion accélère les choses.

Une fois que vous avez posté un commentaire contenant une référence à un utilisateur, ce dernier reçoit une notification. Cela signifie que c'est une manière très pratique de faire entrer des gens dans une conversation plutôt que de leur demander. Très souvent dans des requêtes de tirage sur GitHub, les gens vont attirer d'autres personnes dans leurs équipes ou dans leur société pour vérifier une anomalie ou une requête de tirage.

Si quelqu'un est cité dans une requête de tirage ou une anomalie, il est « inscrit » à celle-ci et continue à recevoir des notifications dès qu'une activité se produit. Vous êtes également inscrit à quelque chose si vous l'ouvrez, si vous observez (*watch*) un dépôt ou si vous faites un commentaire sur quelque chose. Si vous ne souhaitez plus recevoir de notifications, cliquez sur le bouton « Unsubscribe » (se désinscrire) de la page pour arrêter de recevoir les mises à jour.

FIGURE 6-39

Désinscription d’une anomalie ou d’une requête de tirage.

LA PAGE DES NOTIFICATIONS

Lorsque nous parlons de « notifications » ici, par rapport à GitHub, nous voulons parler de la manière spécifique par laquelle GitHub essaye de vous joindre lorsque des événements se produisent et il existe différentes façons de la configurer. Si vous allez dans l’onglet « Notification center » (centre de notification) dans la page des paramètres, vous pouvez voir les différentes options disponibles.

FIGURE 6-40

Options du centre de notification.

Vous pouvez recevoir des notifications soit par « courriel », soit par le « Web » et vous pouvez sélectionner une, aucune ou les deux méthodes si vous voulez participer de manière très active ou pour une activité particulière dans les dépôts que vous surveillez.

Notifications Web

Les notifications Web n'existent que sur GitHub et vous ne pouvez les visionner que sur GitHub. Si vous avez sélectionné cette option dans vos préférences et qu'une notification vous est envoyée, un petit point bleu apparaît sur votre icône des notifications en haut de l'écran comme sur la figure **Figure 6-41**.

FIGURE 6-41

Centre de notification.

Si vous cliquez dessus, la liste de tous les éléments pour lesquels vous avez été notifié apparaît, regroupés par projet. Vous pouvez filtrer les notifications d'un projet particulier en cliquant sur son nom dans la barre latérale gauche. Vous pouvez aussi accepter la notification en cochant l'icône à côté de celle-ci ou accepter *toutes* les notifications d'un projet en cochant la case en haut du groupe. Il y a aussi un bouton « muet » à côté de chaque case que vous pouvez cliquer afin de ne plus recevoir de notifications sur cet élément.

Tous ces outils sont très utiles pour gérer un grand nombre de notifications. Beaucoup d'utilisateurs de GitHub très actifs arrêtent tout simplement complètement les notifications par courriel et gèrent toutes leurs notifications à partir de cette fenêtre.

Notifications par courriel

Les notifications par courriel sont l'autre façon de gérer les notifications provenant de GitHub. Si vous les avez activées, vous recevrez des courriels pour chaque notification. Nous avons vu des exemples concernant cela sur les figures **Figure 6-13** et **Figure 6-34**. Ces courriels peuvent être également suivis

correctement ce qui est bien agréable si vous utilisez un client de messagerie qui suit les fils de discussion.

Un assez grand nombre de métadonnées sont incluses dans les entêtes des courriels que GitHub vous envoie ce qui peut vraiment vous aider à configurer des filtres et des règles personnalisés.

Par exemple si nous observons les entêtes complets du courriel envoyé à Tony dans le courriel de la figure **Figure 6-34**, nous voyons que les informations suivantes sont envoyées :

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

Il y a quelques petites choses intéressantes ici. Si vous voulez mettre en valeur ou rediriger les courriels de ce projet ou d'une requête en tirage en particulier, l'information du champ `Message-ID` vous fournit toutes les données au format `<utilisateur>/<projet>/<type>/<id>`. Si c'était une anomalie, le champ `<type>` aurait été « issues » à la place de « pull ».

Les champs `List-Post` et `List-Unsubscribe` signifient que si votre client de messagerie les prend en compte, vous pouvez facilement écrire (*post*) à la liste ou vous désinscrire (*unsubscribe*) du fil de discussion. Cela correspond à cliquer sur la case « muet » sur la version Web de la notification ou sur « Unsubscribe » sur la page personnelle de l'anomalie ou de la requête de tirage.

Il est aussi intéressant de noter que si les notifications par courriel et par Web sont toutes deux activées et que vous lisez la version courriel de la notification, la version Web sera également marquée comme lue si vous avez autorisé l'affichage des images dans votre client de messagerie.

Fichiers spéciaux

Quelques fichiers spéciaux attirent l'attention de GitHub s'ils existent dans votre dépôt.

README

Le premier est le fichier `README` (LISEZ-MOI) qui peut être écrit sous n'importe quel format textuel reconnu par GitHub. Par exemple, cela pourrait être `RE-`

ADME, `README.md`, `README.asciidoc`, etc. Si GitHub trouve un fichier `README` dans vos sources, celui-ci sera rendu sur la page d'accueil du projet.

Pour beaucoup d'équipes, ce fichier contient toutes les informations importantes du projet pour quelqu'un qui serait nouveau dans le dépôt ou le projet. Il contient habituellement des choses comme :

- À quoi sert le projet.
- Comment le configurer et l'installer.
- Un exemple d'utilisation et comment le lancer.
- La licence sous laquelle le projet est proposé.
- Comment y contribuer.

Puisque GitHub va afficher à l'écran ce fichier, vous pouvez y incorporer des images ou des liens pour faciliter la compréhension.

CONTRIBUTING

L'autre fichier spécial que GitHub reconnaît est le fichier `CONTRIBUTING`. Si vous possédez un fichier nommé `CONTRIBUTING`, peu importe son extension, GitHub affichera la figure **Figure 6-42** lorsque quelqu'un commence à ouvrir une requête de tirage.

FIGURE 6-42

Ouverture d'une requête de tirage si un fichier CONTRIBUTING existe.

L'idée ici est d'expliquer les choses particulières que vous voulez ou ne voulez pas voir soumises dans une requête de tirage envoyée vers votre projet. De cette façon, les gens peuvent vraiment lire les recommandations avant d'ouvrir la requête de tirage.

Administration du projet

Il n'y a généralement pas beaucoup de tâches administratives à faire si vous avez un seul projet, mais ces quelques points peuvent vous intéresser.

MODIFICATION DE LA BRANCHE PAR DÉFAUT

Si vous utilisez une autre branche que « master » comme branche par défaut et que vous voulez que les gens ouvrent les requêtes de tirage dessus ou la voient par défaut, vous pouvez modifier cela dans la page des paramètres de votre dépôt dans l'onglet « Options ».

FIGURE 6-43

Modification de la branche par défaut pour un projet.

Modifiez tout simplement la branche par défaut dans la liste déroulante et celle-ci sera la branche par défaut pour toutes les opérations principales à partir de maintenant, y compris la branche qui sera extraite par défaut lorsque quelqu'un clone le dépôt.

TRANSFERT DE PROJET

Si vous voulez transférer un projet à un autre utilisateur ou une organisation dans GitHub, une option « Transfer ownership » (transférer la propriété) en bas du même onglet « Options » de la page des paramètres de votre dépôt vous permet cela.

FIGURE 6-44

Transfert d'un projet vers un autre utilisateur GitHub ou une organisation.

C'est bien pratique si vous abandonnez un projet et que quelqu'un souhaite le récupérer ou si votre projet devient plus gros et que vous voulez le déplacer vers une organisation.

Non seulement, cela déplace le dépôt ainsi que tous ses observateurs et étoiles vers un autre endroit, mais cela met également en place une redirection de votre URL vers le nouvel emplacement. Cela redirige également les clones et les tirages à partir de Git et pas seulement les requêtes Web.

Gestion d'un regroupement

En plus d'avoir des comptes par utilisateur, GitHub propose également ce qui s'appelle des « Organizations » (regroupements). Tout comme les comptes personnels, les comptes de regroupements possèdent un espace nommé où se trouvent tous les projets mais de nombreuses autres choses sont différentes. Ces comptes représentent un groupe de personnes qui partagent la propriété de projets et de nombreux outils de gestion de sous-groupes parmi ces personnes sont proposés. Normalement ces comptes sont utilisés pour des groupes open-source (tels que « perl » ou « rail ») ou des sociétés (comme « google » ou « twitter »).

Les bases d'un regroupement

Un regroupement est très facile à créer, il suffit de cliquer sur l'icône « + » située dans le coin supérieur droit de n'importe quelle page GitHub et de sélectionner « *New Organization* » (nouveau regroupement) dans le menu.

FIGURE 6-45

L'élément de menu « New organization ».

Vous devrez d'abord donner un nom à votre regroupement et fournir une adresse électronique comme principal point de contact du groupe. Ensuite vous pouvez, si vous voulez, inviter d'autres utilisateurs à devenir co-propriétaires du compte.

En suivant ces étapes, vous devenez le propriétaire d'un tout nouveau regroupement. Tout comme les comptes personnels, les regroupements sont gratuits si tout ce que vous envisagez d'enregistrer est open source.

En tant que propriétaire d'un regroupement, lorsque vous dupliquez un dépôt, vous aurez la possibilité de le dupliquer vers l'espace de nom de votre regroupement. Lorsque vous créez un dépôt, vous pouvez le faire soit dans votre compte personnel, soit dans l'un des regroupements dont vous êtes propriétaire. Vous pouvez aussi automatiquement suivre (*watch*) n'importe quel nouveau dépôt créé sous ce regroupement.

Tout comme dans **"Votre Avatar"**, vous pouvez télécharger un avatar pour votre regroupement pour le personnaliser un peu. Et tout comme pour les comptes personnels, vous possédez une page d'accueil pour le regroupement qui énumère tous vos dépôts et qui peut être vue par tout le monde.

Maintenant, passons aux éléments qui sont un peu différents pour un compte de regroupement.

Équipes

Les regroupements sont associés à des individus au travers d'équipes (*teams*) qui sont tout simplement un groupe de comptes utilisateur individuels et de

dépôts au sein du regroupement et qui définissent le type d'accès que possèdent ces personnes vers ces dépôts.

Par exemple, supposons que votre société possède trois dépôts : `frontend`, `backend` et `deployscripts`. Vous aimeriez que vos développeurs HTML/CSS/Javascript aient accès à `frontend` et peut-être `backend` et que les responsables opérationnels aient accès à `backend` et `deployscripts`. Les équipes vous facilitent la vie, sans avoir à gérer les collaborateurs pour chaque dépôt spécifiquement.

La page du regroupement vous affiche un tableau de bord très simple de tous les dépôts, utilisateurs et équipes dans ce regroupement.

FIGURE 6-46

La page du regroupement.

Pour gérer vos équipes, vous pouvez cliquer sur la barre latérale « Teams » sur le côté droit de la page sur la figure **Figure 6-46**. Cela vous redirige vers une page qui vous permet d'ajouter des membres ou des dépôts dans l'équipe et de gérer les paramètres et les niveaux de contrôle pour l'équipe. Chaque équipe peut avoir un accès en lecture seule, en lecture/écriture ou en administration (*administrative*) aux dépôts. Vous pouvez modifier ce niveau en cliquant sur le bouton « *Settings* » de la figure **Figure 6-47**.

FIGURE 6-47

La page équipe.

Lorsque vous invitez quelqu’un dans une équipe, celui-ci reçoit un courriel lui indiquant qu’il a été invité.

De plus, les `@mentions` d’équipes (telle que `@acmecorp/frontend`) fonctionnent de la même façon que pour les utilisateurs individuels sauf que **tous** les membres de l’équipe sont alors inscrits pour suivre le fil de discussion. C’est utile si vous voulez attirer l’attention de quelqu’un dans une équipe mais vous ne savez pas exactement à qui vous adresser.

Un utilisateur peut appartenir à un grand nombre d’équipes donc ne vous limitez pas seulement à voir les équipes comme des groupes de contrôle d’accès. Des équipes par centre d’intérêt comme `ux`, `css` ou `refactoring` sont utiles pour certains types de points et d’autres comme `legal` et `colorblind` pour tout autre chose.

Journal d’audit

Les regroupements donnent aussi accès aux propriétaires à toute information concernant les activités au sein du regroupement. Dans l’onglet « Audit Log » (journal d’audit), vous pouvez voir les événements qui ont eu lieu d’un point de vue organisationnel, qui y a participé et où elles ont eu lieu dans le monde.

FIGURE 6-48

Journal d'audit.

Vous pouvez aussi filtrer par type d'événement, par lieu ou par personne.

Écriture de scripts pour GitHub

Nous avons pour l'instant traité de toutes les principales fonctionnalités et des cycles de travail de GitHub mais tous les grands groupes ou projets ont des personnalisations qu'ils souhaiteront intégrer ou des services externes qu'ils voudront intégrer.

Heureusement pour nous, il est facile de « bidouiller » GitHub de différentes façons. Dans cette section nous traiterons de la façon d'utiliser le système de crochets (*hooks*) de GitHub et son interface de programmation (*API*) afin que GitHub fonctionne de la façon que nous souhaitons.

Crochets (*Hooks*)

La section « Hooks & Services » (crochets et services) de l'administration de dépôt GitHub est la façon la plus facile de faire interagir GitHub avec des systèmes externes.

SERVICES

Intéressons-nous d'abord aux services. Les intégrations de services et de crochets se trouvent tous les deux dans la section Settings (paramètres) de votre dépôt où nous avions précédemment ajouté des collaborateurs et modifié la branche par défaut de votre projet. La figure **Figure 6-49** vous montre ce que vous verrez en cliquant sur l'onglet « Webhooks and Services ».

FIGURE 6-49

Section configuration des crochets et services.

Vous pouvez choisir parmi des dizaines de services, la plupart sont des intégrations vers d'autres systèmes commerciaux et open source. Certains sont des services d'intégration continue, des analyseurs de bogues et d'anomalies, des systèmes de salon de discussion et des systèmes de documentation. Nous examinerons le paramétrage de l'un d'eux, le crochet Email (courriel). Si vous sélectionnez « email » dans la liste déroulante « Add Service », vous verrez un écran de configuration comme **Figure 6-50**.

FIGURE 6-50

Configuration du service Email.

Dans ce cas, si vous cliquez sur le bouton « Add service » (Ajouter le service), un courriel est envoyé à l'adresse électronique que vous avez indiquée à chaque fois que quelqu'un pousse vers le dépôt. Les services peuvent écouter un grand nombre d'événements de différents types mais la plupart n'écoutent que les événements de poussée puis font quelque chose avec ces données.

Si vous utilisez un système et souhaitez l'intégrer avec GitHub, vous devriez vérifier ici s'il existe déjà un service d'intégration disponible. Par exemple, si vous utilisez Jenkins pour lancer des tests sur votre code, vous pouvez activer l'intégration du service intégré Jenkins pour lancer une série de tests à chaque fois que quelqu'un pousse vers votre dépôt.

CROCHETS (*HOOKS*)

Si vous avez besoin de quelque chose de plus spécifique ou que vous voulez intégrer un service ou un site qui n'est pas dans la liste, vous pouvez utiliser à la place le système plus général des crochets. Les crochets de dépôt GitHub sont assez simples. Vous indiquez un URL et GitHub envoie (*post*) des informations par HTTP (*payload*) vers cet URL pour n'importe quel événement que vous souhaitez.

En général, la façon dont cela fonctionne est que vous configurez un petit service Web qui écoute des informations de crochet GitHub puis font quelque chose avec les données reçues.

Pour activer un crochet, vous cliquez sur le bouton « Add webhook » (Ajouter un crochet Web) de la figure **Figure 6-49**. Cela vous redirige vers une page qui ressemble à **Figure 6-51**.

FIGURE 6-51

Configuration d'un crochet Web.

La configuration d'un crochet Web est assez simple. Dans la plupart des cas, vous saisissez simplement un URL et une clé secrète puis cliquez sur « Add webhook ». Il existe quelques options pour choisir l'événement pour lequel GitHub envoie des informations — par défaut seul l'événement `push` envoie des informations lorsque quelqu'un pousse un nouveau code vers une branche de votre dépôt.

Examinons un petit exemple de service Web que vous pourriez configurer pour gérer un crochet Web. Nous utiliserons l'architecture Web Ruby appelée Sinatra car c'est assez concis et vous devriez être capable de voir facilement ce que nous faisons.

Disons que vous voulez recevoir un courriel si une personne précise pousse vers une branche spécifique de notre projet un fichier particulier. Nous pourrions faire facilement cela avec le code suivant :

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]
```

```
  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
     branch == 'ref/heads/special-branch' &&
     files.include?('special-file.txt')

    Mail.deliver do
      from     'tchacon@example.com'
      to       'tchacon@example.com'
      subject  'Scott Changed the File'
      body     "ALARM"
    end
  end
end
```

Ici nous récupérons les informations JSON que GitHub nous délivre et cherchons qui les a poussées, vers quelle branche et quels fichiers ont été touchés dans tous les *commits* qui ont été poussés. Puis nous comparons cela à nos critères et envoyons un courriel si cela correspond.

Afin de développer et tester quelque chose comme cela, il existe une console développeur sympa sur la même fenêtre que celle où vous avez activé le crochet. Vous pouvez afficher les quelques dernières livraisons que GitHub a essayé de faire pour ce crochet Web. Pour chaque crochet, vous pouvez afficher plus d'informations pour savoir quand il s'est exécuté, s'il a réussi et pour connaître les en-têtes et le corps de la requête et de la réponse. Ceci rend incroyablement facile de tester et débugger vos crochets.

FIGURE 6-52

informations de debuggage du crochet web

L'autre fonctionnalité intéressante est que vous pouvez redéclencher la livraison de n'importe quel message pour tester votre service.

Pour plus d'information sur l'écriture de crochets web et tous les différents types d'événement que vous pouvez écouter, rendez-vous à la documentation du Developpeur GitHub à l'adresse ***https://developer.github.com/webhooks/***.

L'interface de programmation (*API*) GitHub

Les services et les crochets vous fournissent un moyen de recevoir des notifications de poussée sur des événements qui arrivent sur vos dépôts, mais que faire si vous avez besoin de plus d'informations sur ces événements ? Que faire si vous avez besoin d'automatiser quelque chose comme ajouter des collaborateurs ou étiqueter des problèmes (*issues*) ?

C'est là que l'Interface de Programmation (*API*) GitHub s'avère utile. GitHub a des tas de points d'entrée sur l'interface d'application pour faire presque tout ce que vous pouvez faire sur le site web de façon automatisée. Dans cette section, nous apprendrons comment s'authentifier et se connecter à l'interface de programmation, comment commenter un problème et comment changer le statut d'une requête de tirage (*pull request*) à travers l'interface de programmation.

Utilisation Basique

La chose la plus basique que vous pouvez faire est une simple requête GET sur une entrée qui ne requiert pas d'authentification. Cela peut être un utilisateur ou une information en lecture seule sur un projet open source. Par exemple, si nous voulons en savoir plus sur un utilisateur appelé « schacon », nous pouvons lancer quelque chose comme ceci :

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# …
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

Il y a des tas de points d'entrée comme celui-ci pour obtenir des informations sur des regroupements, projets, problèmes, *commits* — en fait tout ce que vous pouvez voir sur le site de GitHub. Vous pouvez même utiliser l'interface de programmation pour écrire du texte en Markdown ou trouver un modèle `.gitignore`.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
```

```
*.war
*.ear

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Commenter un problème

Cependant, si vous voulez faire une action sur le site web comme commenter un problème ou une requête de tirage ou si vous voulez voir ou interagir avec du contenu privé, vous aurez besoin de vous authentifier.

Il y a plusieurs moyens de s'authentifier. Vous pouvez utiliser l'authentification basique avec seulement votre nom d'utilisateur et votre mot de passe, mais en général c'est mieux d'utiliser un jeton d'accès personnel. Vous pouvez en générer depuis l'onglet « Applications » de votre page de paramètres.

FIGURE 6-53

Générez votre jeton d'accès depuis l'onglet « Applications » de votre page de paramètres.

On vous demandera le périmètre applicatif que vous voulez pour ce jeton ainsi qu'une description. Assurez-vous d'utiliser une bonne description pour être certain de supprimer le bon jeton quand votre script ou application ne sera plus utilisé.

GitHub ne vous montrera le jeton qu'une seule fois, alors assurez-vous de le copier. Vous pouvez maintenant l'utiliser pour vous authentifier dans votre script au lieu d'utiliser un nom d'utilisateur et un mot de passe. C'est agréable

parce que vous pouvez limiter la portée de ce que vous voulez faire et le jeton est révocable.

Ceci a l'avantage supplémentaire d'augmenter votre limite horaire du nombre d'accès. Sans authentification, vous serez limité à 60 requêtes par heure. Avec authentification, vous pouvez faire jusqu'à 5 000 requêtes par heure.

Maintenant utilisons-le pour faire un commentaire sur un de nos problèmes. Disons que nous voulons laisser un commentaire sur un problème en particulier, le problème n°6. Pour faire cela, nous devons faire une requête HTTP POST à `repos/<utilisateur>/<dépôt>/issues/<num>/comments` avec le jeton que nous venons de générer en tant qu'en-tête "Authorization".

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Maintenant si vous allez à ce problème, vous pouvez voir le commentaire que nous avons posté avec succès comme dans **Figure 6-54**.

FIGURE 6-54

Un commentaire posté depuis l'interface de programmation GitHub

Vous pouvez utiliser l'interface de programmation pour faire à peu près tout ce que vous pouvez faire sur le site web — créer et définir des jalons, assigner des gens à des problèmes ou à des requêtes de tirage, créer et changer des éti-

quettes, accéder à des données de *commit*, créer de nouveaux *commits* et des branches, ouvrir, fermer ou fusionner des requêtes de tirage, créer et éditer des équipes, commenter des lignes de code dans une requête de tirage, chercher dans le site et bien plus encore.

Changer le statut d'une requête de tirage

Nous allons voir un dernier exemple très utile si vous travaillez avec des requêtes de tirage. Chaque *commit* peut avoir un ou plusieurs statuts associés et il y a une interface de programmation pour ajouter et demander ce statut.

La plupart des services d'Intégration Continue et de test utilisent cette interface de programmation pour réagir aux poussées en testant le code qui a été poussé, et en signalant si ce commit a passé tous les tests. Vous pourriez aussi utiliser ceci pour vérifier que le message de validation est formaté proprement, si l'auteur a suivi les recommandations de contribution, si la signature du *commit* est valide — vous pouvez faire autant de choses que vous le souhaitez.

Supposons que vous souhaitez définir un crochet web sur votre dépôt qui atteint un petit service web qui vérifie que le message de validation contient la chaîne `Signed-off-by`.

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # examine chaque message de validation
  push["commits"].each do |commit|

    # cherche la chaîne "Signed-off-by"
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # envoie le statut à GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
```

```
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

Ça devrait être simple à suivre. Dans ce crochet web, nous examinons chaque *commit* qui vient d'être poussé, nous cherchons la chaîne "Signed-off-by" dans le message de validation et enfin nous faisons un POST via HTTP au point d'entrée applicatif `/repos/<utilisateur>/<dépôt>/statuses/<commit_sha>` avec le statut.

Dans ce cas, vous pouvez envoyer un état ("success", "failure", "error"), une description de ce qui s'est passé, un URL cible où l'utilisateur peut aller pour plus d'informations et un « contexte » dans le cas où il y a de multiples statuts pour un seul *commit*. Par exemple, un service de test peut fournir un statut et un service de validation comme celui-ci peut aussi fournir un statut — le champ « contexte » permet de les différencier.

Si quelqu'un ouvre une nouvelle requête de tirage sur GitHub et que ce crochet est opérationnel, vous pouvez voir quelque chose comme **Figure 6-55**.

FIGURE 6-55

Statut de commit via l'interface de programmation.

Vous pouvez voir maintenant une petite coche verte près du *commit* qui contient la chaîne « Signed-off-by » dans le message et une croix rouge pour celui que l'auteur à oublié de signer. Vous pouvez aussi voir que la requête de tirage prend le statut du dernier *commit* de la branche et avertit si c'est un échec. C'est très utile si vous utilisez cette interface de programmation pour des résultats de test pour que vous ne fusionniez pas accidentellement quelque chose où le dernier *commit* échoue aux tests.

Octokit

Bien que nous ayons presque tout fait à travers `curl` et de simples requêtes HTTP dans ces exemples, il existe plusieurs bibliothèques open source qui rendent cette interface de programmation plus idiomatique. Au moment de la rédaction de ce document, les langages supportés incluent Python, Go, Objective-C, Ruby et .NET. Consultez ***http://github.com/octokit*** pour plus d'informations à ce propos, puisqu'ils gèrent une bonne partie de HTTP pour vous.

Heureusement ces outils devraient vous aider à personnaliser et modifier GitHub pour travailler mieux suivant vos méthodes de travail spécifiques. Pour une documentation complète de l'ensemble de l'interface de programmation ainsi que pour des guides pour les tâches habituelles, consultez ***https://developer.github.com***.

Résumé

Vous êtes maintenant un utilisateur de GitHub. Vous savez comment créer un compte, gérer une organisation, créer des dépôts et pousser dessus, contribuer aux projets d'autres utilisateurs et accepter les contributions sur les vôtres. Dans le chapitre suivant, vous découvrirez d'autres puissants outils et des astuces pour faire face à des situations complexes. Vous deviendrez un expert en Git.

Utilitaires Git 7

À présent, vous avez appris les commandes et modes de fonctionnement usuels requis pour gérer et maintenir un dépôt Git pour la gestion de votre code source. Vous avez déroulé les routines de suivi et de validation de fichiers, vous avez exploité la puissance de l'index, de la création et de la fusion de branches locales de travail.

Maintenant, vous allez explorer un certain nombre de fonctionnalités particulièrement efficaces, fonctionnalités que vous utiliserez moins souvent mais dont vous pourriez avoir l'usage à un moment ou à un autre.

Sélection des versions

Git vous permet de faire référence à certains *commits* ou un ensemble de *commits* de différentes façons. Si elles ne sont pas toutes évidentes, il est bon de les connaître.

Révisions ponctuelles

Naturellement, vous pouvez référencer un *commit* par sa signature SHA-1, mais il existe des méthodes plus confortables pour les humains. Cette section présente les méthodes pour référencer un *commit* simple.

Empreinte SHA courte

Git est capable de deviner de quel *commit* vous parlez si vous ne fournissez que quelques caractères du début de la signature, tant que votre SHA-1 partiel comporte au moins 4 caractères et ne correspond pas à plusieurs *commits*. Dans ces conditions, un seul objet correspondra à ce SHA-1 partiel.

Par exemple, pour afficher un *commit* précis, supposons que vous exécutiez `git log` et que vous identifiiez le *commit* où vous avez introduit une fonctionnalité précise.

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

Pour cet exemple, choisissons `1c002dd...`. Si vous affichez le contenu de ce *commit* via `git show`, les commandes suivantes sont équivalentes (en partant du principe que les SHA-1 courts ne sont pas ambigus).

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git peut déterminer une référence SHA-1 tout à la fois la plus courte possible et non ambigüe. Ajoutez l'option `--abbrev-commit` à la commande `git log` et le résultat affiché utilisera des valeurs plus courtes mais uniques ; par défaut Git retiendra 7 caractères et augmentera au besoin :

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

En règle générale, entre 8 et 10 caractères sont largement suffisant pour assurer l'unicité dans un projet.

Un des plus gros projets utilisant Git, le noyau Linux, est un projet plutôt important de plus de 450k *commits* et 3,6 millions d'objets dont les empreintes SHA sont uniques à partir des 11 premiers caractères.

QUELQUES MOTS SUR SHA-1

Beaucoup de gens s'inquiètent qu'à un moment donné ils auront, par des circonstances hasardeuses, deux objets dans leur référentiel de hachage de même empreinte SHA-1. Qu'en est-il réellement ?

S'il vous arrivait de valider un objet qui se hache à la même empreinte SHA-1 qu'un objet existant dans votre référentiel, Git verrait l'objet existant déjà dans votre base de données et présumerait qu'il était déjà enregistré. Si vous essayez de récupérer l'objet de nouveau à un moment donné, vous auriez toujours les données du premier objet.

Quoi qu'il en soit, vous devriez être conscient à quel point ce scénario est ridiculement improbable. Une empreinte SHA-1 porte sur 20 octets soit 160 bits. Le nombre d'objets aléatoires à hacher requis pour assurer une probabilité de collision de 50 % vaut environ 2^{80} (la formule pour calculer la probabilité de collision est `p = (n(n-1)/2) * (1/2^160)`). 2^{80} vaut $1{,}2 \times 10^{24}$ soit 1 million de milliards de milliards. Cela représente 1200 fois le nombre de grains de sable sur Terre.

Voici un exemple pour vous donner une idée de ce qui pourrait provoquer une collision du SHA-1. Si tous les 6,5 milliards d'humains sur Terre programmaient et que chaque seconde, chacun produisait du code équivalent à l'historique entier du noyau Linux (3,6 million d'objets Git) et le poussait sur un énorme dépôt Git, cela prendrait 2 ans pour que ce dépôt contienne assez d'objets pour avoir une probabilité de 50 % qu'une seule collision SHA-1 existe. Il y a une probabilité plus grande que tous les membres de votre équipe de programmation soient attaqués et tués par des loups dans des incidents sans relation la même nuit.

Références de branches

La méthode la plus commune pour désigner un *commit* est une branche y pointant. Dès lors, vous pouvez utiliser le nom de la branche dans toute commande utilisant un objet de type *commit* ou un SHA-1. Par exemple, si vous souhaitez afficher le dernier *commit* d'une branche, les commandes suivantes sont équivalentes, en supposant que la branche `sujet1` pointe sur `ca82a6d` :

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show sujet1
```

Pour connaître l'empreinte SHA sur laquelle pointe une branche ou pour savoir parmi tous les exemples précédents ce que cela donne en terme de SHA, vous pouvez utiliser la commande de plomberie nommée `rev-parse`. Référez-vous à **Chapter 10** pour plus d'informations sur les commandes de plomberie ;

`rev-parse` sert aux opérations de bas niveau et n'est pas conçue pour être utilisée quotidiennement. Quoi qu'il en soit, elle se révèle utile pour comprendre ce qui se passe. Je vous invite à tester `rev-parse` sur votre propre branche.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

Raccourcis RefLog

Git maintient en arrière-plan un historique des références où sont passés HEAD et vos branches sur les derniers mois — ceci s'appelle le *reflog*.

Vous pouvez le consulter avec la commande `git reflog` :

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

À chaque fois que l'extrémité de votre branche est modifiée, Git enregistre cette information pour vous dans son historique temporaire. Vous pouvez référencer d'anciens *commits* avec cette donnée. Si vous souhaitez consulter le n-ième antécédent de votre HEAD, vous pouvez utiliser la référence `@{n}` du reflog, 5 dans cet exemple :

```
$ git show HEAD@{5}
```

Vous pouvez également remonter le temps et savoir où en était une branche à une date donnée. Par exemple, pour savoir où en était la branche `master` hier (*yesterday* en anglais), tapez :

```
$ git show master@{yesterday}
```

Cette technique fonctionne uniquement si l'information est encore présente dans le reflog et vous ne pourrez donc pas le consulter sur des *commits* plus vieux que quelques mois.

Pour consulter le reflog au format `git log`, exécutez: `git log -g` :

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Veuillez noter que le reflog ne stocke que des informations locales, c'est un historique de ce que vous avez fait dans votre dépôt. Les références sont différentes pour un autre dépôt et juste après le clone d'un dépôt, votre reflog sera vide puisque qu'aucune activité n'aura été produite. Exécuter `git show HEAD@{2.months.ago}` ne fonctionnera que si vous avez dupliqué ce projet depuis au moins 2 mois — si vous l'avez dupliqué il y a 5 minutes, vous n'obtiendrez aucun résultat.

Références ancêtres

Une solution fréquente pour référencer un *commit* est d'utiliser son ascendance. Si vous suffixez une référence par ^, Git la résoudra comme étant le parent de cette référence. Supposons que vous consultiez votre historique :

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Alors, vous pouvez consulter le *commit* précédent en spécifiant HEAD^, ce qui signifie « le parent de HEAD » :

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Vous pouvez également spécifier un nombre après ^ — par exemple, d921970^2 signifie « le second parent de d921970 ». Cette syntaxe ne sert que pour les *commits* de fusion, qui ont plus d'un parent. Le premier parent est la branche depuis laquelle vous avez fusionné, et le second est le *commit* de la branche que vous avez fusionnée :

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

Une autre solution courante pour spécifier une référence ancêtre est le ~. Il fait également référence au premier parent, donc HEAD~ et HEAD^ sont équivalents. La différence apparaît si vous spécifiez un nombre. HEAD~2 signifie « le premier parent du premier parent », ou bien « le grand-parent » ; on remonte les premiers parents autant de fois que demandé. Par exemple, dans l'historique précédemment présenté, HEAD~3 serait :

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500
```

```
    ignore *.gem
```

Cela peut aussi s'écrire HEAD^^^, qui là encore est le premier parent du premier parent du premier parent :

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Vous pouvez également combiner ces syntaxes — vous pouvez obtenir le second parent de la référence précédente (en supposant que c'était un *commit* de fusion) en utilisant HEAD~3^2, et ainsi de suite.

Plages de *commits*

À présent que vous pouvez spécifier des *commits* individuels, voyons comment spécifier des plages de *commits*. Ceci est particulièrement pratique pour la gestion des branches — si vous avez beaucoup de branches, vous pouvez utiliser les plages pour répondre à des questions telles que « Quel travail sur cette branche n'ai-je pas encore fusionné sur ma branche principale ? ».

DOUBLE POINT

La spécification de plage de *commits* la plus fréquente est la syntaxe double-point. En gros, cela demande à Git de résoudre la plage des *commits* qui sont accessibles depuis un *commit* mais ne le sont pas depuis un autre. Par exemple, disons que votre historique ressemble à **Figure 7-1**.

FIGURE 7-1

Exemple d'historique pour la sélection de plage.

Si vous voulez savoir ce qui n'a pas encore été fusionné sur votre branche `master` depuis votre branche `experiment`, vous pouvez demander à Git de vous montrer un journal des *commits* avec `master..experiment` — ce qui signifie « tous les *commits* accessibles par `experiment` qui ne le sont pas par `master` ». Dans un souci de brièveté et de clarté de ces exemples, je vais utiliser les lettres des *commits* issus du diagramme à la place de la vraie liste dans l'ordre où ils auraient dû être affichés :

```
$ git log master..experiment
D
C
```

Si, par contre, vous souhaitez voir l'opposé — tous les *commits* dans `master` mais pas encore dans `experiment` — vous pouvez inverser les noms de branches, `experiment..master` vous montre tout ce que `master` accède mais qu'`experiment` ne voit pas :

```
$ git log experiment..master
F
E
```

C'est pratique si vous souhaitez maintenir `experiment` à jour et anticiper les fusions. Un autre cas d'utilisation fréquent consiste à voir ce que vous vous apprêtez à pousser sur une branche distante :

```
$ git log origin/master..HEAD
```

Cette commande vous affiche tous les *commits* de votre branche courante qui ne sont pas sur la branche `master` du dépôt distant `origin`. Si vous exécutez `git push` et que votre branche courante suit `origin/master`, les *commits* listés par `git log origin/master..`HEAD sont les *commits* qui seront transférés sur le serveur. Vous pouvez également laisser tomber une borne de la syntaxe pour faire comprendre à Git que vous parlez de HEAD. Par exemple, vous pouvez obtenir les mêmes résultats que précédemment en tapant `git log origin/master..` — Git utilise HEAD si une des bornes est manquante.

EMPLACEMENTS MULTIPLES

La syntaxe double-point est pratique comme raccourci ; mais peut-être souhaitez-vous utiliser plus d'une branche pour spécifier une révision, comme pour voir quels *commits* sont dans plusieurs branches mais sont absents de la branche courante. Git vous permet cela avec `^` ou `--not` en préfixe de toute référence de laquelle vous ne souhaitez pas voir les *commits*. Les 3 commandes ci-après sont équivalentes :

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

C'est utile car cela vous permet de spécifier plus de 2 références dans votre requête, ce que vous ne pouvez accomplir avec la syntaxe double-point. Par exemple, si vous souhaitez voir les *commits* qui sont accessibles depuis `refA` et `refB` mais pas depuis `refC`, vous pouvez taper ces 2 commandes :

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Ceci vous fournit un système de requêtage des révisions très puissant, pour vous aider à saisir ce qui se trouve sur vos branches.

TRIPLE POINT

La dernière syntaxe majeure de sélection de plage de *commits* est la syntaxe triple-point qui spécifie tous les *commits* accessibles par l'une des deux références, exclusivement. Toujours avec l'exemple d'historique de **Figure 7-1**, si vous voulez voir ce qui se trouve sur `master` ou `experiment` mais pas sur les deux, exécutez :

```
$ git log master...experiment
F
E
D
C
```

Encore une fois, cela vous donne un `log` normal mais ne vous montre les informations que pour ces quatre *commits*, dans l'ordre naturel des dates de validation.

Une option courante à utiliser avec la commande `log` dans ce cas est `--left-right` qui vous montre la borne de la plage à laquelle ce *commit* appartient. Cela rend les données plus utiles :

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Avec ces outils, vous pourrez spécifier à Git les *commits* que vous souhaitez inspecter.

Indexation interactive

Git propose quelques scripts qui rendent les opérations en ligne de commande plus simples. Nous allons à présent découvrir des commandes interactives vous permettant de choisir les fichiers ou les parties d'un fichier à incorporer à un *commit*. Ces outils sont particulièrement pratiques si vous modifiez un grand nombre de fichiers et que vous souhaitez valider ces changements en modifications plus atomiques plutôt que d'un tenant. De la sorte, vous vous assurez que vos *commits* sont des ensembles cohérents de modifications et qu'ils peuvent être facilement revus par vos collaborateurs. Si vous exécutez `git add` avec l'option `-i` ou `--interactive`, Git entre en mode interactif et affiche quelque chose comme :

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

Vous vous apercevrez que cette commande propose une vue bien différente de votre index ; en gros, c'est la même information que vous auriez obtenue avec `git status` mais en plus succinct et plus instructif. Cela liste les modifications que vous avez indexées à gauche et celles hors index à droite.

En dessous vient la section des commandes (**Commands**). Vous aurez accès à un certain nombre d'actions, notamment indexer des fichiers, les enlever de l'index, indexer des parties de fichiers, ajouter des fichiers non indexés, et vérifier les différences de ce que vous avez indexé.

Indexation et désindexation des fichiers

Si vous tapez 2 ou u au prompt `What now>`, le script vous demande quels fichiers vous voulez indexer :

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Pour indexer les fichiers TODO et index.html, vous pouvez taper ces nombres :

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Le caractère `*` au début de la ligne de chaque fichier indique que celui-ci est sélectionné. Si vous tapez Entrée sur l'invite `Update>>`, Git prend tout ce qui est sélectionné et l'indexe pour vous :

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
```

```
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

À présent, vous pouvez voir que les fichiers TODO et index.html sont indexés (*staged* en anglais) et que `simplegit.rb` ne l'est toujours pas. Si vous souhaitez enlever de l'index le fichier TODO, utilisez 3 (ou `r` pour revert en anglais) :

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Un aperçu rapide à votre statut Git et vous pouvez voir que vous avez enlevé le fichier TODO de l'index :

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Pour voir la modification que vous avez indexée, utilisez 6 ou `d` (pour différence). Cela vous affiche la liste des fichiers indexés et vous pouvez choisir ceux pour lesquels vous voulez consulter la différence. C'est équivalent à `git diff --cached` en ligne de commande :

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

Avec ces commandes élémentaires, vous pouvez utiliser l'ajout interactif pour manipuler votre index un peu plus facilement.

Indexations partielles

Git est également capable d'indexer certaines parties d'un fichier. Par exemple, si vous modifiez en deux endroits votre fichier `simplegit.rb` et que vous souhaitez indexer une modification seulement, cela peut se faire très aisément avec Git. En mode interactif, tapez 5 ou p (pour *patch* en anglais). Git vous demandera quels fichiers vous voulez indexer partiellement, puis, pour chacun des fichiers sélectionnés, il affichera les parties du fichier où il y a des différences et vous demandera si vous souhaitez les indexer, une par une :

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end
```

```
   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

À cette étape, vous disposez de bon nombre d'options. `?` vous liste les actions possibles dont voici une traduction :

```
Indexer cette partie [y,n,a,d,/,j,J,g,e,?]? ?
y - indexer cette partie
n - ne pas indexer cette partie
a - indexer cette partie et toutes celles restantes dans ce fichier
d - ne pas indexer cette partie ni aucune de celles restantes dans ce fichier
g - sélectionner une partie à voir
/ - chercher une partie correspondant à la regexp donnée
j - laisser cette partie non décidée, voir la prochaine partie non encore décidée
J - laisser cette partie non décidée, voir la prochaine partie
k - laisser cette partie non décidée, voir la partie non encore décidée précédente
K - laisser cette partie non décidée, voir la partie précédente
s - couper la partie courante en parties plus petites
e - modifier manuellement la partie courante
? - afficher l'aide
```

En règle générale, vous choisirez `y` ou `n` pour indexer ou non chacun des blocs, mais tout indexer pour certains fichiers ou remettre à plus tard le choix pour un bloc peut également être utile. Si vous indexez une partie d'un fichier et une autre non, votre statut ressemblera à peu près à ceci :

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

Le statut pour le fichier `simplegit.rb` est intéressant. Il vous montre que quelques lignes sont indexées et d'autres non. Vous avez partiellement indexé ce fichier. Dès lors, vous pouvez quitter l'ajout interactif et exécuter `git commit` pour valider les fichiers partiellement indexés.

Enfin, vous pouvez vous passer du mode interactif pour indexer partiellement un fichier ; vous pouvez faire de même avec `git add -p` ou `git add --patch` en ligne de commande.

De plus, vous pouvez utiliser le mode patch pour réinitialiser partiellement des fichiers avec la commande `reset --patch`, pour extraire des parties de fichiers avec `checkout --patch` et pour remiser des parties de fichiers avec

`stash save --patch`. Nous explorerons plus en détail chacune des ces commandes quand nous aborderons les usages avancés de ces commandes.

Remisage et nettoyage

Souvent, lorsque vous avez travaillé sur une partie de votre projet, les choses sont dans un état instable mais vous voulez changer de branche pour travailler momentanément sur autre chose. Le problème est que vous ne voulez pas valider un travail à moitié fait seulement pour pouvoir y revenir plus tard. La réponse à cette problématique est la commande `git stash`.

Remiser prend l'état en cours de votre répertoire de travail, c'est-à-dire les fichiers modifiés et l'index, et l'enregistre dans la pile des modifications non finies que vous pouvez ré-appliquer à n'importe quel moment.

Remiser votre travail

Pour démontrer cette possibilité, allez dans votre projet et commencez à travailler sur quelques fichiers et indexez l'un de ces changements. Si vous exécutez `git status`, vous pouvez voir votre état modifié :

```
$ git status
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

        modifié :   index.html

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

        modifié :   lib/simplegit.rb
```

À ce moment-là, vous voulez changer de branche, mais vous ne voulez pas encore valider ce travail ; vous allez donc remiser vos modifications. Pour créer une nouvelle remise sur votre pile, exécutez `git stash` :

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Votre répertoire de travail est propre :

```
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

À ce moment, vous pouvez facilement changer de branche et travailler autre part ; vos modifications sont conservées dans votre pile. Pour voir quelles remises vous avez sauvegardées, vous pouvez utiliser la commande `git stash list` :

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

Dans ce cas, deux remises ont été créées précédemment, vous avez donc accès à trois travaux remisés différents. Vous pouvez ré-appliquer celui que vous venez juste de remiser en utilisant la commande affichée dans la sortie d'aide de la première commande de remise : `git stash apply`. Si vous voulez appliquer une remise plus ancienne, vous pouvez la spécifier en la nommant, comme ceci : `git stash apply stash@{2}`. Si vous ne spécifiez pas une remise, Git présume que vous voulez la remise la plus récente et essaye de l'appliquer.

```
$ git stash apply
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la

     modified:   index.html
     modified:   lib/simplegit.rb
```

Vous pouvez observer que Git remodifie les fichiers non validés lorsque vous avez créé la remise. Dans ce cas, vous aviez un répertoire de travail propre lorsque vous avez essayé d'appliquer la remise et vous l'avez fait sur la même branche que celle où vous l'aviez créée ; mais avoir un répertoire de travail propre et l'appliquer sur la même branche n'est pas nécessaire pour réussir à appliquer une remise. Vous pouvez très bien créer une remise sur une branche, changer de branche et essayer d'appliquer ces modifications. Vous pouvez même avoir

des fichiers modifiés et non validés dans votre répertoire de travail quand vous appliquez une remise, Git vous indique les conflits de fusions si quoi que ce soit ne s'applique pas proprement.

Par défaut, les modifications de vos fichiers sont ré-appliquées, mais pas les indexations. Pour cela, vous devez exécuter la commande `git stash apply` avec l'option `--index` pour demander à Git d'essayer de ré-appliquer les modifications de votre index. Si vous exécutez cela à la place de la commande précédente, vous vous retrouvez dans la position d'origine précédent la remise :

```
$ git stash apply --index
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

     modifié :   index.html

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

     modified:   lib/simplegit.rb
```

L'option `apply` essaye seulement d'appliquer le travail remisé, vous aurez toujours la remise dans votre pile. Pour la supprimer, vous pouvez exécuter `git stash drop` avec le nom de la remise à supprimer :

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Vous pouvez également exécuter `git stash pop` pour appliquer et supprimer immédiatement la remise de votre pile.

Remisage créatif

Il existe des variantes de remisages qui peuvent s'avérer utiles. La première option assez populaire est l'option `--keep-index` de la commande `stash save`. Elle indique à Git de ne pas remiser ce qui aurait été déjà indexé au moyen de la commande `git add`.

C'est particulièrement utile si vous avez réalisé des modifications mais souhaitez ne valider que certaines d'entre elles et gérer le reste plus tard.

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index fil
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Une autre option utile de `stash` est la possibilité de remiser les fichiers non suivis aussi bien que les fichiers suivis. Par défaut, `git stash` ne sauve que les fichiers qui sont déjà suivis ou indexés. Si vous spécifiez l'option `--include-untracked` ou `-u`, Git remisera aussi les fichiers non-suivis du répertoire de travail.

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index fil
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Enfin, si vous ajoutez l'option `--patch`, Git ne remisera pas tout le contenu modifié, mais vous invitera à sélectionner interactivement les modifications que vous souhaitez remiser et celles que vous souhaiter conserver dans la copie de travail.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
```

```
        end
      end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Défaire l'effet d'une remise

Dans certains cas, il est souhaitable de pouvoir appliquer une modification remisée, réaliser d'autres modifications, puis défaire les modifications de la remise. Git ne fournit pas de commande `stash unapply` mais il est possible d'obtenir le même effet en extrayant les modifications qui constituent la remise et en appliquant leur inverse :

```
$ git stash show -p stash@{0} | git apply -R
```

Ici aussi, si la remise n'est pas indiquée, Git utilise la plus récente.

```
$ git stash show -p | git apply -R
```

La création d'un alias permettra d'ajouter effectivement la commande `stash-unapply` à votre Git. Par exemple :

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... travail, travail, travail
$ git stash-unapply
```

Créer une branche depuis une remise

Si vous remisez votre travail, et l'oubliez pendant un temps en continuant sur la branche où vous avez créé la remise, vous pouvez avoir un problème en réappliquant le travail. Si l'application de la remise essaye de modifier un fichier

que vous avez modifié depuis, vous allez obtenir des conflits de fusion et vous devrez essayer de les résoudre. Si vous voulez un moyen plus facile de tester une nouvelle fois les modifications remisées, vous pouvez exécuter `git stash branch` qui créera une nouvelle branche à votre place, récupérant le *commit* où vous étiez lorsque vous avez créé la remise, ré-appliquera votre travail dedans, et supprimera finalement votre remise si cela a réussi :

```
$ git stash branch testchanges
Basculement sur la nouvelle branche 'testchanges'
Sur la branche testchanges
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

     modifié :   index.html

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

     modified:   lib/simplegit.rb

refs/stash@{0} supprimé (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

C'est un bon raccourci pour récupérer facilement du travail remisé et pouvoir travailler dessus dans une nouvelle branche.

Nettoyer son répertoire de travail

Enfin, vous pouvez ne pas souhaiter remiser certain fichiers de votre répertoire de travail, mais simplement vous en débarrasser. La commande `git clean` s'en chargera pour vous.

Le besoin le plus commun pourra être d'éliminer les scories générées par les fusions ou les outils externes ou d'éliminer les artefacts de compilation pour pouvoir relancer une compilation propre.

Faîtes néanmoins très attention avec cette commande car elle supprime des fichiers non-suivis de votre répertoire de travail. Si vous changez d'avis, il est souvent impossible de récupérer après coup le contenu de ces fichiers. Une option plus sécurisée consiste à lancer `git stash --all` pour tout sauvegarder dans une remise.

En supposant que vous souhaitez réellement éliminer les scories et nettoyer votre répertoire de travail, vous pouvez lancer `git clean`. Pour supprimer tous les fichiers non-suivis, vous pouvez lancer `git clean -f -d`, qui effacera aus-

si tout sous-répertoire vide. L'option `-f` signifie « force », soit « fais-le réellement ».

Si vous souhaitez visualiser ce qui serait fait, vous pouvez lancer la commande avec l'option `-n` qui signifie « fais-le à blanc et montre-moi ce qui *serait* supprimé ».

```
$ git clean -d -n
Supprimerait test.o
Supprimerait tmp/
```

Par défaut, la commande `git clean` ne va supprimer que les fichiers non-suivis qui ne sont pas ignorés. Tout fichier qui correspond à un motif de votre fichier `.gitignore` ou tout autre fichier similaire ne sera pas supprimé. Si vous souhaitez supprimer aussi ces fichiers, comme par exemple les fichiers `.o` généré par un compilateur pour faire une compilation totale, vous pouvez ajouter l'option `-x` à la commande de nettoyage.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Supprimerait build.TMP
Supprimerait tmp/

$ git clean -n -d -x
Supprimerait build.TMP
Supprimerait test.o
Supprimerait tmp/
```

Si vous ne savez pas ce que la commande `git clean` va effectivement supprimer, lancez-la une première fois avec `-n` par sécurité avant de transformer le `-n` en `-f` et nettoyer définitivement. Un autre choix pour s'assurer de ce qui va être effacé consiste à lancer la commande avec l'option `-i` ou `--interactive`.

La commande sera lancée en mode interactif.

```
$ git clean -x -i
Supprimerait les éléments suivants :
  build.TMP  test.o
*** Commandes ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each
```

```
    6: help
Et maintenant ?>
```

De cette manière, vous pouvez détailler chaque fichier individuellement ou spécifier un motif pour la suppression interactive.

Signer votre travail

Git est cryptographiquement sûr, mais il n'est pas infaillible. Si vous récupérez le travail d'autres personnes sur Internet et souhaitez vérifier que les *commits* ont effectivement une source de confiance, Git propose quelques méthodes pour signer et vérifier ceci au moyen de GPG.

Introduction à GPG

Avant tout, si vous voulez pouvoir signer quoique ce soit, vous devez avoir un GPG configuré et une clé personnelle.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

Si vous n'avez pas de clé, vous pouvez en générer une avec la commande `gpg --gen-key`.

```
gpg --gen-key
```

A présent que vous avez une clé privée permettant de signer, vous pouvez configurer Git pour l'utiliser pour signer diverses choses en renseignant le paramètre de configuration `user.signingkey`.

```
git config --global user.signingkey 0A46826A
```

A partir de maintenant, Git utilisera par défaut votre clé pour signer les étiquettes et les *commits* que vous souhaitez.

Signer des étiquettes

Avec votre clé privée GPG renseignée, vous pouvez signer des étiquettes. Tout ce que vous avez à faire, c'est remplacer `-a` par `-s` :

```
$ git tag -s v1.5 -m 'mon étiquette signée 1.5'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Si vous lancez `git show` sur cette étiquette, vous pouvez voir votre signature GPG attachée :

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

mon étiquette signée 1.5
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

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

    changed the version number
```

Verifier des étiquettes

Pour vérifier une étiquette signée, vous utilisez `git tag -v [nom-de-l-etiquette]`. Cette commande utilise GPG pour vérifier la signature. Vous devez posséder la clé publique du signataire dans votre trousseau pour que cela fonctionne.

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

Si vous ne possédez pas la clé publique du signataire, vous obtiendrez plutôt quelque chose comme :

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Signer des *commits*

Dans les versions les plus récentes de Git (à partir de v1.7.9), vous pouvez maintenant signer aussi les *commits* individuels. Si signer directement des *commits* au lieu d'étiquettes vous intéresse, tout ce que vous avez à faire est d'ajouter l'option `-S` à votre commande `git commit`.

```
$ git commit -a -S -m 'commit signé'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] commit signé
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Pour visualiser et vérifier ces signatures, il y a l'option `--show-signature` pour `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    commit signé
```

En complément, vous pouvez configurer `git log` pour vérifier toutes les signatures qu'il trouvera et les montrer grâce au formatage `%G?`.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  commit signé
ca82a6d N Scott Chacon  changed the verison number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Ici nous pouvons voir que seul le dernier *commit* est signé et valide tandis que les précédents ne le sont pas.

Depuis Git 1.8.3, `git merge` et `git pull` peuvent vérifier et annuler une fusion d'un *commit* qui ne porte pas de signature GPG de confiance, avec la commande `--verify-signatures`.

Si vous utilisez cette option lors de la fusion d'une branche et qu'elle contient des *commits* qui ne sont pas signés et valides, la fusion échouera.

```
$ git merge --verify-signatures non-verify
fatal: La validation ab06180 n'a pas de signature GPG.
```

Si la fusion ne contient que des *commits* signés valides, la commande de fusion vous montrera toutes les signatures vérifiées et démarrera la fusion proprement dite.

```
$ git merge --verify-signatures signed-branch
La validation 13ad65e a une signature GPG correcte par Scott Chacon (Git signing key) <schac
Mise à jour 5c3386c..13ad65e
Avance rapide
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Vous pouvez aussi utiliser l'option `-S` avec la commande `git merge` elle-même pour signer le *commit* de fusion. L'exemple suivant vérifie que tous les *commits* dans la branche à fusionner sont signés et de plus signe le *commit* de fusion résultant.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e a une signature GPG correcte par Scott Chacon (Git signing key) <sc

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Tout le monde doit signer

Signer les étiquettes et les *commits*, c'est bien mais si vous décidez d'utiliser cette fonction dans votre méthode de travail, il faudra s'assurer que tous les membres de votre équipe comprennent comment s'y prendre. Sinon, vous allez devoir passer du temps à aider les personnes à réécrire leurs *commits* en version signée. Assurez-vous de bien comprendre GPG et les bénéfices de la signature avant d'adopter cette pratique dans vos méthodes de travail.

Recherche

Quelle que soit la taille de votre code, vous avez souvent besoin de chercher où une fonction est appelée ou définie, ou de retrouver l'historique d'une méthode. Git fournit quelques outils permettant rapidement de rechercher dans le code et les *commits* stockés dans votre base de données. Nous allons en détailler quelques uns.

Git grep

Git est livré avec une commande appelée `grep` qui permet de rechercher facilement une chaîne de caractères ou une expression régulière dans une arborescence validée ou dans le répertoire de travail. Pour tous les exemples qui suivent, nous allons utiliser le dépôt de Git lui-même.

Par défaut, `git grep` recherche dans le répertoire de travail. Vous pouvez passer l'option `-n` pour afficher les numéros des lignes des correspondances.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

La commande `grep` peut être enrichie grâce à un certain nombre d'options intéressantes.

Par exemple, pour l'appel précédent, vous pouvez indiquer à Git de résumer le résultat en ne montrant que les fichiers et le nombre de correspondances au moyen de l'option `--count` :

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

Si vous souhaitez voir dans quelle méthode ou fonction la correspondance a été trouvée, vous pouvez passer l'option `-p` :

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char *end,
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int *tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
```

Ici, on peut voir que `gmtime_r` est appelée dans les fonctions `match_multi_number` et `match_digit` du fichier `date.c`.

Vous pouvez aussi rechercher des combinaisons plus complexes de chaînes de caractères avec l'option `--and` qui force plusieurs correspondances sur la même ligne. Par exemple, recherchons toutes les lignes qui définissent une constante qui contient au choix « LINK » ou « BUF_MAX » dans la base de code de Git avant la version 1.8.0.

Ici, nous allons utiliser les options `--break` et `--heading` qui aident à découper le résultat dans un format plus digeste.

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

La commande `git grep` a quelques avantages sur les commandes de recherche normales telles que `grep` et `ack`. Le premier est qu'elle est vraiment rapide, le second est qu'elle vous permet de rechercher dans n'importe quelle arborescence Git, pas seulement la copie de travail. Comme nous l'avons vu dans l'exemple ci-dessus, nous avons cherché des termes dans une version ancienne du code source de Git, pas dans la dernière version extraite.

Recherche dans le journal Git

Peut-être ne cherchez-vous pas **où** un terme apparaît, mais plutôt **quand** il existait ou fut introduit. La commande `git log` comprend un certain nombre d'outils puissants pour trouver des *commits* spécifiques par le contenu de leurs messages ou le contenu des diffs qu'ils introduisent.

Si vous voulez trouver par exemple quand la constante `ZLIB_BUF_MAX` a été initialement introduite, nous pouvons indiquez à Git de ne montrer que les *commits* qui soit ajoutent soit retirent la chaîne avec l'option `-S`.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Si nous regardons la diff de ces *commits*, nous pouvons voir que dans `ef49a7a`, la constante a été introduite et qu'elle a été modifiée dans `e01503b`.

Si vous devez être plus spécifique, vous pouvez fournir une expression régulière à rechercher avec l'option `-G`.

RECHERCHE DES ÉVOLUTIONS D'UNE LIGNE

Un autre outil avancé de recherche dans l'historique qui peut s'avérer très utile est la recherche de l'historique d'une ligne. C'est une addition assez récente et peu connue, mais elle peut être très efficace. On l'active avec l'option `-L` de `git log` et elle vous montre l'historique d'une fonction ou d'une ligne dans votre base de code.

Par exemple, si nous souhaitions voir toutes les modifications réalisées sur la fonction `git_deflate_bound` dans le fichier `zlib.c`, nous pourrions lancer `git log -L :git_deflate_bound:zlib.c`. Cette commande va essayer de déterminer les limites de cette fonction et de rechercher dans l'historique chaque modification réalisée sur la fonction comme une série de patchs jusqu'au moment de sa création.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
```

```
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

Si Git ne peut pas déterminer comment trouver la fonction ou la méthode dans votre langage de programmation, vous pouvez aussi fournir une regex. Par exemple, cela aurait donné le même résultat avec `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. Vous auriez pu aussi spécifier un intervalle de lignes ou un numéro de ligne et vous auriez obtenu le même type de résultat.

Réécrire l'historique

Bien souvent, lorsque vous travaillez avec Git, vous souhaitez modifier votre historique de validation pour une raison quelconque. Une des choses merveilleuses de Git est qu'il vous permet de prendre des décisions le plus tard possible. Vous pouvez décider quels fichiers vont dans quel *commit* avant que vous ne validiez l'index, vous pouvez décider que vous ne voulez pas encore montrer que vous travaillez sur quelque chose avec les remises, et vous pouvez réécrire les *commits* déjà sauvegardés pour qu'ils ressemblent à quelque chose d'autre. Cela peut signifier changer l'ordre des *commits*, modifier les messages ou modifier les fichiers appartenant au *commit*, rassembler ou scinder des *commits*, ou supprimer complètement des *commits* ; tout ceci avant de les partager avec les autres.

Dans cette section, nous expliquerons comment accomplir ces tâches très utiles pour que vous puissiez remodeler votre historique de validation comme vous le souhaitez avant de le partager avec autrui.

Modifier la dernière validation

Modifier votre dernière validation est probablement la réécriture de l'historique que vous allez utiliser le plus souvent. Vous voudrez souvent faire deux choses

basiques à votre dernier *commit* : modifier le message de validation ou changer le contenu que vous avez enregistré en ajoutant, modifiant ou supprimant des fichiers.

Si vous voulez seulement modifier votre dernier message de validation, c'est vraiment simple :

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

Cela ouvre votre éditeur de texte contenant votre dernier message, prêt à être modifié. Lorsque vous sauvegardez et fermez l'éditeur, Git enregistre la nouvelle validation contenant le message et en fait votre dernier *commit*.

Si vous voulez modifier le contenu de votre validation en ajoutant ou modifiant des fichiers, sûrement parce que vous avez oublié d'ajouter les fichiers nouvellement créés quand vous avez validé la première fois, la procédure fonctionne grosso-modo de la même manière. Vous indexez les modifications que vous voulez en exécutant `git add` ou `git rm`, et le prochain `git commit --amend` prendra votre index courant et en fera le contenu de votre nouvelle validation.

Vous devez être prudent avec cette technique car votre modification modifie également le SHA-1 du *commit*. Cela ressemble à un tout petit `rebase`. Ne modifiez pas votre dernière validation si vous l'avez déjà publiée !

Modifier plusieurs messages de validation

Pour modifier une validation qui est plus loin dans votre historique, vous devez utiliser des outils plus complexes. Git ne contient pas d'outil de modification d'historique, mais vous pouvez utiliser l'outil `rebase` pour rebaser une suite de *commits* depuis la branche HEAD plutôt que de les déplacer vers une autre branche. Avec l'outil `rebase` interactif, vous pouvez vous arrêter après chaque *commit* que vous voulez modifier et changer le message, ajouter des fichiers ou quoi que ce soit que vous voulez. Vous pouvez exécuter `rebase` interactivement en ajoutant l'option `-i` à `git rebase`. Vous devez indiquer jusqu'à quand remonter dans votre historique en donnant à la commande le *commit* sur lequel vous voulez vous rebaser.

Par exemple, si vous voulez modifier les 3 derniers messages de validation ou n'importe lequel des messages dans ce groupe, vous fournissez à `git rebase -i` le parent du dernier *commit* que vous voulez éditer, qui est `HEAD~2^` or `HEAD~3`. Il peut être plus facile de se souvenir de `~3`, car vous essayez de modifier les 3 derniers *commits*, mais gardez à l'esprit que vous désignez le 4e, le parent du dernier *commit* que vous voulez modifier :

```
$ git rebase -i HEAD~3
```

Souvenez-vous également que ceci est une commande de rebasage, chaque *commit* inclus dans l'intervalle `HEAD~3..HEAD` sera réécrit, que vous changiez le message ou non. N'incluez pas, dans cette commande, de *commit* que vous avez déjà poussé sur un serveur central. Le faire entraînera la confusion chez les autres développeurs en leur fournissant une version altérée des mêmes modifications.

Exécuter cette commande vous donne la liste des validations dans votre éditeur de texte, ce qui ressemble à :

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# 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
```

Il est important de signaler que les *commits* sont listés dans l'ordre inverse de celui que vous voyez normalement en utilisant la commande `log`. Si vous exécutez la commande `log`, vous verrez quelque chose de ce genre :

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Remarquez l'ordre inverse. Le rebasage interactif va créer un script à exécuter. Il commencera au *commit* que vous spécifiez sur la ligne de commande (HEAD~3) et refera les modifications introduites dans chacun des *commits* du début à la fin. Il ordonne donc le plus vieux au début, plutôt que le plus récent, car c'est celui qu'il refera en premier.

Vous devez éditer le script afin qu'il s'arrête au *commit* que vous voulez modifier. Pour cela, remplacer le mot « pick » par le mot « edit » pour chaque *commit* après lequel vous voulez que le script s'arrête. Par exemple, pour modifier uniquement le message du troisième *commit*, vous modifiez le fichier pour ressembler à :

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Au moment où vous sauvegardez et quittez l'éditeur, Git revient au dernier *commit* de cette liste et vous laisse sur une ligne de commande avec le message suivant :

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

Ces instructions vous disent exactement quoi faire. Entrez :

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

Modifiez le message de *commit* et quittez l'éditeur. Puis exécutez :

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

Cette commande appliquera les deux autres *commits* automatiquement. Si vous remplacez « pick » en « edit » sur plusieurs lignes, vous pouvez répéter ces étapes pour chaque *commit* que vous avez marqué pour modification. Chaque

fois, Git s'arrêtera, vous laissant modifier le *commit* et continuera lorsque vous aurez fini.

Réordonner les *commits*

Vous pouvez également utiliser les rebasages interactifs afin de réordonner ou supprimer entièrement des *commits*. Si vous voulez supprimer le *commit* « added cat-file » et modifier l'ordre dans lequel les deux autres *commits* se trouvent dans l'historique, vous pouvez modifier le script de rebasage :

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

afin qu'il ressemble à ceci :

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Lorsque vous sauvegardez et quittez l'éditeur, Git remet votre branche au niveau du parent de ces *commits*, applique `310154e` puis `f7f3f6d` et s'arrête. Vous venez de modifier l'ordre de ces *commits* et de supprimer entièrement le *commit* « added cat-file ».

Écraser un *commit*

Il est également possible de prendre une série de *commits* et de les rassembler en un seul avec l'outil de rebasage interactif. Le script affiche des instructions utiles dans le message de rebasage :

```
#
# 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
```

Si, à la place de « *pick* » ou « *edit* », vous spécifiez « *squash* », Git applique cette modification et la modification juste précédente et fusionne les messages de validation. Donc, si vous voulez faire un seul *commit* de ces trois validations, vous faites en sorte que le script ressemble à ceci :

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Lorsque vous sauvegardez et quittez l'éditeur, Git applique ces trois modifications et vous remontre l'éditeur contenant maintenant la fusion des 3 messages de validation :

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

Lorsque vous sauvegardez cela, vous obtenez un seul *commit* amenant les modifications des trois *commits* précédents.

Diviser un *commit*

Pour diviser un *commit*, il doit être défait, puis partiellement indexé et validé autant de fois que vous voulez pour en finir avec lui. Par exemple, supposons que vous voulez diviser le *commit* du milieu dans l'exemple des trois *commits* précédents. Plutôt que « *updated README formatting and added blame* », vous voulez le diviser en deux *commits* : « *updated README formatting* » pour le premier, et « *added blame* » pour le deuxième. Vous pouvez le faire avec le script

`rebase -i` en remplaçant l'instruction sur le *commit* que vous voulez diviser en « *edit* » :

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Puis, lorsque le script vous laisse accès à la ligne de commande, vous pouvez annuler (**reset**) ce *commit* et revalider les modifications que vous voulez pour créer plusieurs *commits*. En reprenant l'exemple, lorsque vous sauvegardez et quittez l'éditeur, Git revient au parent de votre premier *commit* de votre liste, applique le premier *commit* (`f7f3f6d`), applique le deuxième (`310154e`), et vous laisse accès à la console. Là, vous pouvez faire une réinitialisation mélangée (**mixed reset**) de ce *commit* avec `git reset HEAD^`, qui défait ce *commit* et laisse les fichiers modifiés non indexés. Maintenant, vous pouvez indexer et valider les fichiers sur plusieurs validations, et exécuter `git rebase --continue` quand vous avez fini :

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git applique le dernier *commit* (`a5f4a0d`) de votre script, et votre historique ressemblera alors à :

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Une fois encore, ceci modifie les empreintes SHA-1 de tous les *commits* dans votre liste, soyez donc sûr qu'aucun *commit* de cette liste n'ait été poussé dans un dépôt partagé.

L'option nucléaire : `filter-branch`

Il existe une autre option de la réécriture d'historique que vous pouvez utiliser si vous avez besoin de réécrire un grand nombre de *commits* d'une manière scriptable ; par exemple, modifier globalement votre adresse mail ou supprimer un fichier de tous les *commits*. La commande est `filter-branch`, et elle peut réécrire des pans entiers de votre historique, vous ne devriez donc pas l'utiliser à moins que votre projet ne soit pas encore public ou que personne n'ait encore travaillé sur les *commits* que vous allez réécrire. Cependant, cela peut être très utile. Vous allez maintenant apprendre quelques usages communs pour vous donner une idée de ses capacités.

SUPPRIMER UN FICHIER DE CHAQUE *COMMIT*

Cela arrive assez fréquemment. Quelqu'un a accidentellement validé un énorme fichier binaire avec une commande `git add .` irréfléchie, et vous voulez le supprimer partout. Vous avez peut-être validé un fichier contenant un mot de passe et vous voulez rendre votre projet open source. `filter-branch` est l'outil que vous voulez probablement utiliser pour nettoyer votre historique entier. Pour supprimer un fichier nommé « passwords.txt » de tout votre historique, vous pouvez utiliser l'option `--tree-filter` de `filter-branch` :

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

L'option `--tree-filter` exécute la commande spécifiée pour chaque *commit* et le revalide ensuite. Dans le cas présent, vous supprimez le fichier nommé « passwords.txt » de chaque contenu, qu'il existait ou non. Si vous voulez supprimer tous les fichiers temporaires des éditeurs validés accidentellement, vous pouvez exécuter une commande telle que `git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD`.

Vous pourrez alors regarder Git réécrire l'arbre des *commits* et revalider à chaque fois, pour finir en modifiant la référence de la branche. C'est généralement une bonne idée de le faire dans un branche de test puis de faire une réinitialisation forte (**hard-reset**) de votre branche `master` si le résultat vous convient. Pour exécuter `filter-branch` sur toutes vos branches, vous pouvez ajouter `--all` à la commande.

FAIRE D'UN SOUS-RÉPERTOIRE LA NOUVELLE RACINE

Supposons que vous avez importé votre projet depuis un autre système de gestion de configuration et que vous avez des sous-répertoires qui n'ont aucun sens (trunk, tags, etc.). Si vous voulez faire en sorte que le sous-répertoire `trunk` soit la nouvelle racine de votre projet pour tous les *commits*, `filter-branch` peut aussi vous aider à le faire :

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Maintenant votre nouvelle racine est remplacée par le contenu du répertoire `trunk`. De plus, Git supprimera automatiquement les *commits* qui n'affectent pas ce sous-répertoire.

MODIFIER GLOBALEMENT L'ADRESSE MAIL

Un autre cas habituel est que vous oubliez d'exécuter `git config` pour configurer votre nom et votre adresse mail avant de commencer à travailler, ou vous voulez peut-être rendre un projet du boulot open source et donc changer votre adresse professionnelle pour celle personnelle. Dans tous les cas, vous pouvez modifier l'adresse mail dans plusieurs *commits* avec un script `filter-branch`. Vous devez faire attention de ne changer que votre adresse mail, utilisez donc `--commit-filter` :

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

Cela passe sur chaque *commit* et le réécrit pour avoir votre nouvelle adresse. Mais puisque les *commits* contiennent l'empreinte SHA-1 de leur parent, cette commande modifie tous les *commits* dans votre historique, pas seulement ceux correspondant à votre adresse mail.

Reset démystifié

Avant d'aborder des outils plus spécialisés, parlons un instant de `reset` et `checkout`. Ces commandes sont deux des plus grandes sources de confusion à leur premier contact. Elles permettent de faire tant de choses et il semble impossible de les comprendre et les employer correctement. Pour ceci, nous vous recommandons une simple métaphore.

Les trois arbres

Le moyen le plus simple de penser à `reset` et `checkout` consiste à représenter Git comme un gestionnaire de contenu de trois arborescences différentes. Par « arborescence », il faut comprendre « collection de fichiers », pas spécifiquement structure de données. Il existe quelques cas pour lesquels l'index ne se comporte pas exactement comme une arborescence, mais pour ce qui nous concerne, c'est plus simple de l'imaginer de cette manière pour le moment.

Git, comme système, gère et manipule trois arbres au cours de son opération normale :

Arbre	Rôle
HEAD	instantané de la dernière validation, prochain parent
Index	instantané proposé de la prochaine validation
Répertoire de travail	bac à sable

HEAD

HEAD est un pointeur sur la référence de la branche actuelle, qui est à son tour un pointeur sur le dernier *commit* réalisé sur cette branche. Ceci signifie que HEAD sera le parent du prochain *commit* à créer. C'est généralement plus simple de penser HEAD comme l'instantané de **votre dernière validation**.

En fait, c'est assez simple de visualiser ce à quoi cet instantané ressemble. Voici un exemple de liste du répertoire et des sommes de contrôle SHA-1 pour chaque fichier de l'instantané HEAD :

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700
```

```
initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

Les commandes `cat-file` et `ls-tree` sont des commandes de « plomberie » qui sont utilisées pour des activités de base niveau et ne sont pas réellement utilisées pour le travail quotidien, mais elles nous permettent de voir ce qui se passe ici.

L'INDEX

L'index est votre **prochain commit proposé**. Nous avons aussi fait référence à ce concept comme la « zone de préparation » de Git du fait que c'est ce que Git examine lorsque vous lancez `git commit`.

Git remplit cet index avec une liste de tous les contenus des fichiers qui ont été extraits dans votre copie de travail et ce qu'ils contenaient quand ils ont été originellement extraits. Vous pouvez alors remplacer certains de ces fichiers par de nouvelles versions de ces mêmes fichiers, puis `git commit` convertit cela en arborescence du nouveau *commit*.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0       README
100644 8f94139338f9404f26296befa88755fc2598c289 0       Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0       lib/simplegit.rb
```

Encore une fois, nous utilisons ici `ls-files` qui est plus une commande de coulisses qui vous montre l'état actuel de votre index.

L'index n'est pas techniquement parlant une structure arborescente - c'est en fait un manifeste aplati - mais pour nos besoins, c'est suffisamment proche.

LE RÉPERTOIRE DE TRAVAIL

Finalement, vous avez votre répertoire de travail. Les deux autres arbres stockent leur contenu de manière efficace mais peu pratique dans le répertoire `.git`. Le répertoire de travail les dépaquette comme fichiers réels, ce qui rend tout de même plus facile leur modification. Il faut penser à la copie de travail comme un **bac à sable** où vous pouvez essayer vos modifications avant de les transférer dans votre index puis le valider dans votre historique.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

1 directory, 3 files
```

Le flux de travail

L'objet principal de Git est d'enregistrer des instantanés de votre projet comme des états successifs évolutifs en manipulant ces trois arbres.

FIGURE 7-2

Visualisons ce processus : supposons que vous allez dans un nouveau répertoire contenant un fichier unique. Nous appellerons ceci **v1** du fichier et nous le marquerons en bleu. Maintenant, nous allons lancer `git init`, ce qui va créer le dépôt Git avec une référence HEAD qui pointe sur une branche à naître (`master` n'existe pas encore).

FIGURE 7-3

À ce point, seul le répertoire de travail contient quelque chose.

Maintenant, nous souhaitons valider ce fichier, donc nous utilisons `git add` qui prend le contenu du répertoire de travail et le copie dans l'index.

FIGURE 7-4

Ensuite, nous lançons `git commit`, ce qui prend le contenu de l'index et le sauve comme un instantané permanent, crée un objet commit qui pointe sur cet instantané et met à jour `master` pour pointer sur ce *commit*.

FIGURE 7-5

Si nous lançons `git status`, nous ne verrons aucune modification parce que les trois arborescences sont identiques.

Maintenant, nous voulons faire des modifications sur ce fichier et le valider. Nous suivons le même processus ; en premier nous changeons le fichier dans notre copie de travail. Appelons cette version du fichier **v2** et marquons-le en rouge.

FIGURE 7-6

Si nous lançons `git status` maintenant, nous verrons le fichier en rouge comme « Modifications qui ne seront pas validées » car cette entrée est différente entre l'index et le répertoire de travail. Ensuite, nous lançons `git add` dessus pour le monter dans notre index.

FIGURE 7-7

À ce point, si nous lançons `git status`, nous verrons le fichier en vert sous « Modifications qui seront validées » parce que l'index et HEAD diffèrent, c'est-à-dire que notre prochain *commit* proposé est différent de notre dernier *commit*. Finalement, nous lançons `git commit` pour finaliser la validation.

FIGURE 7-8

Maintenant, `git status` n'indique plus rien, car les trois arborescences sont à nouveau identiques.

Les basculements de branches ou les clonages déroulent le même processus. Quand vous extrayez une branche, cela change **HEAD** pour pointer sur la nouvelle référence de branche, popule votre **index** avec l'instantané de ce *commit*, puis copie le contenu de l'index dans votre **répertoire de travail**.

Le rôle de reset

La commande `reset` est plus compréhensible dans ce contexte.

Pour l'objectif des exemples à suivre, supposons que nous avons modifié `file.txt` à nouveau et validé une troisième fois. Donc maintenant, notre historique ressemble à ceci :

FIGURE 7-9

Détaillons maintenant ce que `reset` fait lorsque vous l'appelez. Il manipule directement les trois arborescences d'une manière simple et prédictible. Il réalise jusqu'à trois opérations basiques.

ÉTAPE 1: DÉPLACER HEAD

La première chose que `reset` va faire consiste à déplacer ce qui est pointé par HEAD. Ce n'est pas la même chose que changer HEAD lui-même (ce que fait `checkout`). `reset` déplace la branche que HEAD pointe. Ceci signifie que si HEAD est pointé sur la branche `master` (par exemple, si vous êtes sur la branche `master`), lancer `git reset 9e5e64a` va commencer par faire pointer `master` sur `9e5e64a`.

FIGURE 7-10

Quelle que soit la forme du `reset` que vous invoquez pour un *commit*, ce sera toujours le première chose qu'il tentera de faire. Avec `reset --soft`, il n'ira pas plus loin.

Maintenant, arrêtez-vous une seconde et regardez le diagramme ci-dessus pour comprendre ce qu'il s'est passé : en essence, il a défait ce que la dernière commande `git commit` a créé. Quand vous lancez `git commit`, Git crée un nouvel objet *commit* and déplace la branche pointée par HEAD dessus. Quand vous faîtes un `reset` sur HEAD~ (le parent de HEAD), vous replacez la branche où elle était, sans changer ni l'index ni la copie de travail. Vous pourriez maintenant mettre à jour l'index et relancer `git commit` pour accomplir ce que `git commit --amend` aurait fait (voir **"Modifier la dernière validation"**).

ÉTAPE 2 : MISE À JOUR DE L'INDEX (--MIXED)

Notez que si vous lancez `git status` maintenant, vous verrez en vert la différence entre l'index et la nouvelle HEAD.

La chose suivante que `reset` réalise est de mettre à jour l'index avec le contenu de l'instantané pointé par HEAD.

FIGURE 7-11

Si vous spécifiez l'option `--mixed`, `reset` s'arrêtera à cette étape. C'est aussi le comportement par défaut, donc si vous ne spécifiez aucune option (juste `git reset HEAD~` dans notre cas), c'est ici que la commande s'arrêtera.

Maintenant arrêtez-vous encore une seconde et regardez le diagramme ci-dessus pour comprendre ce qui s'est passé : il a toujours défait la dernière validation, mais il a aussi tout *désindéxé*. Vous êtes revenu à l'état précédant vos commandes `git add` et `git commit`.

ÉTAPE 3: MISE À JOUR DE LA COPIE DE TRAVAIL (--HARD)

La troisième chose que `reset` va faire est de faire correspondre la copie de travail avec l'index. Si vous utilisez l'option `--hard`, il continuera avec cette étape.

FIGURE 7-12

Donc réfléchissons à ce qui vient d'arriver. Vous avez défait la dernière validation, les commandes `git add` et `git commit` **ainsi que** tout le travail que vous avez réalisé dans le répertoire de travail.

Il est important de noter que cette option (`--hard`) est le seul moyen de rendre la commande `reset` dangereuse et est un des très rares cas où Git va réellement détruire de la donnée. Toute autre invocation de `reset` peut être défaite, mais l'option `--hard` ne le permet pas, car elle force l'écrasement des fichiers dans le répertoire de travail. Dans ce cas particulier, nous avons toujours la version **v3** du fichier dans un *commit* dans notre base de donnée Git, et nous pourrions la récupérer en parcourant notre `reflog`, mais si nous ne l'avions pas validé, Git aurait tout de même écrasé les fichiers et rien n'aurait pu être récupéré.

RÉCAPITULATIF

La commande `reset` remplace ces trois arbres dans un ordre spécifique, s'arrêtant lorsque vous lui indiquez :

1. Déplace la branche pointée par HEAD *(s'arrête ici si `--soft`)*

2. Fait ressembler l'index à HEAD *(s'arrête ici à moins que `--hard`)*
3. Fait ressembler le répertoire de travail à l'index.

Reset avec un chemin

Tout cela couvre le comportement de `reset` dans sa forme de base, mais vous pouvez aussi lui fournir un chemin sur lequel agir. Si vous spécifiez un chemin, `reset` sautera la première étape et limitera la suite de ses actions à un fichier spécifique ou à un ensemble de fichiers. Cela fait sens, en fait, HEAD n'est rien de plus qu'un pointeur et vous ne pouvez pas pointer sur une partie d'un *commit* et une partie d'un autre. Mais l'index et le répertoire de travail *peuvent* être partiellement mis à jour, donc `reset` continue avec les étapes 2 et 3.

Donc, supposons que vous lancez `git reset file.txt`. Cette forme (puisque vous n'avez pas spécifié un SHA-1 de commit ni de branche, et que vous n'avez pas non plus spécifié `--soft` ou `--hard`) est un raccourcis pour `git reset --mixed HEAD file.txt`, qui va :

1. déplacer la branche pointée par HEAD *(sauté)*
2. faire ressembler l'index à HEAD *(s'arrête ici)*

Donc, en substance, il ne fait que copier `file.txt` de HEAD vers index.

FIGURE 7-13

Ceci a l'effet pratique de *désindexer* le fichier. Si on regarde cette commande dans le diagramme et qu'on pense à ce que `git add` fait, ce sont des opposés exacts.

FIGURE 7-14

C'est pourquoi le résultat de la commande `git status` suggère que vous lanciez cette commande pour désindexer le fichier (voir **"Désindexer un fichier déjà indexé"** pour plus de détail).

Nous pourrions tout aussi bien ne pas laisser Git considérer que nous voulions dire « tirer les données depuis HEAD » en spécifiant un *commit* spécifique d'où tirer ce fichier. Nous lancerions juste quelque chose comme `git reset eb43bf file.txt`.

FIGURE 7-15

Ceci fait effectivement la même chose que si nous remettions le contenu du fichier à la **v1** dans le répertoire de travail, lancions `git add` dessus, puis le ramenions à nouveau à la **v3** (sans forcément passer par toutes ces étapes). Si nous lançons `git commit` maintenant, il enregistrera la modification qui remet le fichier à la version **v1**, même si nous ne l'avons jamais eu à nouveau dans notre répertoire de travail.

Il est intéressant de noter que comme `git add`, la commande `reset` accepte une option `--patch` pour désindexer le contenu section par section. Vous pouvez donc sélectivement désindexer ou ramener du contenu.

Écraser les *commits*

Voyons comment faire quelque chose d'intéressant avec ce tout nouveau pouvoir - écrasons des *commits*.

Supposons que vous avez une série de *commits* contenant des messages tels que « oups », « en chantier » ou « ajout d'un fichier manquant ». Vous pouvez utiliser `reset` pour les écraser tous rapidement et facilement en une seule vali-

dation qui vous donne l'air vraiment intelligent (**"Écraser un *commit*"** explique un autre moyen de faire pareil, mais dans cet exemple, c'est plus simple de faire un `reset`).

Disons que vous avez un projet où le premier *commit* contient un fichier, le second *commit* a ajouté un nouveau fichier et à modifié le premier, et le troisième à remodifié le premier fichier. Le second *commit* était encore en chantier et vous souhaitez le faire disparaître.

FIGURE 7-16

Vous pouvez lancer `git reset --soft HEAD~2` pour ramener la branche de HEAD sur l'ancien *commit* (le premier *commit* que vous souhaitez garder) :

FIGURE 7-17

Ensuite, relancez simplement `git commit` :

FIGURE 7-18

Maintenant vous pouvez voir que votre historique accessible, l'historique que vous pousseriez, ressemble à présent à un premier *commit* avec le fichier `file-a.txt` v1, puis un second qui modifie à la fois `file-a.txt` à la version 3 et ajoute `file-b.txt`. Le *commit* avec la version v2 du fichier ne fait plus partie de l'historique.

Et checkout

Finalement, vous pourriez vous demander quelle différence il y a entre `checkout` et `reset`. Comme `reset`, `checkout` manipule les trois arborescences et se

comporte généralement différemment selon que vous indiquez un chemin vers un fichier ou non.

SANS CHEMIN

Lancer `git checkout [branche]` est assez similaire à lancer `git reset --hard [branche]` en ce qu'il met à jour les trois arborescences pour qu'elles ressemblent à `[branche]`, mais avec deux différences majeures.

Premièrement, à la différence de `reset --hard`, `checkout` préserve le répertoire de travail ; il s'assure de ne pas casser des fichiers qui ont changé. En fait, il est même un peu plus intelligent que ça – il essaie de faire une fusion simple dans le répertoire de travail, de façon que tous les fichiers **non modifiés** soient mis à jour. `reset --hard`, par contre, va simplement tout remplacer unilatéralement sans rien vérifier.

La seconde différence majeure concerne sa manière de mettre à jour HEAD. Là où `reset` va déplacer la branche pointée par HEAD, `checkout` va déplacer HEAD lui-même pour qu'il pointe sur une autre branche.

Par exemple, supposons que nous avons des branches `master` et `develop` qui pointent sur des *commits* différents et que nous sommes actuellement sur `develop` (donc HEAD pointe dessus). Si nous lançons `git reset master`, `develop` lui-même pointera sur le même *commit* que `master`. Si nous lançons plutôt `git checkout master`, `develop` ne va pas bouger, seul HEAD va changer. HEAD pointera alors sur `master`.

Donc, dans les deux cas, nous déplaçons HEAD pour pointer sur le commit A, mais la manière diffère beaucoup. `reset` va déplacer la branche pointée par HEAD, alors que `checkout` va déplacer HEAD lui-même.

FIGURE 7-19

AVEC DES CHEMINS

L'autre façon de lancer `checkout` est avec un chemin de fichier, ce qui, comme `reset`, ne déplace pas HEAD. Cela correspond juste à `git reset [branche] fichier` car cela met à jour l'index avec ce fichier à ce *commit*, mais en remplaçant le fichier dans le répertoire de travail. Ce serait exactement comme `git reset --hard [branche] fichier` (si `reset` le permettait) – cela ne préserve pas le répertoire de travail et ne déplace pas non plus HEAD.

De même que `git reset` et `git add`, `checkout` accepte une option `--patch` permettant de réinitialiser sélectivement le contenu d'un fichier section par section.

Résumé

J'espère qu'à présent, vous comprenez mieux et vous sentez plus à l'aise avec la commande `reset`, même si vous pouvez vous sentir encore un peu confus sur ce qui le différencie exactement de `checkout` et avoir du mal à vous souvenir de toutes les règles de ses différentes invocations.

Voici un aide-mémoire sur ce que chaque commande affecte dans chaque arborescence. La colonne « HEAD » contient « RÉF » si cette commande déplace

la référence (branche) pointée par HEAD, et « HEAD » si elle déplace HEAD lui-même. Faites particulièrement attention à la colonne « préserve RT ? » (préserve le répertoire de travail) – si elle indique **NON**, réfléchissez à deux fois avant de lancer la commande.

	HEAD	**Index**	**Rép. Travail**	**préserve RT ?**
Niveau commit				
`reset --soft [commit]`	RÉF	NON	NON	OUI
`reset [commit]`	RÉF	OUI	NON	OUI
`reset --hard [commit]`	RÉF	OUI	OUI	**NON**
`checkout [commit]`	HEAD	OUI	OUI	OUI
Niveau Fichier				
`reset (commit) [fichier]`	NON	OUI	NON	OUI
`checkout (commit) [fichier]`	NON	OUI	OUI	**NON**

Fusion avancée

La fusion avec Git est généralement plutôt facile. Puisque Git rend facile la fusion d'une autre branche plusieurs fois, cela signifie que vous pouvez avoir une branche à très longue durée de vie que vous pouvez mettre à jour au fil de l'eau, en résolvant souvent les petits conflits plutôt que d'être surpris par un énorme conflit à la fin de la série.

Cependant, il arrive quelques fois des conflits compliqués. À la différence d'autres systèmes de contrôle de version, Git n'essaie pas d'être plus intelligent que de mesure pour la résolution des conflits. La philosophie de Git, c'est d'être malin pour déterminer lorsque la fusion est sans ambiguïté mais s'il y a un conflit, il n'essaie pas d'être malin pour le résoudre automatiquement. De ce fait, si vous attendez trop longtemps pour fusionner deux branches qui divergent rapidement, vous rencontrerez des problèmes.

Dans cette section, nous allons détailler ce que certains de ces problèmes peuvent être et quels outils Git vous offre pour vous aider à gérer ces situations délicates. Nous traiterons aussi quelques types de fusions différents, non-standard, ainsi que la manière de mémoriser les résolutions que vous avez déjà réalisées.

Conflits de fusion

Bien que nous avons couvert les bases de la résolution de conflits dans **“Conflits de fusions (*Merge conflicts*)”**, pour des conflits plus complexes, Git fournit quelques outils pour vous aider à vous y retrouver et à mieux gérer les conflits.

Premièrement, si c’est seulement possible, essayer de démarrer d’un répertoire de travail propre avant de commencer une fusion qui pourrait engendrer des conflits. Si vous avez un travail en cours, validez-le dans une branche temporaire ou remisez-le. Cela vous permettra de défaire **tout** ce que vous pourrez essayer. Si vous avez des modifications non sauvegardées dans votre répertoire de travail quand vous essayez une fusion, certaines des astuces qui vont suivre risque de vous faire perdre ce travail.

Parcourons ensemble un exemple très simple. Nous avons un fichier Ruby super simple qui affiche « hello world ».

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

Dans notre dépôt, nous créons une nouvelle branche appelée `whitespace` et nous entamons la transformation de toutes les fins de ligne Unix en fin de lignes DOS, ce qui revient à modifier chaque ligne, mais juste avec des caractères invisibles. Ensuite, nous changeons la ligne « hello world » en « hello mundo ».

```
$ git checkout -b whitespace
Basculement sur la nouvelle branche 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -w
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
```

```
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

À présent, nous rebasculons sur `master` et nous ajoutons une documentation de la fonction.

```
$ git checkout master
Basculement sur la branche 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Et maintenant, nous essayons de fusionner notre branche `whitespace` et nous allons générer des conflits dûs aux modifications de fins de ligne.

```
$ git merge whitespace
Fusion automatique de hello.rb
CONFLIT (contenu) : Conflit de fusion dans hello.rb
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

ABANDONNER UNE FUSION

Nous avons ici plusieurs options. Une première consiste à sortir de cette situation. Vous ne vous attendiez peut-être pas à rencontrer un conflit et vous ne souhaitez pas encore le gérer, alors vous pouvez simplement faire marche arrière avec `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

L'option `git merge --abort` essaie de vous ramener à l'état précédent la fusion. Les seuls cas dans lesquels il n'y parvient pas parfaitement seraient ceux pour lesquels vous aviez déjà auparavant des modifications non validées ou non remisées dans votre répertoire de travail au moment de la fusion. Sinon, tout devrait se passer sans problème.

Si, pour une raison quelconque, vous vous trouvez dans une situation horrible et que vous souhaitez repartir à zéro, vous pouvez aussi lancer `git reset --hard HEAD` ou sur toute autre référence où vous souhaitez revenir. Souvenez-vous tout de même que cela va balayer toutes les modifications de votre répertoire de travail, donc assurez-vous de n'avoir aucune modification de valeur avant.

IGNORER LES CARACTÈRE INVISIBLES

Dans ce cas spécifique, les conflits sont dûs à des espaces blancs. Nous le savons parce que le cas est simple, mais cela reste assez facile à déterminer dans les cas réels en regardant les conflits parce que chaque ligne est supprimée à une ligne puis réintroduite à la suivante. Par défaut, Git voit toutes ces lignes comme modifiées et il ne peut pas fusionner les fichiers.

La stratégie de fusion par défaut accepte quand même des arguments, et certains d'entre eux traitent le cas des modifications impliquant les caractères blancs. Si vous vous rendez compte que vous avez de nombreux conflits de caractères blancs lors d'une fusion, vous pouvez simplement abandonner la fusion et en relancer une en utilisant les options `-Xignore-all-space` ou `-Xignore-space-change`. La première option ignore **complètement** tous les

espaces tandis que la seconde traite les séquences d'un ou plusieurs espaces comme équivalentes.

```
$ git merge -Xignore-all-space whitespace
Fusion automatique de hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Puisque dans ce cas, les modifications réelles n'entraient pas en conflit, une fois les modifications d'espaces ignorées, tout fusionne parfaitement bien.

Ça sauve la vie si vous avez dans votre équipe une personne qui reformate tout d'espaces en tabulations ou vice-versa.

RE-FUSION MANUELLE D'UN FICHIER

Bien que Git gère le pré-traitement d'espaces plutôt bien, il existe d'autres types de modifications que Git ne peut pas gérer automatiquement, mais dont la fusion peut être scriptable. Par exemple, supposons que Git n'ait pas pu gérer les espaces et que nous ayons dû résoudre le problème à la main.

Ce que nous devons réellement faire est de passer le fichier que nous cherchons à fusionner à travers `dos2unix` avant d'essayer de le fusionner réellement. Comment pourrions-nous nous y prendre ?

Premièrement, nous entrons dans l'état de conflit de fusion. Puis, nous voulons obtenir des copies de la version locale (**ours**), de la version distante (**theirs**, celle qui vient de la branche à fusionner) et de la version commune (l'ancêtre commun depuis lequel les branches sont parties). Ensuite, nous voulons corriger au choix la version locale ou la distante et réessayer de fusionner juste ce fichier.

Obtenir les trois versions des fichiers est en fait assez facile. Git stocke toutes ces versions dans l'index sous formes d'étapes (**stages**), auxquelles chacune y a un nombre associé. Stage 1 est l'ancêtre commun, stage 2 est notre version, stage 3 est la version de `MERGE_HEAD`, la version qu'on cherche à fusionner (**theirs**).

Vous pouvez extraire une copie de chacune de ces versions du fichier en conflit avec la commande `git show` et une syntaxe spéciale.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Si vous voulez rentrer un peu plus dans le dur, vous pouvez aussi utiliser la commande de plomberie `ls-files -u` pour récupérer les SHA-1 des blobs Git de chacun de ces fichiers.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1       hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2       hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3       hello.rb
```

La syntaxe `:1:hello.rb` est juste un raccourcis pour la recherche du SHA-1 de ce blob.

À présent que nous avons le contenu des trois étapes dans notre répertoire de travail, nous pouvons réparer manuellement la copie distante pour résoudre le problème d'espaces et re-fusionner le fichier avec la commande méconnue `git merge-file` dont c'est l'exacte fonction.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -w
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

À ce moment, nous avons un fichier joliment fusionné. En fait, cela fonctionne même mieux que l'option `ignore-all-space` parce que le problème d'espace est corrigé avant la fusion plutôt que simplement ignoré. Dans la fusion `ignore-all-space`, nous avons en fait obtenu quelques lignes contenant des fins de lignes DOS, ce qui a mélangé les styles.

Si vous voulez vous faire une idée avant de finaliser la validation sur ce qui a réellement changé entre un côté et l'autre, vous pouvez demander à `git diff` de comparer le contenu de votre répertoire de travail que vous êtes sur le point de valider comme résultat de la fusion avec n'importe quelle étape. Détaillons chaque comparaison.

Pour comparer votre résultat avec ce que vous aviez dans votre branche avant la fusion, en d'autres termes, ce que la fusion a introduit, vous pouvez lancer `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

Donc nous voyons ici que ce qui est arrivé à notre branche, ce que nous introduisons réellement dans ce fichier avec cette fusion, n'est qu'une ligne modifiée.

Si nous voulons voir le résultat de la fusion modifiée depuis la version distante, nous pouvons lancer `git diff --theirs`. Dans cet exemple et le suivant, nous devons utiliser `-w` pour éliminer les espaces parce que nous le comparons à ce qui est dans Git et non pas notre version nettoyée `hello.theirs.rb` du fichier.

```
$ git diff --theirs -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
```

```
 def hello
   puts 'hello mundo'
 end
```

Enfin, nous pouvons voir comment le fichier a été modifié dans les deux branches avec `git diff --base`.

```
$ git diff --base -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

À ce point, nous pouvons utiliser la commande `git clean` pour éliminer les fichiers supplémentaires maintenant inutiles que nous avons créés pour notre fusion manuelle.

```
$ git clean -f
Suppression de hello.common.rb
Suppression de hello.ours.rb
Suppression de hello.theirs.rb
```

EXAMINER LES CONFLITS

Peut-être ne sommes-nous pas heureux de la résolution actuelle, ou bien l'édition à la main d'un côté ou des deux ne fonctionne pas correctement et nécessite plus de contexte.

Modifions un peu l'exemple. Pour cet exemple, nous avons deux branches à longue durée de vie qui comprennent quelques *commits* mais créent des conflits de contenu légitimes à la fusion.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

Nous avons maintenant trois *commits* uniques qui n'existent que sur la branche `master` et trois autres sur la branche `mundo`. Si nous essayons de fusionner la branche `mundo`, nous obtenons un conflit.

```
$ git merge mundo
Fusion automatique de hello.rb
CONFLIT (contenu): Conflit de fusion dans hello.rb
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Nous souhaitons voir ce qui constitue le conflit de fusion. Si nous ouvrons le fichier, nous verrons quelque chose comme :

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Les deux côtés de la fusion on ajouté du contenu au fichier, mais certains *commits* ont modifié le fichier au même endroit, ce qui a causé le conflit.

Explorons quelques outils que vous avez à disposition pour déterminer comment ce conflit est apparu. Peut-être le moyen de résoudre n'est-il pas évident. Il nécessite plus de contexte.

Un outil utile est `git checkout` avec l'option `--conflict`. Il va re-extraire le fichier et remplacer les marqueurs de conflit. Cela peut être utile si vous souhaitez éliminer les marqueurs et essayer de résoudre le conflit à nouveau.

Vous pouvez passer en paramètre à `--conflict`, soit `diff3` soit `merge` (le paramètre par défaut). Si vous lui passez `diff3`, Git utilisera une version différente des marqueurs de conflit, vous fournissant non seulement les versions locales (*ours*) et distantes (*theirs*), mais aussi la version « base » intégrée pour vous fournir plus de contexte.

```
$ git checkout --conflict=diff3 hello.rb
```

Une fois que nous l'avons lancé, le fichier ressemble à ceci :

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

Si vous appréciez ce format, vous pouvez le régler comme défaut pour les futur conflits de fusion en renseignant le paramètre `merge.conflictstyle` avec `diff3`.

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

La commande `git checkout` peut aussi accepter les options `--ours` et `--theirs`, qui peuvent servir de moyen rapide de choisir unilatéralement une version ou une autre sans fusion.

Cela peut être particulièrement utile pour les conflits de fichiers binaires où vous ne pouvez que choisir un des côté, ou des conflits où vous souhaitez fusionner certains fichiers depuis d'autres branches - vous pouvez fusionner, puis extraire certains fichiers depuis un côté ou un autre avant de valider le résultat.

JOURNAL DE FUSION

Un autre outil utile pour la résolution de conflits de fusion est `git log`. Cela peut vous aider à obtenir du contexte ce qui a contribué aux conflits. Parcourir

un petit morceau de l'historique pour se rappeler pourquoi deux lignes de développement ont touché au même endroit dans le code peut s'avérer quelque fois très utile.

Pour obtenir une liste complète de tous les *commits* uniques qui ont été introduits dans chaque branche impliquée dans la fusion, nous pouvons utiliser la syntaxe « triple point » que nous avons apprise dans **"Triple point"**.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

Voilà une belle liste des six *commits* impliqués, ainsi que chaque ligne de développement sur laquelle chaque *commit* se trouvait.

Néanmoins, nous pouvons simplifier encore plus ceci pour fournir beaucoup plus de contexte. Si nous ajoutons l'option `--merge` à `git log`, il n'affichera que les *commits* de part et d'autre de la fusion qui modifient un fichier présentant un conflit.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

Si nous lançons cela avec l'option `-p` à la place, vous obtenez les diffs limités au fichier qui s'est retrouvé en conflit. Cela peut s'avérer **vraiment** utile pour vous donner le contexte nécessaire à la compréhension de la raison d'un conflit et à sa résolution intelligente.

FORMAT DE DIFF COMBINÉ

Puisque Git indexe tous les résultats de fusion couronnés de succès, quand vous lancez `git diff` dans un état de conflit de fusion, vous n'obtenez que ce qui toujours en conflit à ce moment. Il peut s'avérer utile de voir ce qui reste à résoudre.

Quand vous lancez `git diff` directement après le conflit de fusion, il vous donne de l'information dans un format de diff plutôt spécial.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

Ce format s'appelle « diff combiné » (*combined diff*) et vous fournit deux colonnes d'information sur chaque ligne. La première colonne indique que la ligne est différente (ajoutée ou supprimée) entre la branche « *ours* » et le fichier dans le répertoire de travail. La seconde colonne fait de même pour la branche « *theirs* » et la copie du répertoire de travail.

Donc dans cet exemple, vous pouvez voir que les lignes `<<<<<<<` et `>>>>>>>` sont dans la copie de travail mais n'étaient dans aucun des deux côtés de la fusion. C'est logique parce que l'outil de fusion les a collés ici pour donner du contexte, mais nous devrons les retirer.

Si nous résolvons le conflit et relançons `git diff`, nous verrons la même chose, mais ce sera un peu plus utile.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```

```
  hello()
```

Ceci nous montre que « hola world » était présent de notre côté mais pas dans la copie de travail, que « hello mundo » était présent de l'autre côté mais pas non plus dans la copie de travail et que finalement, « hola mundo » n'était dans aucun des deux côtés, mais se trouve dans la copie de travail. C'est particulièrement utile lors d'une revue avant de valider la résolution.

Vous pouvez aussi l'obtenir depuis `git log` pour toute fusion pour visualiser comment quelque chose a été résolu après coup. Git affichera ce format si vous lancez `git show` sur un *commit* de fusion, ou si vous ajoutez une option `--cc` à `git log -p` (qui par défaut ne montre que les patchs des *commits* qui ne sont pas des fusions).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Défaire des fusions

Comme vous savez créer des *commits* de fusion à présent, vous allez certainement en faire par erreur. Un des grands avantages de l'utilisation de Git est qu'il

n'est pas interdit de faire des erreurs, parce qu'il reste toujours possible (et très souvent facile) de les corriger.

Les *commits* de fusion ne font pas exception. Supposons que vous avez commencé à travailler sur une branche thématique, que vous l'avez accidentellement fusionnée dans `master` et qu'en conséquence votre historique ressemble à ceci :

FIGURE 7-20

Commit de fusion accidentel

Il existe deux façons d'aborder ce problème, en fonction du résultat que vous souhaitez obtenir.

CORRECTION DES RÉFÉRENCES

Si le *commit* de fusion non désiré n'existe que dans votre dépôt local, la solution la plus simple et la meilleure consiste à déplacer les branches pour qu'elles pointent où on le souhaite. La plupart du temps, en faisant suivre le `git merge` malencontreux par un `git reset --hard HEAD~`, on remet les pointeurs de branche dans l'état suivant :

FIGURE 7-21

Historique après `git reset --hard HEAD~`

Nous avons détaillé `reset` dans **“Reset démystifié”** et il ne devrait pas être très difficile de comprendre ce résultat. Voici néanmoins un petit rappel : `reset --hard` réalise généralement trois étapes :

1. Déplace la branche pointée par HEAD ; dans notre cas, nous voulons déplacer `master` sur son point avant la fusion (C6),
2. Faire ressembler l’index à HEAD,
3. Faire ressembler le répertoire de travail à l’index.

Le défaut de cette approche est qu’elle ré-écrit l’historique, ce qui peut être problématique avec un dépôt partagé. Reportez-vous à **“Les dangers du rebasage”** pour plus d’information ; en résumé si d’autres personnes ont déjà les *commits* que vous ré-écrivez, il vaudrait mieux éviter un `reset`. Cette approche ne fonctionnera pas non plus si d’autres *commits* ont été créés depuis la fusion ; déplacer les références des branches éliminera effectivement ces modifications.

INVERSER LE *COMMIT*

Si les déplacements des pointeurs de branche ne sont pas envisageables, Git vous donne encore l’option de créer un nouveau *commit* qui défait toutes les modifications d’un autre déjà existant. Git appelle cette option une « inversion » (*revert*), et dans ce scénario particulier, vous l’invoqueriez comme ceci :

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

L'option `-m 1` indique quel parent est le principal et devrait être conservé. Si vous invoquez une fusion dans HEAD (`git merge topic`), le nouveau *commit* a deux parents : le premier est HEAD (C6), et le second est le sommet de la branche en cours de fusion (C4). Dans ce cas, nous souhaitons défaire toutes les modifications introduites dans le parent numéro 2 (C4), tout en conservant tout le contenu du parent numéro 1 (C6).

L'historique avec le *commit* d'inversion ressemble à ceci :

FIGURE 7-22

Historique après `git revert -m 1`

Le nouveau *commit* `^M` a exactement le même contenu que C6, et partant de là, c'est comme si la fusion n'avait pas eu lieu, mis à part que les *commits* qui ne sont plus fusionnés sont toujours dans l'historique de HEAD. Git sera confus si vous tentez de re-fusionner `topic` dans `master` :

```
$ git merge topic
Already up-to-date.
```

Il n'y a rien dans `topic` qui ne soit pas déjà joignable depuis `master`. Pire encore, si vous ajoutez du travail à `topic` et re-fusionnez, Git n'ajoutera que les modifications *depuis* la fusion inversée :

FIGURE 7-23

Historique avec une mauvaise fusion

Le meilleur contournement de ceci est de dé-inverser la fusion originale, puisque vous voulez ajouter les modifications qui ont été annulées, **puis** de créer un nouveau *commit* de fusion :

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

FIGURE 7-24

Historique après re-fusion de la fusion annulée

Dans cet exemple, `M` et `^M` s'annulent. `^^M` fusionne effectivement les modifications depuis `C3` et `C4`, et `C8` fusionne les modifications depuis `C7`, donc à présent, `topic` est totalement fusionnée.

Autres types de fusions

Jusqu'ici, nous avons traité les fusions normales entre deux branches qui ont été gérées normalement avec ce qui s'appelle la stratégie « récursive » de fusion. Il existe cependant d'autres manières de fusionner des branches. Traitons en quelques unes rapidement.

PRÉFÉRENCE *OUR* OU *THEIRS*

Premièrement, il existe un autre mode utile que nous pouvons utiliser avec le mode « recursive » normal de fusion. Nous avons déjà vu les options `ignore-all-space` et `ignore-space-change` qui sont passées avec `-X` mais nous pouvons aussi indiquer à Git de favoriser un côté plutôt que l'autre lorsqu'il rencontre un conflit.

Par défaut, quand Git rencontre un conflit entre deux branches en cours de fusion, il va ajouter des marqueurs de conflit de fusion dans le code et marquer le fichier en conflit pour vous laisser le résoudre. Si vous préférez que Git choisisse simplement un côté spécifique et qu'il ignore l'autre côté au lieu de vous laisser fusionner manuellement le conflit, vous pouvez passer `-Xours` ou `-Xtheirs` à la commande `merge`.

Si une des options est spécifiée, Git ne va pas ajouter de marqueurs de conflit. Toutes les différences qui peuvent être fusionnées seront fusionnées. Pour toutes les différences qui génèrent un conflit, Git choisira simplement la version du côté que vous avez spécifié, y compris pour les fichiers binaires.

Si nous retournons à l'exemple « hello world » précédent, nous pouvons voir que la fusion provoque des conflits.

```
$ git merge mundo
Fusion automatique de  hello.rb
CONFLIT (contenu): Conflit de fusion dans hello.rb
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Cependant, si nous la lançons avec `-Xours` ou `-Xtheirs`, elle n'en provoque pas.

```
$ git merge -Xours mundo
Fusion automatique de hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

Dans ce dernier cas, au lieu d'obtenir des marqueurs de conflit dans le fichier avec « hello mundo » d'un côté et « hola world » de l'autre, Git choisira simplement « hola world ». À part cela, toutes les autres modifications qui ne génèrent pas de conflit sont fusionnées sans problème.

Cette option peut aussi être passée à la commande `git merge-file` que nous avons utilisée plus tôt en lançant quelque chose comme `git merge-file --ours` pour les fusions de fichiers individuels.

Si vous voulez faire quelque chose similaire mais indiquer à Git de ne même pas essayer de fusionner les modifications de l'autre côté, il existe une option draconienne qui s'appelle la *stratégie* de fusion « *ours* ».

Cela réalisera une fusion factice. Cela enregistrera un nouveau *commit* de fusion avec les deux branches comme parents, mais ne regardera même pas la branche en cours de fusion. Cela enregistrera simplement le code exact de la branche courante comme résultat de la fusion.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Vous pouvez voir qu'il n'y a pas de différence entre la branche sur laquelle nous étions précédemment et le résultat de la fusion.

Cela peut s'avérer utile pour faire croire à Git qu'une branche est déjà fusionnée quand on fusionne plus tard. Par exemple, disons que vous avez créé une branche depuis une branche « release » et avez travaillé dessus et que vous allez vouloir réintégrer ce travail dans `master`. Dans l'intervalle, les correctifs de `master` doivent être reportés dans la branche `release`. Vous pouvez fusionner la branche de correctif dans la branche `release` et aussi faire un `merge -s ours` de cette branche dans la branche `master` (même si le correctif est déjà présent) de sorte que lorsque fusionnerez plus tard la branche `release` , il n'y aura pas de conflit dû au correctif.

SUBTREE MERGING

L'idée de la fusion de sous-arbre est que vous avez deux projets, et l'un des projets se réfère à un sous-dossier de l'autre et vice-versa. Quand vous spécifiez une fusion de sous-arbre, Git est souvent assez malin pour se rendre compte que l'un est un sous-arbre de l'autre et fusionner comme il faut.

Nous allons explorer à travers un exemple comment ajouter un projet séparé à l'intérieur d'un projet existant et ensuite fusionner le code du second dans un sous-dossier du premier.

D'abord, nous ajouterons l'application Rack à notre projet. Nous ajouterons le projet Rack en tant que référence distante dans notre propre projet puis l'extrairons dans sa propre branche :

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Maintenant nous avons la racine du projet Rack dans notre branche `rack_branch` et notre propre projet dans la branche `master`. Si vous extrayez un projet puis l'autre, vous verrez qu'ils ont des racines de projet différentes :

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

C'est un concept assez étrange. Toutes les branches de votre dépôt n'ont pas vraiment besoin d'être des branches du même projet. C'est inhabituel, parce que c'est rarement utile, mais c'est assez facile d'avoir des branches qui contiennent des historiques totalement différents.

Dans notre cas, nous voulons tirer le projet Rack dans notre projet `master` en tant que sous-dossier. Nous pouvons faire cela dans Git avec la commande `git read-tree`. Vous en apprendrez plus sur `read-tree` et ses amis dans **Chapter 10**, mais pour l'instant sachez qu'elle lit l'arborescence d'une branche dans votre index courant et dans le répertoire de travail. Nous venons de rebasculer dans notre branche `master`, et nous tirons la branche `rack` dans le sous-dossier `rack` de notre branche `master` de notre projet principal :

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Quand nous validons, c'est comme si nous avions tous les fichiers Rack dans ce sous-dossier – comme si les avions copiés depuis une archive. Ce qui est intéressant est que nous pouvons assez facilement fusionner les modifications d'une des branches dans l'autre. Donc, si le projet Rack est mis à jour, nous pouvons tirer en amont les modifications en basculant sur cette branche et en tirant :

```
$ git checkout rack_branch
$ git pull
```

Ensuite, nous pouvons fusionner les modifications dans notre brancher `master`. Nous pouvons utiliser `git merge -s subtree` et cela marchera bien, mais Git fusionnera lui aussi les historiques ensemble, ce que nous ne voudrons probablement pas. Pour tirer les modifications et préremplir le message de validation, utilisez les options `--squash` et `--no-commit` en complément de l'option de stratégie `-s subtree` :

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Toutes les modifications du projet Rach sont fusionnées et prêtes à être validées localement. Vous pouvez aussi faire l'inverse – faire les modifications dans le sous-dossier `rack` de votre branche `master` et ensuite les fusionner plus tard dans votre branche `rack_branch` pour les soumettre aux mainteneurs ou les pousser en amont.

Ceci nous donne un moyen d'avoir un flux de travail quelque peu similaire au flux de travail des sous-modules sans utiliser les sous-modules (que nous couvrirons dans **"Sous-modules"**). Npus pouvons garder dans notre dépôt des branches avec d'autres projets liés et les fusionner façon sous-arbre dans notre projet occasionnellement. C'est bien par certains côtés ; par exemple tout le code est validé à un seul endroit. Cependant, cela a d'autres défauts comme le fait que c'est un petit peu plus complexe et c'est plus facile de faire des erreurs en réintégrant les modifications ou en poussant accidentellement une branche dans un dépôt qui n'a rien à voir.

Une autre chose un peu étrange est que pour obtenir la différence entre ce que vous avez dans votre sous-dossier `rack` et le code dans votre branche `rack_branch` – pour voir si vous avez besoin de les fusionner – vous ne pouvez

pas utiliser la commande `diff` classique. À la place, vous devez lancer `git diff-tree` avec la branche que vous voulez comparer :

```
$ git diff-tree -p rack_branch
```

Ou, pour comparer ce qui est dans votre sous-dossier `rack` avec ce qu'était la branche `master` sur le serveur la dernière fois que vous avez tiré, vous pouvez lancer

```
$ git diff-tree -p rack_remote/master
```

Rerere

La fonctionalité `git rerere` est une fonction un peu cachée. Le nom vient de l'anglais *reuse recorded resolution* (« *ré* utiliser les *ré* solutions en *re* gistrées ») et comme son nom l'indique, cela permet de demander à Git de se souvenir comment vous avez résolu un conflit sur une section de diff de manière que la prochaine fois qu'il rencontre le même conflit, il le résolve automatiquement pour vous.

Il existe pas mal de scénarios pour lesquels cette fonctionalité peut se montrer efficace. Un exemple mentionné dans la documentation cite le cas d'une branche au long cours qui finira par fusionner proprement mais ne souhaite pas montrer des fusions intermédiaires. Avec `rerere` activé, vous pouvez fusionner de temps en temps, résoudre les conflits, puis sauvegarder la fusion. Si vous faites ceci en continu, alors la dernière fusion devrait être assez facile parce que `rerere` peut quasiment tout faire automatiquement pour vous.

La même tactique peut être utilisée si vous souhaitez rebaser plusieurs fois une branche tout en ne souhaitant pas avoir à gérer les mêmes conflits de rebasage à chaque fois. Ou si vous voulez prendre la branche que vous avez fusionnée et si vous avez eu à corriger des conflits, puis décidez de la rebaser pour finir - vous souhaitez sûrement ne pas avoir à recorriger les mêmes conflits.

Une autre situation similaire apparaît quand vous fusionnez ensemble de temps en temps une série de branches thématiques évolutives dans un sommet testable, comme le projet Git lui-même le fait souvent. Si les tests échouent, vous pouvez rembobiner vos fusions et les rejouer en écartant la branche qui a provoqué l'erreur sans devoir résoudre à nouveau tous les conflits.

Pour activer la fonctionnalité `rerere`, vous devez simplement lancer le paramétrage :

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

Vous pouvez aussi l'activer en créant le répertoire `.git/rr-cach` dans un dépôt spécifique, mais l'activation par ligne de commande reste plus claire et permet d'activer la fonction globalement.

Voyons maintenant un exemple similaire au précédent. Supposons que nous avons un fichier qui contient ceci :

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

Dans une branche, nous changeons « hello » en « hola », puis dans une autre branche nous changeons « world » en « mundo », comme précédemment.

FIGURE 7-25

Quand nous fusionnons les deux branches ensemble, nous obtenons un conflit de fusion :

```
$ git merge i18n-world
Fusion automatique de hello.rb
CONFLIT (contenu): Conflit de fusion dans hello.rb
```

```
Recorded preimage for 'hello.rb'
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Vous devriez avoir noté la présence d'un nouvelle ligne `Recorded preimage for FILE` (« Enregistrement de la pré-image pour FICHIER »). À part ce détail, cela ressemble à un conflit de fusion tout à fait normal. À ce stade, `rerere` peut déjà nous dire un certain nombre de choses. Normalement, vous lanceriez un `git status` pour voir l'état actuel des conflits.

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#       both modified:      hello.rb
#
```

Cependant, `git rerere` vous indiquera aussi les conflits pour lesquels il a enregistré la pré-image grâce à `git rerere status` :

```
$ git rerere status
hello.rb
```

Et `git rerere diff` montrera l'état actuel de la résolution - quel était le conflit de départ et comment vous l'avez résolu.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
```

```
+>>>>>>> i18n-world
 end
```

En complément (et bien que ça n'ait pas vraiment à voir avec `rerere`), vous pouvez utiliser `ls-files -u` pour voir les fichiers en conflit ainsi que les versions précédentes, à droite et à gauche :

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1       hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2       hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3       hello.rb
```

Maintenant, vous pouvez le résoudre pour que la ligne de code soit simplement `puts 'hola mundo'` et vous pouvez relancer la commande `rerere diff` pour visualiser ce que rerere va mémoriser :

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

Cela indique simplement que quand Git voit un conflit de section dans un fichier `hello.rb` qui contient « hello mundo » d'un côté et « hola world » de l'autre, il doit résoudre ce conflit en « hola mundo ».

Maintenant, nous pouvons le marquer comme résolu et le valider :

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Vous pouvez voir qu'il a « enregistré la résolution pour FICHER » (*Recorded resolution for FILE*).

FIGURE 7-26

Maintenant, défaisons la fusion et rebasons plutôt la branche sur la branche master. Nous pouvons déplacer notre branche en arrière en utilisant `reset` comme vu dans **"Reset démystifié"**.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Notre fusion est défaite. Rebasons notre branche thématique.

```
$ git checkout i18n-world
Basculement sur la branche 'i18n-world'

$ git rebase master
Premièrement, rembobinons head pour rejouer votre travail par-dessus...
Application : i18n world
Utilisation de l'information de l'index pour reconstruire un arbre de base...
M       hello.rb
Retour à un patch de la base et fusion à 3 points...
Fusion automatique de hello.rb
CONFLIT (contenu) : Conflit de fusion dans hello.rb
Resolved 'hello.rb' using previous resolution.
```

```
Échec d'intégration des modifications.
Le patch a échoué à 0001 i18n world
```

Ici, nous avons obtenu le conflit de fusion auquel nous nous attendions, mais des lignes supplémentaires sont apparues, en particulier `Resolved FILE using previous resolution` (FICHIER résolu en utilisant une résolution précédente). Si nous inspectons le fichier `hello.rb`, il ne contient pas de marqueur de conflit.

```
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

`git diff` nous montrera comment le conflit a été re-résolu automatiquement :

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```

FIGURE 7-27

Vous pouvez aussi recréer l'état de conflit du fichier avec la commande `checkout` :

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

Nous avons vu un exemple de ceci dans **"Fusion avancée"**. Pour le moment, re-résolvons-le en relançant `rerere` :

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Nous avons re-résolu le conflit du fichier automatiquement en utilisant la résolution mémorisée par `rerere`. Vous pouvez le valider avec `add` et terminer de rebaser.

```
$ git add hello.rb
$ git rebase --continue
Application: i18n one word
```

Dans les cas où vous souhaitez réaliser de nombreuses fusions successives d'une branche thématique ou si vous souhaitez la synchroniser souvent avec `master` sans devoir gérer des tas de conflits de fusion, ou encore si vous rebasez souvent, vous pouvez activer `rerere` qui vous simplifiera la vie.

Déboguer avec Git

Git fournit aussi quelques outils pour vous aider à déboguer votre projet. Puisque Git est conçu pour fonctionner avec pratiquement tout type de projet, ces outils sont plutôt génériques, mais ils peuvent souvent vous aider à traquer un bogue ou au moins cerner où cela tourne mal.

Fichier annoté

Si vous traquez un bogue dans votre code et que vous voulez savoir quand il est apparu et pourquoi, annoter les fichiers est souvent le meilleur moyen. Cela vous montre la dernière validation qui a modifié chaque ligne de votre fichier. Donc, si vous voyez une méthode dans votre code qui est boguée, vous pouvez visualiser le fichier annoté avec `git blame` pour voir quand chaque ligne de la méthode a été modifiée pour la dernière fois et par qui. Cet exemple utilise l'option `-L` pour limiter la sortie des lignes 12 à 22 :

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
```

```
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

Remarquez que le premier champ est le SHA-1 partiel du dernier *commit* à avoir modifié la ligne. Les deux champs suivants sont des valeurs extraites du *commit* : l'auteur et la date du *commit*, vous pouvez donc facilement voir qui a modifié la ligne et quand. Ensuite arrive le numéro de ligne et son contenu. Remarquez également les lignes dont le *commit* est `^4832fe2`, elles désignent les lignes qui étaient dans la version du fichier lors du premier *commit* de ce fichier. Ce *commit* contient le premier ajout de ce fichier, et ces lignes n'ont pas été modifiées depuis. Tout ça est un peu confus, parce que vous connaissez maintenant au moins trois façons différentes que Git interprète `^` pour modifier l'empreinte SHA, mais au moins, vous savez ce qu'il signifie ici.

Une autre chose sympa sur Git, c'est qu'il ne suit pas explicitement les renommages de fichier. Il enregistre les contenus puis essaye de deviner ce qui a été renommé implicitement, après coup. Ce qui nous permet d'utiliser cette fonctionnalité intéressante pour suivre toutes sortes de mouvements de code. Si vous passez `-C` à `git blame`, Git analyse le fichier que vous voulez annoter et essaye de deviner d'où les bouts de code proviennent par copie ou déplacement. Récemment, j'ai remanié un fichier nommé `GITServerHandler.m` en le divisant en plusieurs fichiers, dont le fichier `GITPackUpload.m`. En annotant `GITPackUpload.m` avec l'option `-C`, je peux voir quelles sections de code en sont originaires :

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

C'est vraiment utile, non ? Normalement, vous obtenez comme *commit* originel celui dont votre code a été copié, puisque ce fut la première fois que vous avez touché à ces lignes dans ce fichier. Git vous montre le *commit* d'origine, celui où vous avez écrit ces lignes, même si c'était dans un autre fichier.

Recherche dichotomique

Annoter un fichier peut aider si vous savez déjà où le problème se situe. Si vous ne savez pas ce qui a cassé le code, il peut y avoir des dizaines, voire des centaines de *commits* depuis le dernier état où votre code fonctionnait et vous aimeriez certainement exécuter `git bisect` pour vous aider. La commande `bisect` effectue une recherche par dichotomie dans votre historique pour vous aider à identifier aussi vite que possible quel *commit* a vu le bogue naître.

Disons que vous venez juste de pousser une version finale de votre code en production, vous récupérez un rapport de bogue à propos de quelque chose qui n'arrivait pas dans votre environnement de développement, et vous n'arrivez pas à trouver pourquoi votre code le fait. Vous retournez sur votre code et il apparaît que vous pouvez reproduire le bogue mais vous ne savez pas ce qui se passe mal. Vous pouvez faire une recherche par dichotomie pour trouver ce qui ne va pas. D'abord, exécutez `git bisect start` pour démarrer la procédure, puis utilisez la commande `git bisect bad` pour dire que le *commit* courant est bogué. Ensuite, dites à `bisect` quand le code fonctionnait, en utilisant `git bisect good [bonne_version]` :

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git trouve qu'il y a environ 12 *commits* entre celui que vous avez marqué comme le dernier bon connu (v1.0) et la version courante qui n'est pas bonne, et il a récupéré le *commit* du milieu à votre place. À ce moment, vous pouvez dérouler vos tests pour voir si le bogue existait dans ce *commit*. Si c'est le cas, il a été introduit quelque part avant ce *commit* médian, sinon, il l'a été évidemment après. Il apparait que le bogue ne se reproduit pas ici, vous le dites à Git en tapant `git bisect good` et continuez votre périple :

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Vous êtes maintenant sur un autre *commit*, à mi-chemin entre celui que vous venez de tester et votre *commit* bogué. Vous exécutez une nouvelle fois votre test et trouvez que ce *commit* est bogué, vous le dites à Git avec `git bisect bad` :

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Ce *commit*-ci est bon, et Git a maintenant toutes les informations dont il a besoin pour déterminer où le bogue a été créé. Il vous affiche le SHA-1 du premier *commit* bogué, quelques informations du *commit* et quels fichiers ont été modifiés dans celui-ci, vous pouvez donc trouver ce qui s'est passé pour créer ce bogue :

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Lorsque vous avez fini, vous devez exécuter `git bisect reset` pour réinitialiser votre HEAD où vous étiez avant de commencer, ou vous travaillerez dans un répertoire de travail non clairement défini :

```
$ git bisect reset
```

C'est un outil puissant qui vous aidera à vérifier des centaines de *commits* en quelques minutes. En plus, si vous avez un script qui sort avec une valeur 0 s'il est bon et autre chose sinon, vous pouvez même automatiser `git bisect`. Premièrement vous lui spécifiez l'intervalle en lui fournissant les bon et mauvais *commits* connus. Vous pouvez faire cela en une ligne en les entrant à la suite de la commande `bisect start`, le mauvais *commit* d'abord :

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Cela exécute automatiquement `test-error.sh` sur chaque *commit* jusqu'à ce que Git trouve le premier *commit* bogué. Vous pouvez également exécuter

des commandes comme `make` ou `make tests` ou quoi que ce soit qui exécute des tests automatisés à votre place.

Sous-modules

Il arrive souvent lorsque vous travaillez sur un projet que vous deviez utiliser un autre projet comme dépendance. Cela peut être une bibliothèque qui est développée par une autre équipe ou que vous développez séparément pour l'utiliser dans plusieurs projets parents. Ce scénario provoque un problème habituel : vous voulez être capable de gérer deux projets séparés tout en utilisant l'un dans l'autre.

Voici un exemple. Supposons que vous développez un site web et que vous créez des flux Atom. Plutôt que d'écrire votre propre code de génération Atom, vous décidez d'utiliser une bibliothèque. Vous allez vraisemblablement devoir soit inclure ce code depuis un gestionnaire partagé comme CPAN ou Ruby gem, soit copier le code source dans votre propre arborescence de projet. Le problème d'inclure la bibliothèque en tant que bibliothèque externe est qu'il est difficile de la personnaliser de quelque manière que ce soit et encore plus de la déployer, car vous devez vous assurer de la disponibilité de la bibliothèque chez chaque client. Mais le problème d'inclure le code dans votre propre projet est que n'importe quelle personnalisation que vous faites est difficile à fusionner lorsque les modifications du développement principal arrivent.

Git gère ce problème avec les sous-modules. Les sous-modules vous permettent de gérer un dépôt Git comme un sous-répertoire d'un autre dépôt Git. Cela vous laisse la possibilité de cloner un dépôt dans votre projet et de garder isolés les **commits** de ce dépôt.

Démarrer un sous-module

Détaillons le développement d'un projet simple qui a été divisé en un projet principal et quelques sous-projets.

Commençons par ajouter le dépôt d'un projet Git existant comme sous-module d'un dépôt sur lequel nous travaillons. Pour ajouter un nouveau sous-module, nous utilisons la commande `git submodule add` avec l'URL du projet que nous souhaitons suivre. Dans cette exemple, nous ajoutons une bibliothèque nommée « DbConnector ».

```
$ git submodule add https://github.com/chaconinc/DbConnector
Clonage dans 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
```

```
remote: Total 11 (delta 0), reused 11 (delta 0)
Dépaquetage des objets: 100% (11/11), fait.
Vérification de la connectivité... fait.
```

Par défaut, les sous-modules ajoutent le sous-projet dans un répertoire portant le même nom que le dépôt, dans notre cas « DbConnector ». Vous pouvez ajouter un chemin différent à la fin de la commande si vous souhaitez le placer ailleurs.

Si vous lancez `git status` à ce moment, vous noterez quelques différences.

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Modifications qui seront validées :
  (utilisez "git reset <fichier>..." pour désindexer)

        nouveau fichier :   .gitmodules
        nouveau fichier :   DbConnector
```

Premièrement, un fichier `.gitmodules` vient d'apparaître. C'est le fichier de configuration qui stocke la liaison entre l'URL du projet et le sous-répertoire local dans lequel vous l'avez tiré.

```
$ cat .gitmodules
[submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
```

Si vous avez plusieurs sous-modules, vous aurez plusieurs entrées dans ce fichier. Il est important de noter que ce fichier est en gestion de version comme vos autres fichiers, à l'instar de votre fichier `.gitignore`. Il est poussé et tiré comme le reste de votre projet. C'est également le moyen que les autres personnes qui clonent votre projet ont de savoir où récupérer le projet du sous-module.

> Comme l'URL dans le fichier `.gitmodules` est ce que les autres personnes essaieront en premier de cloner et de tirer, assurez-vous que cette URL est effectivement accessible par les personnes concernées. Par exemple, si vous utilisez une URL différente pour pousser que celle que les autres utiliseront pour tirer, utilisez l'URL auquelle les autres ont accès. Vous pouvez surcharger cette URL localement pour votre usage propre avec la commande `git config submodule.DbConnector.url PRIVATE_URL`.

L'autre information dans la sortie de `git status` est l'entrée du répertoire du projet. Si vous exécutez `git diff`, vous verrez quelque chose d'intéressant :

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Même si `DbConnector` est un sous-répertoire de votre répertoire de travail, Git le voit comme un sous-module et ne suit pas son contenu (si vous n'êtes pas dans ce répertoire). En échange, Git l'enregistre comme un **commit** particulier de ce dépôt.

Si vous souhaitez une sortie diff plus agréable, vous pouvez passer l'option `--submodule` à `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

Au moment de valider, vous voyez quelque chose comme :

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Remarquez le mode 160000 pour l'entrée `DbConnector`. C'est un mode spécial de Git qui signifie globalement que vous êtes en train d'enregistrer un **commit** comme un répertoire plutôt qu'un sous-répertoire ou un fichier.

Cloner un projet avec des sous-modules

Maintenant, vous allez apprendre à cloner un projet contenant des sous-modules. Quand vous récupérez un tel projet, vous obtenez les différents répertoires qui contiennent les sous-modules, mais encore aucun des fichiers :

```
$ git clone https://github.com/chaconinc/MainProject
Clonage dans 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Dépaquetage des objets: 100% (14/14), fait.
Vérification de la connectivité... fait.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

Le répertoire `DbConnector` est présent mais vide. Vous devez exécuter deux commandes : `git submodule init` pour initialiser votre fichier local de configuration, et `git submodule update` pour tirer toutes les données de ce projet et récupérer le **commit** approprié tel que listé dans votre super-projet :

```
$ git submodule init
Sous-module 'DbConnector' (https://github.com/chaconinc/DbConnector) enregistré pour le chem
$ git submodule update
Clonage dans 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Votre répertoire `DbConnector` est maintenant dans l'état exact dans lequel il était la dernière fois que vous avez validé.

Il existe une autre manière plus simple d'arriver au même résultat. Si vous passez l'option `--recursive` à la commande `git clone`, celle-ci initialisera et mettra à jour automatiquement chaque sous-module du dépôt.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Clonage dans 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Dépaquetage des objets: 100% (14/14), fait.
Vérification de la connectivité... fait.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbCo
Clonage dans 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Dépaquetage des objets: 100% (11/11), fait.
Vérification de la connectivité... fait.
chemin du sous-module 'DbConnector' : 'c3f01dc8862123d317dd46284b05b6892c7b29bc' extrait
```

Travailler sur un projet comprenant des sous-modules

Nous avons à présent une copie d'un projet comprenant des sous-modules, et nous allons collaborer à la fois sur le projet principal et sur le projet du sous-module.

TIRER DES MODIFICATIONS AMONT

Le modèle le plus simple d'utilisation des sous-modules est le cas de la simple consommation d'un sous-projet duquel on souhaite obtenir les mises à jour de

temps en temps mais auquel on n'apporte pas de modification dans la copie de travail. Examinons un exemple simple.

Quand vous souhaitez vérifier si le sous-module a évolué, vous pouvez vous rendre dans le répertoire correspondant et lancer `git fetch` et puis `git merge` de la branche amont pour mettre à jour votre code local.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Mise à jour c3f01dc..d0354fc
Avance rapide
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Si vous revenez maintenant dans le projet principal et lancez `git diff --submodule`, vous pouvez remarquer que le sous-module a été mis à jour et vous pouvez obtenir une liste des *commits* qui y ont été ajoutés. Si vous ne voulez pas taper `--submodule` à chaque fois que vous lancez `git diff`, vous pouvez le régler comme format par défaut en positionnant le paramètre de configuration `diff.submodule` à la valeur « `log` ».

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Si vous validez à ce moment, vous fixez la version du sous-module à la version actuelle quand d'autres personnes mettront à jour votre projet.

Il existe aussi un moyen plus facile, si vous préférez ne pas avoir à récupérer et fusionner manuellement les modifications dans le sous-répertoire. Si vous lancez la commande `git submodule update --remote`, Git se rendra dans vos sous-modules et réalisera automatiquement le `fetch` et le `merge`.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
```

```
   3f19983..d0354fc  master     -> origin/master
chemin du sous-module 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Cette commande considère par défaut que vous souhaitez mettre à jour la copie locale vers la branche `master` du dépôt du sous-module. Vous pouvez, cependant, indiquer une autre branche. Par exemple, si le sous-module DbConnector suit la branche `stable` du dépôt amont, vous pouvez l'indiquer soit dans votre fichier `.gitmodules` (pour que tout le monde le suive de même) ou juste dans votre fichier local `.git/config`. Voyons ceci dans le cas du fichier `.gitmodules` :

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
chemin du sous-module 'DbConnector' : 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687' extrait
```

Si vous ne spécifiez pas la partie `-f .gitmodules`, la commande ne fera qu'une modification locale, mais il semble plus logique d'inclure cette information dans l'historique du projet pour que tout le monde soit au diapason.

Quand vous lancez `git status`, Git vous montrera que nous avons de nouveaux *commits* (« *new commits* ») pour le sous-module.

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

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

  modifié :   .gitmodules
  modifié :   DbConnector (new commits)

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

Si vous activez le paramètre de configuration `status.submodulesummary`, Git vous montrera aussi un résumé des modifications dans vos sous-modules :

```
$ git config status.submodulesummary 1

$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

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

        modifié :   .gitmodules
        modifié :   DbConnector (new commits)

Sous-modules modifiés mais non mis à jour :

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

Ici, si vous lancez `git diff`, vous pouvez voir que le fichier `.gitmodules` a été modifié mais aussi qu'il y a un certain nombre de *commits* qui ont été tirés et sont prêts à être validés dans le projet du sous-module.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

C'est une information intéressante car vous pouvez voir le journal des modifications que vous vous apprêtez à valider dans votre sous-module. Une fois validées, vous pouvez encore visualiser cette information en lançant `git log -p`.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
```

```
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Par défaut, Git essaiera de mettre à jour **tous** les sous-modules lors d'une commande `git submodule update --remote`, donc si vous avez de nombreux sous-modules, il est préférable de spécifier le sous-module que vous souhaitez mettre à jour.

TRAVAILLER SUR UN SOUS-MODULE

Il y a fort à parier que si vous utilisez des sous-modules, vous le faîtes parce que vous souhaitez en réalité travailler sur le code du sous-module en même temps que sur celui du projet principal (ou à travers plusieurs sous-modules). Sinon, vous utiliseriez plutôt un outil de gestion de dépendances plus simple (tel que Maven ou Rubygems).

De ce fait, détaillons un exemple de modifications réalisées dans le sous-module en même temps que dans le projet principal et de validation et de publication des modifications dans le même temps.

Jusqu'à maintenant, quand nous avons lancé la commande `git submodule update` pour récupérer les modifications depuis les dépôts des sous-modules, Git récupérait les modifications et mettait les fichiers locaux à jour mais en laissant le sous-répertoire dans un état appelé « HEAD détachée ». Cela signifie qu'il n'y pas de branche locale de travail (comme `master`, par exemple) pour y valider les modifications. Donc, toutes les modifications que vous y faites ne sont pas suivies non plus.

Pour rendre votre sous-module plus adapté à la modification, vous avez besoin de deux choses. Vous devez vous rendre dans chaque sous-module et ex-

traire une branche de travail. Ensuite vous devez dire à Git ce qu'il doit faire si vous avez réalisé des modifications et que vous lancez `git submodule update --remote` pour tirer les modifications amont. Les options disponibles sont soit de les fusionner dans votre travail local, soit de tenter de rebaser le travail local par dessus les modifications distantes.

En premier, rendons-nous dans le répertoire de notre sous-module et extrayons une branche.

```
$ git checkout stable
Basculement sur la branche 'stable'
```

Attaquons-nous au choix de politique de gestion. Pour le spécifier manuellement, nous pouvons simplement ajouter l'option `--merge` à l'appel de `update`. Nous voyons ici qu'une modification était disponible sur le serveur pour ce sous-module et qu'elle a été fusionnée.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Mise à jour de c87d55d..92c7337
Avance rapide
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
chemin du sous-module 'DbConnector': fusionné dans '92c7337b30ef9e0893e758dac2459c
```

Si nous nous rendons dans le répertoire `DbConnector`, les nouvelles modifications sont déjà fusionnées dans notre branche locale `stable`. Voyons maintenant ce qui arrive si nous modifions localement la bibliothèque et que quelqu'un pousse une autre modification en amont dans le même temps.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Maintenant, si nous mettons à jour notre sous-module, nous pouvons voir ce qui arrive lors d'un rebasage de deux modifications concurrentes.

```
$ git submodule update --remote --rebase
Premièrement, rembobinons head pour rejouer votre travail par-dessus...
Application : unicode support
chemin du sous-module 'DbConnector': rebasé dans '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Si vous oubliez de spécifier `--rebase` ou `--merge`, Git mettra juste à jour le sous-module vers ce qui est sur le serveur et réinitialisera votre projet à l'état « HEAD détachée ».

```
$ git submodule update --remote
chemin du sous-module 'DbConnector' : '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94' extrait
```

Si cela arrive, ne vous inquiétez pas, vous pouvez simplement revenir dans le répertoire et extraire votre branche (qui contiendra encore votre travail) et fusionner ou rebaser `origin/stable` (ou la branche distante que vous souhaitez) à la main.

Si vous n'avez pas validé vos modifications dans votre sous-module, et que vous lancez une mise à jour de sous-module qui causerait des erreurs, Git récupérera les modifications mais n'écrasera pas le travail non validé dans votre répertoire de sous-module.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Dépaquetage des objets: 100% (4/4), fait.
Depuis https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Vos modifications locales seraient écrasées par checkout:
        scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Impossible d'extraire 'c75e92a2b3855c9e5b66f915308390d9db204aca' dans le chemin du sous-modu
```

Si vous avez réalisé des modifications qui entrent en conflit avec des modifications amont, Git vous en informera quand vous mettrez à jour.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLIT (contenu): Conflit de fusion dans scripts/setup.sh
```

```
La fusion automatique a échoué ; réglez les conflits et validez le résultat
Impossible de fusionner 'c75e92a2b3855c9e5b66f915308390d9db204aca' dans le chemin
```

Vous pouvez vous rendre dans le répertoire du sous-module et résoudre le conflit normalement.

PUBLIER LES MODIFICATIONS DANS UN SOUS-MODULE

Nous avons donc des modifications dans notre répertoire de sous-module, venant à la fois du dépôt amont et de modifications locales non publiées.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnessesary method
  > add new option for conn pooling
```

Si nous validons dans le projet principal et que nous le poussons en amont sans pousser les modifications des sous-modules, les autres personnes qui voudront essayer notre travail vont avoir de gros problèmes vu qu'elles n'auront aucun moyen de récupérer les modifications des sous-modules qui en font partie. Ces modifications n'existent que dans notre copie locale.

Pour être sur que cela n'arrive pas, vous pouvez demander à Git de vérifier que tous vos sous-modules ont été correctement poussés avant de pouvoir pousser le projet principal. La commande `git push` accepte un argument `--recurse-submodules` qui peut avoir pour valeur « *check* » ou « *on-demand* ». L'option « check » fera échouer `push` si au moins une des modifications des sous-modules n'a pas été poussée.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

        git push --recurse-submodules=on-demand

or cd to the path and use

        git push
```

```
to push them to a remote.
```

Comme vous pouvez le voir, il donne aussi quelques conseils utiles sur ce que nous pourrions vouloir faire ensuite. L'option simple consiste à se rendre dans chaque sous-module et à pousser manuellement sur les dépôts distants pour s'assurer qu'ils sont disponibles publiquement, puis de réessayer de pousser le projet principal.

L'autre option consiste à utiliser la valeur « on-demand » qui essaiera de faire tout ceci pour vous.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Décompte des objets: 9, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (8/8), fait.
Écriture des objets: 100% (9/9), 917 bytes | 0 bytes/s, fait.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Décompte des objets: 2, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (2/2), fait.
Écriture des objets: 100% (2/2), 266 bytes | 0 bytes/s, fait.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

Comme vous pouvez le voir, Git s'est rendu dans le module `DbConnector` et l'a poussé avant de pousser le projet principal. Si la poussée du sous-module échoue pour une raison quelconque, la poussée du projet principal sera annulée.

FUSION DE MODIFICATIONS DE SOUS-MODULES

Si vous changez la référence d'un sous-module en même temps qu'une autre personne, il se peut que cela pose problème. Particulièrement, si les historiques des sous-modules ont divergé et sont appliqués à des branches divergentes dans un super-projet, rapprocher toutes les modifications peut demander un peu de travail.

Si un des *commits* est un ancêtre direct d'un autre (c'est-à-dire une fusion en avance rapide), alors Git choisira simplement ce dernier pour la fusion et cela se résoudra tout seul. Cependant, Git ne tentera pas de fusion, même très simple,

pour vous. Si les *commits* d'un sous-module divergent et doivent être fusionnés, vous obtiendrez quelque chose qui ressemble à ceci :

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Dépaquetage des objets: 100% (2/2), fait.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Fusion automatique de DbConnector
CONFLIT (sous-module): Conflit de fusion dans DbConnector
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Donc, ce qui s'est passé en substance est que Git a découvert que les deux points de fusion des branches à fusionner dans l'historique du sous-module sont divergents et doivent être fusionnés. Il l'explique par « *merge following commits not found* » (fusion suivant les *commits* non trouvée), ce qui n'est pas clair mais que nous allons expliquer d'ici peu.

Pour résoudre le problème, vous devez comprendre l'état dans lequel le sous-module devrait se trouver. Étrangement, Git ne vous donne pas d'information utile dans ce cas, pas même les SHA-1 des *commits* des deux côtés de l'historique. Heureusement, c'est assez facile à comprendre. Si vous lancez `git diff`, vous pouvez obtenir les SHA-1 des *commits* enregistrés dans chacune des branches que vous essayiez de fusionner.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Donc, dans ce cas, `eb41d76` est le *commit* dans notre sous-module que **nous** avions et `c771610` est le *commit* amont. Si nous nous rendons dans le répertoire du sous-module, il devrait déjà être sur `eb41d76` parce que la fusion ne l'a pas touché. S'il n'y est pas, vous pouvez simplement créer et extraire une branche qui pointe dessus.

Ce qui importe, c'est le SHA-1 du *commit* venant de l'autre branche. C'est ce que nous aurons à fusionner. Vous pouvez soit essayer de fusionner avec le SHA-1 directement ou vous pouvez créer une branche à partir du *commit* puis

essayer de la fusionner. Nous suggérons d'utiliser cette dernière méthode, ne serait-ce que pour obtenir un message de fusion plus parlant.

Donc, rendons-nous dans le répertoire du sous-module, créons une branche basée sur ce second SHA-1 obtenu avec `git diff` et fusionnons manuellement.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Fusion automatique de src/main.c
CONFLIT (contenu): Conflit de fusion dans src/main.c
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Nous avons eu un conflit de fusion ici, donc si nous le résolvons et validons, alors nous pouvons simplement mettre à jour le projet principal avec le résultat.

```
$ vim src/main.c ❶
$ git add src/main.c
$ git commit -am 'fusion de nos modifications'
[master 9fd905e] fusion de nos modifications

$ cd .. ❷
$ git diff ❸
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ❹

$ git commit -m "Fusion du travail de Tom" ❺
[master 10d2c60] Fusion du travail de Tom
```

❶ Nous résolvons le conflit

❷ Ensuite, nous retournons dans le projet principal

❸ Nous pouvons revérifier les SHA-1

❹ Nous résolvons l'entrée en conflit dans le sous-module

❺ Enfin, nous validons la résolution.

Cela peut paraître compliqué mais ce n'est pas très difficile.

Curieusement, il existe un autre cas que Git gère seul. Si un *commit* de fusion existe dans le répertoire du sous-module qui contient **les deux** *commits* dans ses ancêtres, Git va le suggérer comme solution possible. Il voit qu'à un certain point de l'historique du projet du sous-module, quelqu'un a fusionné les branches contenant ces deux *commits*, donc vous désirerez peut-être utiliser celui-ci.

C'est pourquoi le message d'erreur précédent s'intitulait « *merge following commits not found* », parce que justement, il ne pouvait pas trouver **le *commit* de fusion**. C'est déroutant car qui s'attendrait à ce qu'il **essaie** de le chercher ?

S'il trouve un seul *commit* de fusion acceptable, vous verrez ceci :

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "Db

which will accept this suggestion.
Fusion automatique de DbConnector
CONFLIT (submodule): Conflit de fusion dans DbConnector
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

Ce qu'il suggère de faire est de mettre à jour l'index comme si vous aviez lancé `git add`, ce qui élimine le conflit, puis de valider. Vous ne devriez cependant pas le faire. Vous pouvez plus simplement vous rendre dans le répertoire du sous-module, visualiser la différence, avancer en avance rapide sur le *commit*, le tester puis le valider.

```
$ cd DbConnector/
$ git merge 9fd905e
Mise à jour eb41d76..9fd905e
Avance rapide

$ cd ..
```

```
$ git add DbConnector
$ git commit -am 'Avance rapide sur un fils commun dans le sous-module'
```

Cela revient au même, mais de cette manière vous pouvez au moins vérifier que ça fonctionne et vous avez le code dans votre répertoire de sous-module quand c'est terminé.

Trucs et astuces pour les sous-modules

Il existe quelques commandes qui permettent de travailler plus facilement avec les sous-modules.

SUBMODULE FOREACH

Il existe une commande submodule `foreach` qui permet de lancer une commande arbitraire dans chaque sous-module. C'est particulièrement utile si vous avez plusieurs sous-modules dans le même projet.

Par exemple, supposons que nous voulons développer une nouvelle fonctionnalité ou faire un correctif et que nous avons déjà du travail en cours dans plusieurs sous-modules. Nous pouvons facilement remiser tout le travail en cours dans tous les sous-modules.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Ensuite, nous pouvons créer une nouvelle branche et y basculer dans tous nos sous-modules.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Basculement sur la nouvelle branche 'featureA'
Entering 'DbConnector'
Basculement sur la nouvelle branche  'featureA'
```

Vous comprenez l'idée. Une commande vraiment utile permet de produire un joli diff unifié des modifications dans le projet principal ainsi que dans tous les sous-projets.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

Ici, nous pouvons voir que nous définissons une fonction dans un sous-module et que nous l'appelons dans le projet principal. C'est un exemple exagérément simplifié, mais qui aide à mieux comprendre l'utilité de cette commande.

ALIAS UTILES

Vous pourriez être intéressé de définir quelques alias pour des commandes longues pour lesquelles vous ne pouvez pas régler la configuration par défaut. Nous avons traité la définition d'alias Git dans **"Les alias Git"**, mais voici un exemple d'alias que vous pourriez trouver utiles si vous voulez travailler sérieusement avec les sous-modules de Git.

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

De cette manière, vous pouvez simplement lancer `git supdate` lorsque vous souhaitez mettre à jour vos sous-module ou `git spush` pour pousser avec une gestion de dépendance de sous-modules.

Les problèmes avec les sous-modules

Cependant, utiliser des sous-modules ne se déroule pas sans accroc.

Commuter des branches qui contiennent des sous-modules peut également s'avérer difficile. Si vous créez une nouvelle branche, y ajoutez un sous-module, et revenez ensuite à une branche dépourvue de ce sous-module, vous aurez toujours le répertoire de ce sous-module comme un répertoire non suivi :

```
$ git checkout -b add-crypto
Basculement sur la nouvelle branche 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Clonage dans 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.

$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Fichiers non suivis :
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        CryptoLibrary/

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (util
```

Supprimer le répertoire n'est pas difficile, mais sa présence est assez déroutante. Si vous le supprimez puis que vous rebasculez sur la branche qui contient le sous-module, vous devrez lancer `submodule update --init` pour le repopuler.

```
$ git clean -ffdx
Suppression de CryptoLibrary/

$ git checkout add-crypto
Basculement sur la branche 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e026854517

$ ls CryptoLibrary/
Makefile        includes        scripts         src
```

Une fois de plus, ce n'est réellement difficile, mais cela peut être déroutant.

Une autre difficulté commune consiste à basculer de sous-répertoires en sous-modules. Si vous suiviez des fichiers dans votre projet et que vous voulez les déplacer dans un sous-module, vous devez être très prudent ou Git sera inflexible. Présumons que vous avez les fichiers dans un sous-répertoire de votre projet, et que vous voulez les transformer en un sous-module. Si vous supprimez le sous-répertoire et que vous exécutez `submodule add`, Git vous hurle dessus avec :

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Vous devez d'abord supprimer le répertoire `CryptoLibrary` de l'index. Vous pourrez ensuite ajouter le sous-module :

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
```

```
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Maintenant, supposons que vous avez fait cela dans une branche. Si vous essayez de basculer dans une ancienne branche où ces fichiers sont toujours dans l'arbre de projet plutôt que comme sous-module, vous aurez cette erreur :

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

Vous pouvez le forcer à basculer avec `checkout -f`, mais soyez attentif à ce qu'il soit propre ou les modifications seraient écrasées.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Basculement sur la branche 'master'
```

Ensuite, lorsque vous rebasculez, vous aurez un répertoire `CryptoLibrary` vide et `git submodule update` pourrait ne pas le remettre en état. Vous allez devoir vous rendre dans le répertoire de votre sous-module et lancer `git checkout .` pour retrouver tous vos fichiers. Vous pouvez lancer ceci dans un script `submodule foreach` dans le cas de multiples sous-modules.

Il est important de noter que depuis les versions de Git récentes, les sous-modules conservent leurs données Git dans le répertoire `.git` du projet principal, ce qui à la différence des versions antérieures, permet de supprimer le dossier du sous-module sans perdre les *commits* et les branches qu'il contenait.

Avec ces outils, les sous-modules peuvent être une méthode assez simple et efficace pour développer simultanément sur des projets connexes mais séparés.

Empaquetage (*bundling*)

Bien que nous ayons déjà abordé les méthodes les plus communes de transfert de données Git par réseau (HTTP, SSH, etc.), il existe en fait une méthode supplémentaire qui n'est pas beaucoup utilisée mais qui peut s'avérer utile.

Git est capable d'empaqueter ses données dans un fichier unique. Ceci peut servir dans de nombreux scénarios. Le réseau peut être en panne et vous souhaitez envoyer des modifications à vos collègues. Peut-être êtes-vous en train de travailler à distance et vous ne pouvez pas vous connecter au réseau local pour des raisons de sécurité. Peut-être que votre carte réseau ou votre carte wi-fi vient de tomber en panne. Peut-être encore n'avez-vous pas accès à un serveur partagé, et vous souhaitez envoyer à quelqu'un des mises à jour sans devoir transférer 40 *commits* via `format-patch`.

Ce sont des situations où la commande `git bundle` est utile. La commande `bundle` va empaqueter tout ce qui serait normalement poussé sur le réseau avec une commande `git push` dans un fichier binaire qui peut être envoyé à quelqu'un par courriel ou copié sur une clé USB, puis de le dépaqueter dans un autre dépôt.

Voyons un exemple simple. Supposons que vous avez un dépôt avec deux *commits* :

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

Si vous souhaitez envoyer ce dépôt à quelqu'un et que vous n'avez pas accès en poussée à un dépôt, ou que simplement vous ne voulez pas en créer un, vous pouvez l'empaqueter avec `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Décompte des objets: 6, fait.
Delta compression using up to 2 threads.
Compression des objets: 100% (2/2), fait.
Écriture des objets : 100% (6/6), 441 bytes, fait.
Total 6 (delta 0), reused 0 (delta 0)
```

À présent, vous avez un fichier `repo.bundle` qui contient toutes les données nécessaires pour recréer la branche `master` du dépôt. Avec la commande

`bundle`, vous devez lister toutes les références ou les intervalles spécifiques de *commits* que vous voulez inclure. Si vous le destinez à être cloné ailleurs, vous devriez aussi ajouter HEAD comme référence, comme nous l'avons fait.

Vous pouvez envoyer ce fichier `repo.bundle` par courriel, ou le copier sur une clé USB et la tendre à un collègue.

De l'autre côté, supposons qu'on vous a envoyé ce fichier `repo.bundle` et que vous voulez travailler sur le projet. Vous pouvez cloner le fichier binaire dans un répertoire, de la même manière que vous le feriez pour une URL.

```
$ git clone repo.bundle repo
Initialized empty Git repository in /private/tmp/bundle/repo/.git/
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

Si vous n'incluez pas HEAD dans les références, vous devez aussi spécifier `-b master` ou n'importe quelle branche incluse dans le paquet car sinon, il ne saura pas quelle branche extraire.

Maintenant, supposons que vous faites 3 *commits* et que vous voulez renvoyer ces nouveaux *commits* via courriel ou clé USB.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

Nous devons déjà déterminer l'intervalle de *commits* que nous voulons inclure dans le colis. À la différence des protocoles réseau qui calculent automatiquement l'ensemble minimum des données à transférer, nous allons devoir les définir manuellement. Ici, vous pourriez tout à fait lancer la même commande et empaqueter le dépôt complet, ce qui marcherait mais c'est mieux de n'empaqueter que la différence - seulement les 3 *commits* que nous avons localement créés.

Pour le faire, vous allez devoir calculer la différence. Comme décrit dans **"Plages de *commits*"**, vous pouvez faire référence à un intervalle de *commits* de différentes manières. Pour désigner les trois *commits* que nous avons dans notre branche master et qui n'était pas dans la branche que nous avons initialement clonée, nous pouvons utiliser quelque chose comme `origin/`

`master..master` ou `master ^origin/master`. Vous pouvez tester cela avec la sortie de la commande `log`.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

Comme nous avons maintenant la liste des *commits* que nous voulons inclure dans le colis, empaquetons-les. Cela est réalisé avec la commande `git bundle create`, suivie d'un nom de fichier et des intervalles des *commits* que nous souhaitons inclure.

```
$ git bundle create commits.bundle master ^9a466c5
Comptage des objets : 11, fait.
Delta compression using up to 2 threads.
Compression des objets : 100% (3/3), fait.
Écriture de objets : 100% (9/9), 775 bytes, fait.
Total 9 (delta 0), reused 0 (delta 0)
```

Nous avons à présent un fichier `commits.bundle` dans notre répertoire. Si nous le prenons et l'envoyons à un partenaire, il pourra l'importer dans le dépôt d'origine, même si du travail a été ajouté entre temps.

Quand il récupère le colis, il peut l'inspecter pour voir ce qu'il contient avant de l'importer dans son dépôt. La première commande est `bundle verify` qui va s'assurer que le fichier est une fichier bundle Git valide et que le dépôt contient tous les ancêtres nécessaires pour appliquer correctement le colis.

```
$ git bundle verify ../commits.bundle
Le colis contient 1 référence :
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
Le colis exige cette référence
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle est correct
```

Si la personne avait créé un colis ne contenant que les deux derniers commits qu'il avait ajoutés, plutôt que les trois, le dépôt initial n'aurait pas pu l'importer, car il aurait manqué un commit dans l'historique à reconstituer. La commande `verify` aurait ressemblé plutôt à ceci :

```
$ git bundle verify ../commits-bad.bundle
error: Le dépôt ne dispose pas des commits prérequis suivants :
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

Cependant, notre premier colis est valide, et nous pouvons récupérer des *commits* depuis celui-ci. Si vous souhaitez voir les branches présentes dans le colis qui peuvent être importées, il y a aussi une commande pour donner la liste des sommets des branches :

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

La sous-commande `verify` vous indiquera aussi les sommets. L'objectif est de voir ce qui peut être tiré, pour que vous puissiez utiliser les commandes `fetch` et `pull` pour importer des *commits* depuis le colis. Ici, nous allons récupérer la branche `master` du colis dans une branche appelée `other-master` dans notre dépôt :

```
$ git fetch ../commits.bundle master:other-master
Depuis ../commits.bundle
 * [nouvelle branche]      master     -> other-master
```

Maintenant, nous pouvons voir que nous avons importé les *commits* sur la branche `other-master` ainsi que tous les *commits* que nous avons validés entre-temps dans notre propre branche `master`.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

Ainsi, `git bundle` peut vraiment être utile pour partager du code ou réaliser des opérations nécessitant du réseau quand il n'y a pas de réseau ou de dépôt partagé.

Replace

Git manipule des objets immuables mais il fournit un moyen de faire comme s'il pouvait remplacer des objets de sa base de données par d'autres objets.

La commande `replace` vous permet de spécifier un objet dans Git et de lui indiquer : « chaque fois que tu vois ceci, fais comme si c'était cette autre chose ». Ceci sert principalement à remplacer un *commit* par un autre dans votre historique.

Par exemple, supposons que vous avez un énorme historique de code et que vous souhaitez scinder votre dépôt en un historique court pour les nouveaux développeurs et un plus important et long pour ceux intéressés par des statistiques. Vous pouvez générer un historique depuis l'autre avec `replace` en remplaçant le *commit* le plus ancien du nouvel historique par le dernier *commit* de l'historique ancien. C'est sympa parce que cela signifie que vous n'avez pas besoin de réécrire tous les *commits* du nouvel historique, comme vous devriez le faire pour les joindre tous les deux (à cause de l'effet de lien des SHA-1).

Voyons ce que ça donne. Prenons un dépôt existant, découpons-le en deux dépôts, un récent et un historique, puis nous verrons comment les recombiner sans modifier les valeurs SHA-1 du dépôt récent, grâce à `replace`.

Nous allons utiliser un dépôt simple avec cinq *commit* simples :

```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Nous souhaitons couper ceci en deux lignes d'historiques. Une ligne ira de *first commit* à *fourth commit* et sera la ligne historique. La seconde ligne ira de *fourth commit* à *fifth commit* et sera ligne récente.

FIGURE 7-28

Bien, la création de la ligne historique est simple, nous n'avons qu'à créer une branche dans l'historique et la pousser vers la branche `master` d'un nouveau dépôt distant.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

FIGURE 7-29

Maintenant, nous pouvons pousser la nouvelle branche `history` vers la branche `master` du nouveau dépôt :

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Décompte des objets : 12, fait.
Delta compression using up to 2 threads.
Compression des objets : 100% (4/4), fait.
Écriture des objets : 100% (12/12), 907 bytes, fait.
Total 12 (delta 0), reused 0 (delta 0)
Dépaquetage des objets : 100% (12/12), fait.
To git@github.com:schacon/project-history.git
 * [nouvelle branche]      history -> master
```

Bien, notre projet historique est publié. Maintenant, la partie la plus compliquée consiste à tronquer l'historique récent pour le raccourcir. Nous avons besoin d'un recouvrement pour pouvoir remplacer un *commit* dans un historique par un équivalent dans l'autre, donc nous allons tronquer l'historique à *fourth commit* et *fifth commit*, pour que *fourth commit* soit en recouvrement.

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Il peut être utile de créer un *commit* de base qui contient les instructions sur la manière d'étendre l'historique, de sorte que les autres développeurs puissent savoir comment s'y prendre s'ils butent sur le premier *commit* et ont besoin de plus d'histoire. Donc, ce que nous allons faire, c'est créer un objet *commit* initial comme base avec les instructions, puis rebaser les *commits* restants (quatre et cinq) dessus.

Nous avons besoin de choisir un point de découpe, qui pour nous est *third commit*, soit le SHA-1 `9c68fdc`. Donc, notre *commit* de base sera créé sur cet arbre. Nous pouvons créer notre *commit* de base en utilisant la commande `commit-tree`, qui accepte juste un arbre et nous fournit un SHA-1 d'un objet *commit* orphelin tout nouveau.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

La commande `commit-tree` fait partie de ce qu'on appelle les commandes de « plomberie ». Ce sont des commandes qui ne sont pas destinées à être utilisées directement, mais plutôt au sein d'**autres** commandes Git en tant que petits utilitaires. Dans les occasions où nous faisons des choses plus bizarres que de coutume comme actuellement, elles nous permettent de faire des actions de bas niveau qui ne sont pas destinées à une utilisation quotidienne. Pour en savoir plus sur les commandes de plomberie, référez-vous à **"Plomberie et porcelaine"**.

FIGURE 7-30

OK, donc maintenant avec un *commit* de base, nous pouvons rebaser le reste de notre historique dessus avec la commande `git rebase --onto`. L'argument `--onto` sera l'empreinte SHA-1 que nous venons tout juste de récupérer avec la commande `commit-tree` et le point de rebasage sera `third commit` (le parent du premier *commit* que nous souhaitons garder, `9c68fdc`).

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

FIGURE 7-31

Bien, nous avons donc réécrit l'historique récent à la suite du *commit* de base qui contient les instructions pour reconstruire l'historique complet. Nous pouvons pousser ce nouvel historique vers un nouveau projet et quand des personnes clonent ce dépôt, elles ne voient que les deux *commits* les plus récents et un *commit* avec des instructions.

Inversons les rôles et plaçons-nous dans la position d'une personne qui clone le projet pour la première fois et souhaite obtenir l'historique complet. Pour obtenir les données d'historique après avoir cloné ce dépôt tronqué, on doit ajouter un second dépôt distant pointant vers le dépôt historique et tout récupérer :

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
```

```
$ git fetch project-history
From https://github.com/schacon/project-history
 * [nouvelle branche]      master     -> project-history/master
```

À présent, le collaborateur aurait les *commits* récents dans la branche `master` et les *commits* historiques dans la branche `project-history/master`.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Pour combiner ces deux branches, vous pouvez simplement lancer `git replace` avec le *commit* que vous souhaitez remplacer suivi du *commit* qui remplacera. Donc nous voulons remplacer `fourth commit` dans la branche `master` par `fourth commit` de la branche `project-history/master` :

```
$ git replace 81a708d c6e1e95
```

Maintenant, quand on regarde l'historique de la branche `master`, il apparaît comme ceci :

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Sympa, non ? Sans devoir changer tous les SHA-1 en amont, nous avons pu remplacer un *commit* dans notre historique avec un autre entièrement différent et tous les outils normaux (`bisect`, `blame`, etc) fonctionnent de manière transparente.

FIGURE 7-32

Ce qui est intéressant, c'est que `fourth commit` a toujours un SHA-1 de `81a708d`, même s'il utilise en fait les données du *commit* `c6e1e95` par lequel nous l'avons remplacé. Même si vous lancez une commande comme `cat-file`, il montrera les données remplacées :

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Souvenez-vous que le parent réel de `81a708d` était notre *commit* de base (`622e88e`) et non `9c68fdce` comme indiqué ici.

Une autre chose intéressante est que les données sont conservées dans nos références :

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/81a708dd0e167a3f69154
```

Ceci signifie qu'il est facile de partager notre remplacement avec d'autres personnes, puisque nous pouvons pousser ceci sur notre serveur et d'autres personnes pourrons le télécharger. Ce n'est pas très utile dans le cas de la reconstruction d'historique que nous venons de voir (puisque tout le monde téléchargerait quand même les deux historiques, pourquoi alors les séparer ?), mais cela peut être utile dans d'autres circonstances.

Stockage des identifiants

Si vous utilisez le transport SSH pour vous connecter à vos dépôts distants, il est possible d'avoir une clé sans mot de passe qui permet de transférer des données en sécurité sans devoir entrer un nom d'utilisateur et un mot de passe. Cependant, ce n'est pas possible avec les protocoles HTTP - toute connexion nécessite un nom d'utilisateur et un mot de passe. Cela devient même plus difficile avec des systèmes à authentification à deux facteurs, où le mot de passe utilisé est généré dynamiquement au hasard et devient imprononçable.

Heureusement, Git dispose d'un système de gestion d'identifiants qui peut faciliter cette gestion. Git propose de base quelques options :

- Par défaut, rien n'est mis en cache. Toutes les connexions vous demanderont votre nom d'utilisateur et votre mot de passe.
- Le mode « cache » conserve en mémoire les identifiants pendant un certain temps. Aucun mot de passe n'est stocké sur le disque et les identifiants sont oubliés après 15 minutes.
- Le mode « store » sauvegarde les identifiants dans un fichier texte simple sur le disque, et celui-ci n'expire jamais. Ceci signifie que tant que vous ne changerez pas votre mot de passe sur le serveur Git, vous n'aurez plus à entrer votre mot de passe. Le défaut de cette approche et que vos mots de passe sont stockés en clair dans un fichier texte dans votre répertoire personnel.
- Si vous utilisez un Mac, Git propose un mode « osxkeychain », qui met en cache les identifiants dans un trousseau sécurisé attaché à votre compte système.

- Si vous utilisez Windows, vous pouvez installer une application appelée « winstore ». C'est similaire à l'assistant « osxkeychain » décrit ci-dessus, mais utilise le *Windows Credential Store* pour sauvegarder les informations sensibles. *winstore* peut être téléchargé à ***https://gitcredentialstore.codeplex.com***.

Vous pouvez choisir une de ces méthodes en paramétrant une valeur de configuration Git :

```
$ git config --global credential.helper cache
```

Certains de ces assistants ont des options. L'assistant « store » accepte un argument `--file <chemin>` qui permet de personnaliser l'endroit où le fichier texte est sauvegardé (par défaut, c'est `~/.git-credentials`). L'assistant `cache` accepte une option `--timeout <secondes>` qui modifie la période de maintien en mémoire (par défaut, 900, soit 15 minutes). Voici un exemple de configuration de l'option « store » avec un nom de fichier personnalisé :

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git vous permet même de configurer plusieurs assistants. Lors de la recherche d'identifiants pour un serveur donné, Git les interrogera dans l'ordre jusqu'à la première réponse. Pour la sauvegarde des identifiants, Git enverra le nom d'utilisateur et le mot de passe à **tous** les assistants et ceux-ci pourront choisir ce qu'ils en font. Voici à quoi ressemblerait un `.gitconfig` si vous utilisiez un fichier d'identifiants sur une clé USB mais souhaiteriez utilisez l'option de cache pour éviter des frappes trop fréquentes si la clé n'est pas insérée.

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Sous le capot

Comment tout ceci fonctionne-t-il ? La commande d'origine de Git pour le système d'assistants d'indentification est `git credential`, qui accepte une commande comme argument, puis d'autres informations via stdin.

Un exemple peut aider à mieux comprendre cela. Supposons qu'un assistant d'identification a été configuré et que l'assistant a stocké les identifiants pour `mygithost`. Voici une session qui utilise la commande « fill » qui est invoquée quand Git essaie de trouver les identifiants pour un hôte :

```
$ git credential fill ❶
protocol=https ❷
host=mygithost
❸
protocol=https ❹
host=mygithost
username=bob
password=s3cre7
$ git credential fill ❺
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

❶ C'est la ligne de commande qui démarre l'interaction.

❷ Git-credential attend la saisie d'informations sur stdin. Nous lui fournissons les informations que nous connaissons : le protocole et le nom d'hôte.

❸ Une ligne vide indique que l'entrée est complète et le système d'identification devrait répondre avec les informations qu'il connaît.

❹ Git-credential prend alors la main et écrit sur la sortie standard les informations qu'il a trouvées.

❺ Si aucune information d'identification n'a été trouvée, Git demande le nom d'utilisateur et le mot de passe, et les fournit sur la sortie standard d'origine (ici elles sont rattachées à la même console).

Le système d'aide à l'identification invoque en fait un programme complètement séparé de Git lui-même. Lequel est invoqué et comment il est invoqué dépendent de la valeur de configuration `credential.helper`. Cette valeur peut prendre plusieurs formes :

Valeur de configuration	Comportement
`foo`	lance `git-credential-foo`
`foo -a --opt=bcd`	lance `git-credential-foo -a --opt=bcd`

Valeur de configuration	Comportement
`/chemin/absolu/foo -xyz`	lance `/chemin/absolu/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Le code après `!` est évalué dans un shell

Donc les assistants décrits ci-dessus sont en fait appelés `git-credential-cache`, `git-credential-store`, et ainsi de suite et nous pouvons les configurer pour accepter des arguments en ligne de commande.

La forme générale pour ceci est `git-credential-foo [args] <action>`. Le protocole stdin/stdout est le même que pour git-credential, mais en utilisant un ensemble d'actions légèrement différent :

- `get` est une requête pour une paire nom d'utilisateur/mot de passe.
- `store` est une requête pour sauvegarder des identifiants dans la mémoire de l'assistant.
- `erase` purge de la mémoire de l'assistant les identifiants répondants aux critères.

Pour les actions `store` et `erase`, aucune réponse n'est exigée (Git les ignore de toute façon). Pour l'action `get` cependant, Git est très intéressé par ce que l'assistant peut en dire. Si l'assistant n'a rien à en dire d'utile, il peut simplement sortir sans rien produire, mais s'il sait quelque chose, il devrait augmenter l'information fournie avec celle qu'il a stockée. La sortie est traitée comme une série de déclarations d'affectation ; tout ce qui est fourni remplacera ce que Git connaît déjà.

Voici le même exemple que ci-dessus, mais en sautant git-credential et en s'attaquant directement à git-credential-store :

```
$ git credential-store --file ~/git.store store ❶
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ❷
protocol=https
host=mygithost

username=bob ❸
password=s3cre7
```

❶ Ici nous indiquons à `git-credential-store` de sauvegarder des identifiants : le nom d'utilisateur (*username*) « bob » et le mot de passe (*password*) « s3cre7 » doivent être utilisés quand `https://mygithost` est accédé.

❷ Maintenant, nous allons récupérer ces identifiants. Nous fournissons les parties de l'information de connexion que nous connaissons (`https://mygithost`), suivi d'une ligne vide.

❸ `git-credential-store` répond avec le nom d'utilisateur et le mot de passe que nous avons précédemment stockés.

Voici à quoi ressemble le fichier `~/git.store` :

```
https://bob:s3cre7@mygithost
```

C'est juste une série de lignes, chacune contenant des URLs contenant les informations d'identification. Les assistants `osxkeychain` et `winstore` utilisent le format natif de leurs banques de stockage, tandis que `cache` utilise son propre format en mémoire (qu'aucun autre processus ne peut lire).

Un cache d'identifiants personnalisé

Étant donné que `git-credential-store` et consort sont des programmes séparés de Git, il y a peu à penser que *n'importe quel* programme peut être un assistant d'identification Git. Les assistants fournis par Git gèrent de nombreux cas d'utilisation habituels, mais pas tous. Par exemple, supposons que votre équipe dispose de certains identifiants qui sont partagés par tous, pour le déploiement. Ils sont stockés dans un répertoire partagé, mais vous ne les copiez pas dans votre propre magasin d'identifiants parce qu'ils changent souvent. Aucun assistant existant ne gère ce cas ; voyons ce qu'il faudrait pour écrire le nôtre. Ce programme doit présenter certaines fonctionnalités clé :

1. La seule action à laquelle nous devons répondre est `get` ; `store` et `erase` sont des opérations d'écriture, donc nous sortirons directement et proprement dans ces cas.
2. Le format du fichier d'identifiants partagés est identique à celui utilisé par `git-credential-store`.
3. L'emplacement de ce fichier est assez standard, mais nous devrions pouvoir laisser l'utilisateur spécifier une chemin en cas de besoin.

Une fois de plus, nous écrirons cette extension en Ruby, mais n'importe quel langage fonctionnera, tant que Git peut lancer un exécutable à la fin. Voici le code source complet de ce nouvel assistant d'identification :

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ❶
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ❷
exit(0) unless File.exists? path

known = {} ❸
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| ❹
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

❶ Ici, nous analysons les options de la ligne de commande, pour permettre à l'utilisateur de spécifier un fichier. Par défaut, c'est `~/.git-credentials`.

❷ Ce programme ne répondra que si l'action est `get` et si le fichier magasin existe.

❸ Cette boucle lit depuis stdin jusqu'à la première ligne vide. Les entrées sont stockées dans le *hash* known pour référence ultérieure.

❹ Cette boucle lit le contenu du fichier magasin, et recherche les correspondances. Si le protocole et l'hôte depuis known correspondent à la ligne, le programme imprime les résultats sur stdout et sort.

Nous allons sauvegarder notre assistant comme `git-credential-read-only`, le placer quelque part dans notre PATH et le marquer exécutable. Voici à quoi ressemble une session interactive :

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Puisque son nom commence par `git-`, nous pouvons utiliser une syntaxe simple pour la valeur de configuration :

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Comme vous pouvez le voir, étendre ce système est plutôt direct et peut résoudre des problèmes communs pour vous et votre équipe.

Résumé

Vous venez de voir certains des outils avancés vous permettant de manipuler vos *commits* et votre index plus précisément. Lorsque vous remarquez des bogues, vous devriez être capable de trouver facilement quelle validation les a introduits, quand et par qui. Si vous voulez utiliser des sous-projets dans votre projet, vous avez appris plusieurs façons de les gérer. À partir de maintenant, vous devez être capable de faire la plupart de ce dont vous avez besoin avec Git en ligne de commande et de vous y sentir à l'aise.

Personnalisation de Git 8

Jusqu'ici, nous avons traité les bases du fonctionnement et de l'utilisation de Git et introduit un certain nombre d'outils fournis par Git pour travailler plus facilement et plus efficacement. Dans ce chapitre, nous aborderons quelques opérations permettant d'utiliser Git de manière plus personnalisée en vous présentant quelques paramètres de configuration importants et le système d'interceptions. Grâce à ces outils, il devient enfantin de faire fonctionner Git exactement comme vous, votre société ou votre communauté en avez besoin.

Configuration de Git

Comme vous avez pu l'entrevoir dans **Chapter 1**, vous pouvez spécifier les paramètres de configuration de Git avec la commande `git config`. Une des premières choses que vous avez faites a été de paramétrer votre nom et votre adresse de courriel :

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

À présent, vous allez apprendre quelques-unes des options similaires les plus intéressantes pour paramétrer votre usage de Git.

Vous avez vu des détails de configuration simple de Git au premier chapitre, mais nous allons les réviser. Git utilise une série de fichiers de configuration pour déterminer son comportement selon votre personnalisation. Le premier endroit que Git visite est le fichier `/etc/gitconfig` qui contient des valeurs pour tous les utilisateurs du système et tous leurs dépôts. Si vous passez l'option `--system` à `git config`, il lit et écrit ce fichier.

L'endroit suivant visité par Git est le fichier `~/.gitconfig` qui est spécifique à chaque utilisateur. Vous pouvez faire lire et écrire Git dans ce fichier au moyen de l'option `--global`.

Enfin, Git recherche des valeurs de configuration dans le fichier de configuration du répertoire Git (`.git/config`) du dépôt en cours d'utilisation. Ces valeurs sont spécifiques à un unique dépôt.

Chaque niveau surcharge le niveau précédent, ce qui signifie que les valeurs dans `.git/config` écrasent celles dans `/etc/gitconfig`.

> Ces fichiers de configuration Git sont des fichiers texte, donc vous pouvez positionner ces valeurs manuellement en éditant le fichier et en utilisant la syntaxe correcte, mais il reste généralement plus facile de lancer la commande `git config`.

Configuration de base d'un client

Les options de configuration reconnues par Git tombent dans deux catégories : côté client et côté serveur. La grande majorité se situe côté client pour coller à vos préférences personnelles de travail. Parmi les tonnes d'options disponibles, seules les plus communes ou affectant significativement la manière de travailler seront couvertes. De nombreuses options ne s'avèrent utiles qu'en de rares cas et ne seront pas traitées. Pour voir la liste de toutes les options que votre version de Git reconnaît, vous pouvez lancer :

```
$ man git-config
```

Cette commande affiche toutes les options disponibles avec quelques détails. Vous pouvez aussi trouver des informations de référence sur ***http://git-scm.com/docs/git-config.html***.

`CORE.EDITOR`

Par défaut, Git utilise votre éditeur par défaut (`$VISUAL` ou `$EDITOR`) ou se replie sur l'éditeur Vi pour la création et l'édition des messages de validation et d'étiquetage. Pour modifier ce programme par défaut pour un autre, vous pouvez utiliser le paramètre `core.editor` :

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

Maintenant, quel que soit votre éditeur par défaut, Git démarrera Emacs pour éditer les messages.

`COMMIT.TEMPLATE`

Si vous réglez ceci sur le chemin d'un fichier sur votre système, Git utilisera ce fichier comme message par défaut quand vous validez. Par exemple, supposons que vous créiez un fichier modèle dans `$HOME/.gitmessage.txt` qui ressemble à ceci :

```
ligne de sujet

description

[ticket: X]
```

Pour indiquer à Git de l'utiliser pour le message par défaut qui apparaîtra dans votre éditeur quand vous lancerez `git commit`, réglez le paramètre de configuration `commit.template` :

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Ainsi, votre éditeur ouvrira quelque chose ressemblant à ceci comme modèle de message de validation :

```
ligne de sujet

description

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Si vous avez une règle de messages de validation, placez un modèle de cette règle sur votre système et configurez Git pour qu'il l'utilise par défaut, cela améliorera les chances que cette règle soit effectivement suivie.

CORE.PAGER

Le paramètre `core.pager` détermine quel *pager* est utilisé lorsque des pages de Git sont émises, par exemple lors d'un `log` ou d'un `diff`. Vous pouvez le fixer à `more` ou à votre *pager* favori (par défaut, il vaut `less`) ou vous pouvez le désactiver en fixant sa valeur à une chaîne vide :

```
$ git config --global core.pager ''
```

Si vous lancez cela, Git affichera la totalité du résultat de toutes les commandes d'une traite, quelle que soit sa longueur.

USER.SIGNINGKEY

Si vous faites des étiquettes annotées signées (comme décrit dans **"Signer votre travail"**), simplifiez-vous la vie en définissant votre clé GPG de signature en paramètre de configuration. Définissez votre ID de clé ainsi :

```
$ git config --global user.signingkey <gpg-key-id>
```

Maintenant, vous pouvez signer vos étiquettes sans devoir spécifier votre clé à chaque fois que vous utilisez la commande `git tag` :

```
$ git tag -s <nom-étiquette>
```

CORE.EXCLUDESFILE

Comme décrit dans **"Ignorer des fichiers"**, vous pouvez ajouter des patrons dans le fichier `.gitignore` de votre projet pour indiquer à Git de ne pas considérer certains fichiers comme non suivis ou pour éviter de les indexer lorsque vous lancez `git add` sur eux.

Mais vous pouvez souhaiter dans quelques cas ignorer certains fichiers dans tous vos dépôts. Si votre ordinateur utilise Mac OS X, vous connaissez certainement les fichiers `.DS_Store`. Si votre éditeur préféré est Emacs ou Vim, vous connaissez sûrement aussi les fichiers qui se terminent par ~.

Cette option vous permet d'écrire un fichier `.gitignore` global. Si vous créez un fichier `~/.gitignore_global` contenant ceci :

```
*~
.DS_Store
```

et que vous lancez `git config --global core.excludesfile ~/.gitignore_global`, Git ne vous importunera plus avec ces fichiers.

HELP.AUTOCORRECT

Si vous avez fait une faute de frappe en tapant une commande Git, il vous affiche quelque chose comme :

```
$ git chekcout master
git : 'chekcout' n'est pas une commande git. Voir 'git --help'.

Vouliez-vous dire cela ?
        checkout
```

Git essaie de deviner ce que vous avez voulu dire, mais continue de refuser de le faire. Si vous positionnez le paramètre `help.autocorrect` à 1, Git va réellement lancer cette commande à votre place :

```
$ git chekcout master
ATTENTION : vous avez invoqué une commande Git nommée 'chekcout' qui n'existe pas.
Continuons en supposant que vous avez voulu dire 'checkout'
dans 0.1 secondes automatiquement...
```

Notez l'histoire des « 0.1 secondes ». `help.autocorrect` est un fait un entier qui représente des dixièmes de seconde. Ainsi, si vous le réglez à 50, Git vous laissera 5 secondes pour changer d'avis avant de lancer la commande qu'il aura devinée.

Couleurs dans Git

Git sait coloriser ses affichages dans votre terminal, ce qui peut faciliter le parcours visuel des résultats. Un certain nombre d'options peuvent vous aider à régler la colorisation à votre goût.

`COLOR.UI`

Git colorise automatiquement la plupart de ses affichages mais il existe une option globale pour désactiver ce comportement. Pour désactiver toute la colorisation par défaut, lancez ceci :

```
$ git config --global color.ui false
```

La valeur par défaut est `auto`, ce qui colorise la sortie lorsque celle-ci est destinée à un terminal, mais élimine les codes de contrôle de couleur quand la sortie est redirigée dans un fichier ou l'entrée d'une autre commande.

Vous pouvez aussi la régler à `always` (toujours) pour activer la colorisation en permanence. C'est une option rarement utile. Dans la plupart des cas, si vous tenez vraiment à coloriser vos sorties redirigées, vous pourrez passer le drapeau `--color` à la commande Git pour la forcer à utiliser les codes de couleur. Le réglage par défaut est donc le plus utilisé.

`COLOR.*`

Si vous souhaitez être plus spécifique concernant les commandes colorisées, Git propose des paramètres de colorisation par action. Chacun peut être fixé à `true`, `false` ou `always`.

```
color.branch
color.diff
color.interactive
color.status
```

De plus, chacun d'entre eux dispose d'un sous-ensemble de paramètres qui permettent de surcharger les couleurs pour des parties des affichages. Par exemple, pour régler les couleurs de méta-informations du diff avec une écriture en bleu gras (*bold* en anglais) sur fond noir :

```
$ git config --global color.diff.meta "blue black bold"
```

La couleur peut prendre les valeurs suivantes : *normal*, *black*, *red*, *green*, *yellow*, *blue*, *magenta*, *cyan* ou *white*. Si vous souhaitez ajouter un attribut de casse, les valeurs disponibles sont *bold* (gras), *dim* (léger), *ul* (*underlined*, souligné), *blink* (clignotant) et *reverse* (inversé).

Outils externes de fusion et de différence

Bien que Git ait une implémentation interne de diff que vous avez déjà utilisée, vous pouvez sélectionner à la place un outil externe. Vous pouvez aussi sélectionner un outil graphique pour la fusion et la résolution de conflit au lieu de devoir résoudre les conflits manuellement. Je démontrerai le paramétrage avec Perforce Merge Tool (P4Merge) pour visualiser vos différences et résoudre vos fusions parce que c'est un outil graphique agréable et gratuit.

Si vous voulez l'essayer, P4Merge fonctionne sur tous les principaux systèmes d'exploitation. Dans cet exemple, je vais utiliser la forme des chemins usitée sur Mac et Linux. Pour Windows, vous devrez changer `/usr/local/bin` en un chemin d'exécution d'un programme de votre environnement.

Pour commencer, téléchargez P4Merge depuis ***http://www.perforce.com/downloads/Perforce/***. Ensuite, il faudra mettre en place un script d'enrobage pour lancer les commandes. Je vais utiliser le chemin Mac pour l'exécutable ; dans d'autres systèmes, il résidera où votre binaire `p4merge` a été installé. Créez un script enveloppe nommé `extMerge` qui appelle votre binaire avec tous les arguments fournis :

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

L'enveloppe diff s'assure que sept arguments ont été fournis et en passe deux à votre script de fusion. Par défaut, Git passe au programme de diff les arguments suivants :

```
chemin ancien-fichier ancien-hex ancien-mode nouveau-fichier nouveau-hex nouveau-mode
```

Comme seuls les arguments `ancien-fichier` et `nouveau-fichier` sont nécessaires, vous utilisez le script d'enveloppe pour passer ceux dont vous avez besoin.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Vous devez aussi vous assurer que ces fichiers sont exécutables :

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

À présent, vous pouvez régler votre fichier de configuration pour utiliser vos outils personnalisés de résolution de fusion et de différence. Pour cela, il faut un certain nombre de personnalisations : `merge.tool` pour indiquer à Git quelle stratégie utiliser, `mergetool.<tool>.cmd` pour spécifier comment lancer cette commande, `mergetool.<tool>.trustExitCode` pour indiquer à Git si le code de sortie du programme indique une résolution de fusion réussie ou non et `diff.external` pour indiquer à Git quelle commande lancer pour les différences. Ainsi, vous pouvez lancer les quatre commandes :

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

ou vous pouvez éditer votre fichier `~/.gitconfig` pour y ajouter ces lignes :

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Après avoir réglé tout ceci, si vous lancez des commandes de diff telles que celle-ci :

```
$ git diff 32d1776b1^ 32d1776b1
```

Au lieu d'obtenir la sortie du diff dans le terminal, Git lance P4Merge, ce qui ressemble à ceci :

FIGURE 8-1

P4Merge.

Si vous essayez de fusionner deux branches et créez des conflits de fusion, vous pouvez lancer la commande `git mergetool` qui démarrera P4Merge pour vous laisser résoudre les conflits au moyen d'un outil graphique.

Le point agréable avec cette méthode d'enveloppe est que vous pouvez changer facilement d'outils de diff et de fusion. Par exemple, pour changer vos outils `extDiff` et `extMerge` pour une utilisation de l'outil KDiff3, il vous suffit d'éditer le fichier `extMerge` :

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

À présent, Git va utiliser l'outil KDiff3 pour visualiser les différences et résoudre les conflits de fusion.

Git est livré préréglé avec un certain nombre d'autres outils de résolution de fusion pour vous éviter d'avoir à gérer la configuration `cmd`. Pour obtenir une liste des outils qu'il supporte, essayez ceci :

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
```

```
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

Si KDiff3 ne vous intéresse pas pour gérer les différences mais seulement pour la résolution de fusion et qu'il est présent dans votre chemin d'exécution, vous pouvez lancer :

```
$ git config --global merge.tool kdiff3
```

Si vous lancez ceci au lieu de modifier les fichiers `extMerge` ou `extDiff`, Git utilisera KDiff3 pour les résolutions de fusion et l'outil diff normal de Git pour les différences.

Formatage et espaces blancs

Les problèmes de formatage et de blancs sont parmi les plus subtils et frustrants que les développeurs rencontrent lorsqu'ils collaborent, spécifiquement d'une plate-forme à l'autre. Il est très facile d'introduire des modifications subtiles de blancs lors de soumission de patchs ou d'autres modes de collaboration, car les éditeurs de texte les insèrent silencieusement ou les programmeurs Windows ajoutent des retours chariot à la fin des lignes qu'ils modifient. Git dispose de quelques options de configuration pour traiter ces problèmes.

`CORE.AUTOCRLF`

Si vous programmez vous-même sous Windows ou si vous utilisez un autre système d'exploitation mais devez travailler avec des personnes travaillant sous Windows, vous rencontrerez à un moment ou à un autre des problèmes de caractères de fin de ligne. Ceci est dû au fait que Windows utilise pour marquer les fins de ligne dans ses fichiers un caractère « retour chariot » (*carriage return*, CR) suivi d'un caractère « saut de ligne » (*line feed*, LF), tandis que Mac et Linux utilisent seulement le caractère « saut de ligne ». C'est un cas subtil mais incroyablement ennuyeux de problème généré par la collaboration inter plateforme.

Git peut gérer ce cas en convertissant automatiquement les fins de ligne CRLF en LF lorsque vous validez, et inversement lorsqu'il extrait des fichiers sur votre système. Vous pouvez activer cette fonctionnalité au moyen du paramètre `core.autocrlf`. Si vous avez une machine Windows, positionnez-le à `true`. Git convertira les fins de ligne de LF en CRLF lorsque vous extrairez votre code :

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

Si vous utilisez un système Linux ou Mac qui utilise les fins de ligne LF, vous ne souhaitez sûrement pas que Git les convertisse automatiquement lorsque vous extrayez des fichiers. Cependant, si un fichier contenant des CRLF est accidentellement introduit en version, vous souhaitez que Git le corrige. Vous pouvez indiquer à Git de convertir CRLF en LF lors de la validation mais pas dans l'autre sens en fixant `core.autocrlf` à `input` :

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

Ce réglage devrait donner des fins de ligne en CRLF lors d'extraction sous Windows mais en LF sous Mac et Linux et dans le dépôt.

Si vous êtes un programmeur Windows gérant un projet spécifique à Windows, vous pouvez désactiver cette fonctionnalité et forcer l'enregistrement des « retour chariot » dans le dépôt en réglant la valeur du paramètre à `false` :

```
$ git config --global core.autocrlf false
```

`CORE.WHITESPACE`

Git est paramétré par défaut pour détecter et corriger certains problèmes de blancs. Il peut rechercher six problèmes de blancs de base. La correction de trois problèmes est activée par défaut et peut être désactivée et celle des trois autres n'est pas activée par défaut mais peut être activée.

Les trois activées par défaut sont `blank-at-eol` qui détecte les espaces en fin de ligne, `blank-at-eof` qui détecte les espaces en fin de fichier et `space-before-tab` qui recherche les espaces avant les tabulations au début d'une ligne.

Les trois autres qui sont désactivées par défaut mais peuvent être activées sont `indent-with-non-tab` qui recherche des lignes qui commencent par des espaces au lieu de tabulations (contrôlé par l'option `tabwidth`), `tab-in-indent` qui recherche les tabulations dans la portion d'indentation d'une ligne et `cr-at-eol` qui indique à Git que les « retour chariot » en fin de ligne sont acceptés.

Vous pouvez indiquer à Git quelle correction vous voulez activer en fixant `core.whitespace` avec les valeurs que vous voulez ou non, séparées par des virgules. Vous pouvez désactiver des réglages en les éliminant de la chaîne de paramétrage ou en les préfixant avec un `-`. Par exemple, si vous souhaitez activer tout sauf `cr-at-eol`, vous pouvez lancer ceci :

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git va détecter ces problèmes quand vous lancez une commande `git diff` et essayer de les coloriser pour vous permettre de les régler avant de valider.

Il utilisera aussi ces paramètres pour vous aider quand vous appliquerez des patchs avec `git apply`.

Quand vous appliquez des patchs, vous pouvez paramétrer Git pour qu'il vous avertisse s'il doit appliquer des patchs qui présentent les défauts de blancs :

```
$ git apply --whitespace=warn <patch>
```

Ou vous pouvez indiquer à Git d'essayer de corriger automatiquement le problème avant d'appliquer le patch :

```
$ git apply --whitespace=fix <patch>
```

Ces options s'appliquent aussi à `git rebase`. Si vous avez validé avec des problèmes de blancs mais n'avez pas encore poussé en amont, vous pouvez lancer un `rebase` avec l'option `--whitespace=fix` pour faire corriger à Git les erreurs de blancs pendant qu'il réécrit les patchs.

Configuration du serveur

Il n'y a pas autant d'options de configuration de Git côté serveur, mais en voici quelques unes intéressantes dont il est utile de prendre note.

`RECEIVE.FSCKOBJECTS`

Git est capable de vérifier que tous les objets reçus pendant une poussée correspondent à leur somme de contrôle SHA-1 et qu'ils pointent sur des objets valides. Cependant, il ne le fait pas par défaut sur chaque poussée. C'est une opération relativement lourde qui peut énormément allonger les poussées selon la taille du dépôt ou de la poussée. Si vous voulez que Git vérifie la cohérence des objets à chaque poussée, vous pouvez le forcer en fixant le paramètre `receive.fsckObjects` à `true` :

```
$ git config --system receive.fsckObjects true
```

Maintenant, Git va vérifier l'intégrité de votre dépôt avant que chaque poussée ne soit acceptée pour s'assurer que des clients défectueux (ou malicieux) n'introduisent pas des données corrompues.

`RECEIVE.DENYNONFASTFORWARDS`

Si vous rebasez des *commits* que vous avez déjà poussés, puis essayez de pousser à nouveau, ou inversement, si vous essayez de pousser un *commit* sur une branche distante qui ne contient pas le *commit* sur lequel la branche distante pointe, votre essai échouera. C'est généralement une bonne politique, mais dans le cas d'un rebasage, vous pouvez décider que vous savez ce que vous faites et forcer la mise à jour de la branche distante en ajoutant l'option `-f` à votre commande.

Pour désactiver la possibilité de forcer la mise à jour des branches distantes autres qu'en avance rapide, réglez `receive.denyNonFastForwards` :

```
$ git config --system receive.denyNonFastForwards true
```

Un autre moyen de faire consiste à utiliser des crochets côté-serveur, point qui sera abordé plus loin. Cette autre approche permet de réaliser des traitements plus complexes comme de refuser l'avance rapide seulement à un certain groupe d'utilisateurs.

`RECEIVE.DENYDELETES`

Un des contournements possible à la politique `denyNonFastForwards` consiste à simplement effacer la branche distante et à la repousser avec les nouvelles références. Pour interdire ceci, réglez `receive.denyDeletes` à `true` :

```
$ git config --system receive.denyDeletes true
```

Ceci interdit la suppression de branches ou d'étiquettes. Aucun utilisateur n'en a le droit. Pour pouvoir effacer des branches distantes, vous devez effacer manuellement les fichiers de référence sur le serveur. Il existe aussi des moyens plus intéressants de gérer cette politique utilisateur par utilisateur au moyen des listes de contrôle d'accès, point qui sera abordé dans **"Exemple de politique gérée par Git"**.

Attributs Git

Certains de ces réglages peuvent aussi s'appliquer sur un chemin, de telle sorte que Git ne les applique que sur un sous-répertoire ou un sous-ensemble de fichiers. Ces réglages par chemin sont appelés attributs Git et sont définis soit dans un fichier `.gitattributes` dans un répertoire (normalement la racine du projet), soit dans un fichier `.git/info/attributes` si vous ne souhaitez pas que le fichier de description des attributs fasse partie du projet.

Les attributs permettent de spécifier des stratégies de fusion différentes pour certains fichiers ou répertoires dans votre projet, d'indiquer à Git la manière de calculer les différences pour certains fichiers non-texte, ou de faire filtrer à Git le contenu avant qu'il ne soit validé ou extrait. Dans ce chapitre, nous traiterons certains attributs applicables aux chemins et détaillerons quelques exemples de leur utilisation en pratique.

Fichiers binaires

Les attributs Git permettent des trucs cool comme d'indiquer à Git quels fichiers sont binaires (dans les cas où il ne pourrait pas le deviner par lui-même) et de lui donner les instructions spécifiques pour les traiter. Par exemple, certains

fichiers peuvent être générés par machine et impossible à traiter par diff, tandis que pour certains autres fichiers binaires, les différences peuvent être calculées. Nous détaillerons comment indiquer à Git l'un et l'autre.

IDENTIFICATION DES FICHIERS BINAIRES

Certains fichiers ressemblent à des fichiers texte mais doivent en tout état de cause être traités comme des fichiers binaires. Par exemple, les projets Xcode sous Mac contiennent un fichier finissant en `.pbxproj`, qui est en fait un jeu de données JSON (format de données en texte JavaScript) enregistré par l'application EDI pour y sauver les réglages entre autres de compilation. Bien que ce soit techniquement un fichier texte en ASCII, il n'y a aucun intérêt à le gérer comme tel parce que c'est en fait une mini base de données. Il est impossible de fusionner les contenus si deux utilisateurs le modifient et les calculs de différence par défaut sont inutiles. Ce fichier n'est destiné qu'à être manipulé par un programme. En résumé, ce fichier doit être considéré comme un fichier binaire opaque.

Pour indiquer à Git de traiter tous les fichiers `pbxproj` comme binaires, ajoutez la ligne suivante à votre fichier `.gitattributes` :

```
*.pbxproj binary
```

À présent, Git n'essaiera pas de convertir ou de corriger les problèmes des CRLF, ni de calculer ou d'afficher les différences pour ces fichiers quand vous lancez `git show` ou `git diff` sur votre projet.

COMPARAISON DE FICHIERS BINAIRES

Dans Git, vous pouvez utiliser la fonctionnalité des attributs pour comparer efficacement les fichiers binaires. Pour ce faire, indiquez à Git comment convertir vos données binaires en format texte qui peut être comparé via un diff normal.

Comme c'est une fonctionnalité vraiment utile et peu connue, nous allons détailler certains exemples. Premièrement, nous utiliserons cette technique pour résoudre un des problèmes les plus ennuyeux de l'humanité : gérer en contrôle de version les documents Word. Tout le monde convient que Word est l'éditeur de texte le plus horrible qui existe, mais bizarrement, tout le monde persiste à l'utiliser. Si vous voulez gérer en version des documents Word, vous pouvez les coller dans un dépôt Git et les valider de temps à autre. Mais qu'est-ce que ça vous apporte ? Si vous lancez `git diff` normalement, vous verrez quelque chose comme :

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Vous ne pouvez pas comparer directement les versions à moins de les extraire et de les parcourir manuellement. En fait, vous pouvez faire la même chose plutôt bien en utilisant les attributs Git. Ajoutez la ligne suivante dans votre fichier `.gitattributes` :

```
*.docx diff=word
```

Cette ligne indique à Git que tout fichier correspondant au patron (`.docx`) doit utiliser le filtre `word` pour visualiser le diff des modifications. Qu'est-ce que le filtre « word » ? Nous devons le définir. Vous allez indiquer à Git d'utiliser le programme `docx2txt` qui a été écrit spécifiquement pour extraire le texte d'un document MS Word, qu'il pourra comparer correctement.

Installez déjà `docx2text`. Vous pouvez le télécharger depuis ***http://docx2txt.sourceforge.net***. Suivez les instruction dans le fichier `INSTALL` pour le placer à un endroit où votre shell peut le trouver. Ensuite, écrivons un script qui convertit la sortie dans le format que Git comprend. Créez un fichier dans votre chemin d'exécution appelé `docx2txt` et ajoutez ce contenu :

```
#!/bin/bash
docx2txt.pl $1 -
```

N'oubliez pas de faire un `chmod a+x` sur ce fichier. Finalement, vous pouvez configurer Git pour qu'il utilise ce script :

```
$ git config diff.word.textconv docx2txt
```

À présent, Git sait que s'il essaie de faire un diff entre deux instantanés et qu'un des fichiers finit en `.docx`, il devrait faire passer ces fichiers par le filtre `word` défini comme le programme `docx2txt`. Cette méthode fait effectivement des jolies versions texte de vos fichiers Word avant d'essayer de les comparer.

Voici un exemple. J'ai mis le chapitre 1 de ce livre dans Git, ajouté du texte à un paragraphe et sauvegardé le document. Puis, j'ai lancé `git diff` pour visualiser ce qui a changé :

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning by expl
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that recorc
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or layout (
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another directory (per
```

Git m'indique succinctement que j'ai ajouté la chaîne « Testing: 1, 2, 3. », ce qui est correct. Ce n'est pas parfait – les modifications de formatage n'apparaissent pas – mais c'est efficace.

Un autre problème intéressant concerne la comparaison de fichiers d'images. Une méthode consiste à faire passer les fichiers image à travers un filtre qui extrait les données EXIF, les méta-données enregistrées avec la plupart des formats d'image. Si vous téléchargez et installez le programme `exiftool`, vous pouvez l'utiliser pour convertir vos images en texte de méta-données de manière que le diff puisse au moins montrer une représentation textuelle des modifications pratiquées. Mettez la ligne suivante dans votre fichier `.gitattributes` :

```
*.png diff=exif
```

Configurez Git pour utiliser cet outil :

```
$ git config diff.exif.textconv exiftool
```

Si vous remplacez une image dans votre projet et lancez `git diff`, vous verrez ceci :

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
```

```
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

Vous pouvez réaliser rapidement que la taille du fichier et les dimensions des images ont changé.

Expansion des mots-clés

L'expansion de mots-clés dans le style de CVS ou de SVN est souvent une fonctionnalité demandée par les développeurs qui y sont habitués. Le problème principal de ce système avec Git est que vous ne pouvez pas modifier un fichier avec l'information concernant le *commit* après la validation parce que Git calcule justement la somme de contrôle sur son contenu. Cependant, vous pouvez injecter des informations textuelles dans un fichier au moment où il est extrait et les retirer avant qu'il ne soit ajouté à un *commit*. Les attributs Git vous fournissent deux manières de le faire.

Premièrement, vous pouvez injecter automatiquement la somme de contrôle SHA-1 d'un blob dans un champ `$Id$` d'un fichier. Si vous positionnez cet attribut pour un fichier ou un ensemble de fichiers, la prochaine fois que vous extrairez cette branche, Git remplacera chaque champ avec le SHA-1 du blob. Il est à noter que ce n'est pas le SHA du *commit* mais celui du blob lui-même.

Mettez la ligne suivante dans votre fichier `.gitattributes` :

```
*.txt ident
```

Ajoutez une référence `$Id$` à un fichier de test :

```
$ echo '$Id$' > test.txt
```

À la prochaine extraction de ce fichier, Git injecte le SHA du blob :

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Néanmoins, ce résultat n'a que peu d'intérêt. Si vous avez utilisé la substitution avec CVS ou Subversion, il est possible d'inclure la date. Le code SHA n'est pas des plus utiles car il ressemble à une valeur aléatoire et ne vous permet pas de distinguer si tel SHA est plus récent ou plus ancien que tel autre.

Il apparaît que vous pouvez écrire vos propres filtres pour réaliser des substitutions dans les fichiers lors des validations/extractions. Ces filtres s'appellent « *clean* » et « *smudge* ». Dans le fichier `.gitattributes`, vous pouvez indiquer un filtre pour des chemins particuliers puis créer des scripts qui traiteront ces fichiers avant qu'ils soient extraits (« *smudge* », voir **Figure 8-2**) et juste avant qu'ils soient validés (« *clean* », voir **Figure 8-3**). Ces filtres peuvent servir à faire toutes sortes de choses attrayantes.

FIGURE 8-2

Le filtre « smudge » est lancé lors d'une extraction.

FIGURE 8-3

Le filtre « clean » est lancé lorsque les fichiers sont indexés.

Le message de validation d'origine pour cette fonctionnalité donne un exemple simple permettant de passer tout votre code C par le programme `indent` avant de valider. Vous pouvez le faire en réglant l'attribut `filter` dans votre fichier `.gitattributes` pour filtrer les fichiers `*.c` avec le filtre « indent » :

```
*.c filter=indent
```

Ensuite, indiquez à Git ce que le filtre « indent » fait sur *smudge* et *clean*

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Dans ce cas, quand vous validez des fichiers qui correspondent à `*.c`, Git les fera passer par le programme `indent` avant de les valider et les fera passer par le programme `cat` avant de les extraire sur votre disque. Le programme `cat` ne fait rien : il se contente de régurgiter les données telles qu'il les a lues. Cette combinaison filtre effectivement tous les fichiers de code source C par `indent` avant leur validation.

Un autre exemple intéressant fournit l'expansion du mot-clé `$Date$` dans le style RCS. Pour le réaliser correctement, vous avez besoin d'un petit script qui prend un nom de fichier, calcule la date de la dernière validation pour le projet et l'insère dans le fichier. Voici un petit script Ruby qui le fait :

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Tout ce que le script fait, c'est récupérer la date de la dernière validation à partir de la commande `git log`, la coller dans toutes les chaînes `$Date$` qu'il trouve et réécrire le résultat. Ce devrait être simple dans n'importe quel langage avec lequel vous êtes à l'aise. Appelez ce fichier `expand_date` et placez-le dans votre chemin. À présent, il faut paramétrer un filtre dans Git (appelons le `dater`) et lui indiquer d'utiliser le filtre `expand_date` en tant que *smudge* sur les fichiers à extraire. Nous utiliserons une expression Perl pour nettoyer lors d'une validation :

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\$Date[^\\$]*\\$/\\$Date\\$/"'
```

Cette commande Perl extrait tout ce qu'elle trouve dans une chaîne `$Date$` et la réinitialise. Le filtre prêt, on peut le tester en écrivant le mot-clé `$Date$` dans un fichier, puis en créant un attribut Git pour ce fichier qui fait référence au nouveau filtre et en créant un fichier avec votre mot-clé `$Date$` :

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Si vous validez ces modifications et extrayez le fichier à nouveau, vous remarquez le mot-clé correctement substitué :

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Vous pouvez voir à quel point cette technique peut être puissante pour des applications personnalisées. Il faut rester néanmoins vigilant car le fichier `.gitattributes` est validé et inclus dans le projet tandis que le gestionnaire (ici, `dater`) ne l'est pas. Du coup, ça ne marchera pas partout. Lorsque vous créez ces filtres, ils devraient pouvoir avoir un mode dégradé qui n'empêche pas le projet de fonctionner.

Export d'un dépôt

Les données d'attribut Git permettent aussi de faire des choses intéressantes quand vous exportez une archive du projet.

`EXPORT-IGNORE`

Vous pouvez dire à Git de ne pas exporter certains fichiers ou répertoires lors de la génération d'archive. S'il y a un sous-répertoire ou un fichier que vous ne souhaitez pas inclure dans le fichier archive mais que vous souhaitez extraire dans votre projet, vous pouvez indiquer ce fichier via l'attribut `export-ignore`.

Par exemple, disons que vous avez des fichiers de test dans le sous-répertoire `test/` et que ce n'est pas raisonnable de les inclure dans l'archive

d'export de votre projet. Vous pouvez ajouter la ligne suivante dans votre fichier d'attribut Git :

```
test/ export-ignore
```

À présent, quand vous lancez `git archive` pour créer une archive `tar` de votre projet, ce répertoire ne sera plus inclus dans l'archive.

EXPORT-SUBST

Quand vous exportez des fichiers pour un déploiement, vous pouvez appliquer le formatage de `git log` et l'expansion de mot-clés à des portions choisies de fichiers marquées avec l'attribut `export-subst`.

Par exemple, si vous voulez inclure un fichier appelé `LAST_COMMIT` dans votre projet et y injecter automatiquement la date de dernière validation lorsque `git archive` est lancé, vous pouvez définir vos fichiers `.gitattributes` et `LAST_COMMIT` comme ceci :

```
LAST_COMMIT export-subs
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Quand vous lancez `git archive`, le contenu de ce fichier inclus dans l'archive ressemblera à ceci :

```
$ git archive HEAD | tar xCf ../test-deploiement -
$ cat ../test-deploiement/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

Les substitutions peuvent inclure par exemple le message de validation et n'importe quelle note git, et git log peut faire du simple retour à la ligne :

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log'\''s custom formatter

git archive uses git log'\''s `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
'
$ git archive @ | tar xfO - LAST_COMMIT
```

```
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst uses git log's custom formatter

         git archive uses git log's `pretty=format:` processor directly, and
         strips the surrounding `$Format:` and `$` markup from the output.
```

L'archive résultante est appropriée pour le travail de déploiement, mais comme n'importe quelle archive exportée, elle n'est pas appropriée pour continuer un travail de développement.

Stratégies de fusion

Vous pouvez aussi utiliser les attributs Git pour indiquer à Git d'utiliser des stratégies de fusion différenciées pour des fichiers spécifiques dans votre projet. Une option très utile est d'indiquer à Git de ne pas essayer de fusionner des fichiers spécifiques quand ils rencontrent des conflits mais plutôt d'utiliser prioritairement votre version du fichier.

C'est très utile si une branche de votre projet a divergé ou s'est spécialisée, mais que vous souhaitez pouvoir fusionner les modifications qu'elle porte et vous voulez ignorer certains fichiers. Supposons que vous avez un fichier de paramètres de base de données appelé `database.xml` différent sur deux branches et vous voulez les fusionner dans votre autre branche sans corrompre le fichier de base de données. Vous pouvez déclarer un attribut comme ceci :

```
database.xml merge=ours
```

Et définir une bête stratégie de fusion `ours` avec :

```
$ git config --global merge.ours.driver true
```

Si vous fusionnez dans une autre branche, plutôt que de rencontrer des conflits de fusion avec le fichier `database.xml`, vous verrez quelque chose comme :

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

Dans ce cas, `database.xml` reste dans l'état d'origine, quoi qu'il arrive.

Crochets Git

Comme de nombreux autres systèmes de gestion de version, Git dispose d'un moyen de lancer des scripts personnalisés quand certaines actions importantes ont lieu. Il y a deux groupes de crochets : ceux côté client et ceux côté serveur. Les crochets côté client concernent les opérations de client telles que la validation et la fusion. Les crochets côté serveur concernent les opérations de serveur Git telles que la réception de *commits*. Vous pouvez utiliser ces crochets pour toutes sortes de fonctions.

Installation d'un crochet

Les crochets sont tous stockés dans le sous-répertoire `hooks` du répertoire Git. Dans la plupart des projets, c'est `.git/hooks`.

Par défaut, Git popule ce répertoire avec quelques scripts d'exemple déjà utiles par eux-mêmes ; mais ils servent aussi de documentation sur les paramètres de chaque script. Tous les exemples sont des scripts shell avec un peu de Perl mais n'importe quel script exécutable nommé correctement fonctionnera. Vous pouvez les écrire en Ruby ou Python ou ce que vous voudrez. Pour les versions de Git postérieures à 1.6, ces fichiers crochet d'exemple se terminent en `.sample` et il faudra les renommer. Pour les versions de Git antérieures à 1.6, les fichiers d'exemple sont nommés correctement mais ne sont pas exécutables.

Pour activer un script de crochet, placez un fichier dans le sous-répertoire `hook` de votre répertoire Git, nommé correctement et exécutable. À partir de ce moment, il devrait être appelé. Abordons donc les noms de fichiers de crochet les plus importants.

Crochets côté client

Il y a de nombreux crochets côté client. Ce chapitre les classe entre crochets de traitement de validation, scripts de traitement par courriel et le reste des scripts côté client.

> Il est important de noter que les crochets côté client *ne sont pas* copiés quand le dépôt est cloné. Si vous avez l'intention d'utiliser ces scripts pour faire respecter une politique de validation, il vaut mieux utiliser des crochets côté serveur, comme **"Exemple de politique gérée par Git"**.

CROCHETS DE TRAITEMENT DE VALIDATION

Les quatre premiers crochets ont trait au processus de validation.

Le crochet `pre-commit` est lancé en premier, avant même que vous ne saisissiez le message de validation. Il est utilisé pour inspecter l'instantané qui est sur le point d'être validé, pour vérifier si vous avez oublié quelque chose, pour s'assurer que les tests passent ou pour examiner ce que vous souhaitez inspecter dans le code. Un code de sortie non nul de ce crochet annule la validation, bien que vous puissiez le contourner avec `git commit --no-verify`. Vous pouvez réaliser des actions telles qu'une vérification de style (en utilisant lint ou un équivalent), d'absence de blancs en fin de ligne (le crochet par défaut fait exactement cela) ou de documentation des nouvelles méthodes.

Le crochet `prepare-commit-msg` est appelé avant que l'éditeur de message de validation ne soit lancé mais après que le message par défaut a été créé. Il vous permet d'éditer le message par défaut avant que l'auteur ne le voit. Ce crochet accepte quelques options : le chemin du fichier qui contient le message de validation actuel, le type de validation et le SHA-1 du *commit* si c'est un *commit* amendé. Ce crochet ne sert généralement à rien pour les validations normales. Par contre, il est utile pour les validations où le message par défaut est généré, tel que les modèles de message de validation, les validations de fusion, les *commits* écrasés ou amendés. Vous pouvez l'utiliser en conjonction avec un modèle de messages pour insérer de l'information par programme.

Le crochet `commit-msg` accepte un paramètre qui est encore le chemin du fichier temporaire qui contient le message de validation actuel. Si ce script sort avec un code de sortie non nul, Git abandonne le processus de validation, ce qui vous permet de vérifier l'état de votre projet ou du message de validation avant de laisser passer un *commit*. Dans la dernière section de ce chapitre, l'utilisation de ce crochet permettra de vérifier que le message de validation est conforme à un format obligatoire.

Après l'exécution du processus complet de validation, le crochet `post-commit` est appelé. Il n'accepte aucun argument mais vous pouvez facilement accéder au dernier *commit* grâce à `git log -1 HEAD`. Généralement, ce script sert à réaliser des notifications ou des choses similaires.

CROCHETS DE GESTION COURRIEL

Vous pouvez régler trois crochets côté client pour la gestion à base de courriel. Ils sont tous invoqués par la commande `git am`, donc si vous n'êtes pas habitué à utiliser cette commande dans votre mode de gestion, vous pouvez simplement passer la prochaine section. Si vous acceptez des patchs préparés par `git format-patch` par courriel, alors certains de ces crochets peuvent vous être très utiles.

Le premier crochet lancé est `applypatch-msg`. Il accepte un seul argument : le nom du fichier temporaire qui contient le message de validation proposé. Git abandonne le patch si ce script sort avec un code non nul. Vous pouvez l'utiliser pour vérifier que le message de validation est correctement formaté ou pour normaliser le message en l'éditant sur place par script.

Le crochet lancé ensuite lors de l'application de patchs via `git am` s'appelle `pre-applypatch`. Il n'accepte aucun argument et son nom est trompeur car il est lancé après que le patch a été appliqué, ce qui vous permet d'inspecter l'instantané avant de réaliser la validation. Vous pouvez lancer des tests ou inspecter l'arborescence active avec ce script. S'il manque quelque chose ou que les tests ne passent pas, un code de sortie non nul annule la commande `git am` sans valider le patch.

Le dernier crochet lancé pendant l'opération `git am` s'appelle `post-applypatch`. Vous pouvez l'utiliser pour notifier un groupe ou l'auteur du patch que vous venez de l'appliquer. Vous ne pouvez plus arrêter le processus de validation avec ce script.

AUTRES CROCHETS CÔTÉ CLIENT

Le crochet `pre-rebase` est invoqué avant que vous ne rebasiez et peut interrompre le processus s'il sort avec un code d'erreur non nul. Vous pouvez utiliser ce crochet pour empêcher de rebaser tout *commit* qui a déjà été poussé. C'est ce que fait le crochet d'exemple `pre-rebase` que Git installe, même s'il fait des hypothèses qui peuvent ne pas correspondre avec votre façon de faire.

Le crochet `post-rewrite` est lancé par les commandes qui remplacent les *commits*, comme `git commit --amend` et `git rebase` (mais pas par `git filter-branch`). Son seul argument est la commande qui a déclenché la réécriture, et il reçoit une liste de réécritures sur `stdin`. Ce crochet a beaucoup des mêmes utilisations que les crochets `post-checkout` et `post-merge`.

Après avoir effectué avec succès un `git checkout`, le crochet `post-checkout` est lancé. Vous pouvez l'utiliser pour paramétrer correctement votre environnement projet dans votre copie de travail. Cela peut signifier y déplacer des gros fichiers binaires que vous ne souhaitez pas voir en gestion de source, générer automatiquement la documentation ou quelque chose dans le genre.

Le crochet `post-merge` s'exécute à la suite d'une commande `merge` réussie. Vous pouvez l'utiliser pour restaurer certaines données non gérées par Git dans la copie de travail telles que les informations de permission. Ce crochet permet même de valider la présence de fichiers externes au contrôle de Git que vous pourriez souhaiter voir recopiés lorsque la copie de travail change.

Le crochet `pre-push` est lancé pendant un `git push`, après la mise à jour des références distantes mais avant le transfert des objets. Il reçoit le nom et

l'emplacement du dépôt distant en paramètre et une liste des références qui seront mises à jour sur `stdin`. Il peut servir à valider un ensemble de mises à jour de références avant que la poussée n'ait réellement lieu (la poussée est abandonnée sur un code de sortie non nul).

Git lance de temps à autre le ramasse-miettes au cours de son fonctionnement en invoquant `git gc --auto`. Le crochet `pre-auto-gc` est invoqué juste avant le démarrage du ramasse-miettes et peut être utilisé pour vous en notifier ou pour annuler le ramasse-miettes si le moment ne s'y prête pas.

Crochets côté serveur

En complément des crochets côté client, vous pouvez utiliser comme administrateur système quelques crochets côté serveur pour appliquer quasiment toutes les règles de votre projet. Ces scripts s'exécutent avant et après chaque poussée sur le serveur. Les crochets `pre` peuvent rendre un code d'erreur non nul à tout moment pour rejeter la poussée et afficher un message d'erreur au client. Vous pouvez mettre en place des règles aussi complexes que vous le souhaitez.

PRE-RECEIVE

Le premier script lancé lors de la gestion d'une poussée depuis un client est `pre-receive`. Il accepte une liste de références lues sur *stdin*. S'il sort avec un code d'erreur non nul, aucune n'est acceptée. Vous pouvez utiliser ce crochet pour réaliser des tests tels que s'assurer que toutes les références mises à jour le sont en avance rapide ou pour s'assurer que l'utilisateur dispose bien des droits de création, poussée, destruction ou de lecture des mises à jour pour tous les fichiers qu'il cherche à mettre à jour dans cette poussée.

UPDATE

Le script `update` est très similaire au script `pre-receive`, à la différence qu'il est lancé une fois par branche qui doit être modifiée lors de la poussée. Si la poussée s'applique à plusieurs branches, `pre-receive` n'est lancé qu'une fois, tandis qu'`update` est lancé une fois par branche impactée. Au lieu de lire à partir de stdin, ce script accepte trois arguments : le nom de la référence (branche), le SHA-1 du *commit* pointé par la référence avant la poussée et le SHA-1 que l'utilisateur est en train de pousser. Si le script `update` se termine avec un code d'erreur non nul, seule la référence est rejetée. Les autres références pourront être mises à jour.

`POST-RECEIVE`

Le crochet `post-receive` est lancé après l'exécution complète du processus et peut être utilisé pour mettre à jour d'autres services ou pour notifier des utilisateurs. Il accepte les mêmes données sur *stdin* que `pre-receive`. Il peut par exemple envoyer un courriel à une liste de diffusion, notifier un serveur d'intégration continue ou mettre à jour un système de suivi de tickets. Il peut aussi analyser les messages de validation à la recherche d'ordres de mise à jour de l'état des tickets. Ce script ne peut pas arrêter le processus de poussée mais le client n'est pas déconnecté tant qu'il n'a pas terminé. Il faut donc être prudent à ne pas essayer de lui faire réaliser des actions qui peuvent durer longtemps.

Exemple de politique gérée par Git

Dans ce chapitre, nous allons utiliser ce que nous venons d'apprendre pour installer une gestion Git qui vérifie la présence d'un format personnalisé de message de validation, n'autorise que les poussées en avance rapide et autorise seulement certains utilisateurs à modifier certains sous-répertoires dans un projet. Nous construirons des scripts client pour informer les développeurs que leurs poussées vont être rejetées et des scripts sur le serveur pour mettre effectivement en place ces règles.

Nous utilisons Ruby pour les écrire, d'abord par inertie intellectuelle, ensuite parce que ce langage de script s'apparente le plus à du pseudo-code. Ainsi, il devrait être simple de suivre grossièrement le code même sans connaître le langage Ruby. Cependant, tout langage peut être utilisé. Tous les scripts d'exemple distribués avec Git sont soit en Perl soit en Bash, ce qui donne de nombreux autres exemples de crochets dans ces langages.

Crochet côté serveur

Toutes les actions côté serveur seront contenues dans le fichier `update` dans le répertoire `hooks`. Le fichier `update` s'exécute une fois par branche poussée et accepte trois paramètres :

- la référence sur laquelle on pousse
- l'ancienne révision de la branche
- la nouvelle révision de la branche.

Vous pouvez aussi avoir accès à l'utilisateur qui pousse si la poussée est réalisée par SSH. Si vous avez permis à tout le monde de se connecter avec un utilisateur unique (comme « git ») avec une authentification à clé publique, il vous

faudra fournir à cet utilisateur une enveloppe de shell qui déterminera l'identité de l'utilisateur à partir de sa clé publique et positionnera une variable d'environnement spécifiant cette identité. Ici, nous considérons que la variable d'environnement $USER indique l'utilisateur connecté, donc le script update commence par rassembler toutes les informations nécessaires :

```
#!/usr/bin/env ruby

$nomref       = ARGV[0]
$anciennerev  = ARGV[1]
$nouvellerev  = ARGV[2]
$utilisateur  = ENV['USER']

puts "Vérification des règles..."
puts "(#{$nomref}) (#{$anciennerev[0,6]}) (#{$nouvellerev[0,6]})"
```

Oui, ce sont des variables globales. C'est seulement pour simplifier la démonstration.

APPLICATION D'UNE POLITIQUE DE FORMAT DU MESSAGE DE VALIDATION

Notre première tâche consiste à forcer que chaque message de validation adhère à un format particulier. En guise d'objectif, obligeons chaque message à contenir une chaîne de caractère qui ressemble à « ref: 1234 » parce que nous souhaitons que chaque validation soit liée à une tâche de notre système de tickets. Nous devons donc inspecter chaque *commit* poussé, vérifier la présence de la chaîne et sortir avec un code non-nul en cas d'absence pour rejeter la poussée.

Vous pouvez obtenir une liste des valeurs SHA-1 de tous les *commits* en cours de poussée en passant les valeurs `$nouvellerev` et `$anciennerev` à une commande de plomberie Git appelée `git rev-list`. C'est comme la commande `git log` mais elle n'affiche par défaut que les valeurs SHA-1, sans autre information. Donc, pour obtenir une liste de tous les SHA-1 des *commits* introduits entre un SHA de *commit* et un autre, il suffit de lancer quelque chose comme :

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Vous pouvez récupérer la sortie, boucler sur chacun de ces SHA-1 de *commit*, en extraire le message et le tester par rapport à une expression régulière qui cherche un patron.

Vous devez trouver comment extraire le message de validation à partir de chacun des *commits* à tester. Pour accéder aux données brutes du *commit*, vous pouvez utiliser une autre commande de plomberie appelée `git cat-file`. Nous traiterons en détail toutes ces commandes de plomberie dans **Chapter 10** mais pour l'instant, voici ce que cette commande affiche :

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Un moyen simple d'extraire le message de validation d'un *commit* à partir de son SHA-1 consiste à rechercher la première ligne vide et à sélectionner tout ce qui suit. Cela peut être facilement réalisé avec la commande `sed` sur les systèmes Unix :

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

Vous pouvez utiliser cette ligne pour récupérer le message de validation de chaque *commit* en cours de poussée et sortir si quelque chose ne correspond pas à ce qui est attendu. Pour sortir du script et rejeter la poussée, il faut sortir avec un code non nul. La fonction complète ressemble à ceci :

```
$regex = /\[ref: (\d+)\]/

# vérification du format des messages de validation
def verif_format_message
  revs_manquees = `git rev-list #{$anciennerev}..#{$nouvellerev}`.split("\n")
  revs_manquees.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[REGLE] Le message de validation n'est pas conforme"
      exit 1
    end
  end
end
verif_format_message
```

Placer ceci dans un script `update` rejettera les mises à jour contenant des *commits* dont les messages ne suivent pas la règle.

MISE EN PLACE D'UN SYSTÈME D'ACL PAR UTILISATEUR

Supposons que vous souhaitiez ajouter un mécanisme à base de liste de contrôle d'accès (access control list : ACL) qui permette de spécifier quel utilisateur a le droit de pousser des modifications vers quelle partie du projet. Certaines personnes ont un accès complet tandis que d'autres n'ont accès que pour mettre à jour certains sous-répertoires ou certains fichiers. Pour faire appliquer ceci, nous allons écrire ces règles dans un fichier appelé `acl` situé dans le dépôt brut Git sur le serveur. Le crochet `update` examinera ces règles, listera les fichiers impactés par la poussée et déterminera si l'utilisateur qui pousse a effectivement les droits nécessaires sur ces fichiers.

Écrivons en premier le fichier d'ACL. Nous allons utiliser un format très proche de celui des ACL de CVS. Le fichier est composé de lignes dont le premier champ est `avail` ou `unavail`, le second est une liste des utilisateurs concernés séparés par des virgules et le dernier champ indique le chemin pour lequel la règle s'applique (le champ vide indiquant une règle générale). Tous les champs sont délimités par un caractère pipe « | ».

Dans notre cas, il y a quelques administrateurs, des auteurs de documentation avec un accès au répertoire `doc` et un développeur qui n'a accès qu'aux répertoires `lib` et `tests`. Le fichier ACL ressemble donc à ceci :

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Le traitement consiste à lire le fichier dans une structure utilisable. Dans notre cas, pour simplifier, nous ne traiterons que les directives `avail`. Voici une fonction qui crée à partir du fichier un tableau associatif dont la clé est l'utilisateur et la valeur est une liste des chemins pour lesquels l'utilisateur a les droits en écriture :

```
def get_acl_access_data(nom_fichier_acl)
  # Lire le fichier ACL
  fichier_acl = File.read(nom_fichier_acl).split("\n").reject { |line| line == '' }
  acces = {}
  fichier_acl.each do |line|
    avail, utilisateurs, chemin = line.split('|')
    next unless avail == 'avail'
    utilisateurs.split(',').each do |utilisateur|
      access[utilisateur] ||= []
```

```
      access[utilisateur] << chemin
    end
  end
  acces
end
```

Pour le fichier d'ACL décrit plus haut, la fonction `get_acl_access_data` retourne une structure de données qui ressemble à ceci :

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

En plus des permissions, il faut déterminer les chemins impactés par la poussée pour s'assurer que l'utilisateur a bien droit d'y toucher.

La liste des fichiers modifiés est assez simplement obtenue par la commande `git log` complétée par l'option `--name-only` mentionnée dans **Table 2-2**

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Chaque fichier des *commits* doit être vérifié par rapport à la structure ACL retournée par la fonction `get_acl_access_data` pour déterminer si l'utilisateur a le droit de pousser tous ses *commits* :

```
# permission à certains utilisateurs de modifier certains sous-répertoires du proje
def verif_perms_repertoire
  acces = get_acl_access_data('acl')

  # verifier si quelqu'un cherche à pousser où il n'a pas le droit
  nouveaux_commits = `git rev-list #{$anciennerev}..#{$nouvellerev}`.split("\n")
  nouveaux_commits.each do |rev|
    fichiers_modifies = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\
    fichiers_modifies.each do |chemin|
      next if chemin.size == 0
      acces_permis = false
      acces[$utilisateur].each do |chemin_acces|
        if !chemin_acces || # l'utilisateur a un accès complet
```

```
          (chemin.index(chemin_acces) == 0) # acces à ce chemin
          acces_permis = true
        end
      end
      if !acces_permis
        puts "[ACL] Vous n'avez pas le droit de pousser sur #{chemin}"
        exit 1
      end
    end
  end
end

verif_perms_repertoire
```

On récupère la liste des nouveau *commits* poussés au serveur avec `git rev-list`. Ensuite, pour chacun des ces commits, on trouve les fichiers modifiés et on s'assure que l'utilisateur qui pousse a effectivement droit à l'accès au chemin modifié.

À présent, les utilisateurs ne peuvent plus pousser de *commits* comprenant un message incorrectement formaté ou des modifications à des fichiers hors de leur zone réservée.

TEST DE LA POLITIQUE

Après avoir lancé un `chmod u+x .git/hooks/update`, avec `.git/hooks/update` comme fichier dans lequel réside tout ce code, si vous essayez de pousser un *commit* avec un message de validation non conforme, vous obtiendrez la sortie suivante :

```
$ git push -f origin master
Décompte des objets : 5, fait.
Compression des objets: 100% (3/3), fait.
Écriture des objets : 100% (3/3), 323 bytes, fait.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), fait.
Vérification des règles...
(refs/heads/master) (8338c5) (c5b616)
[REGLE] Le message de validation n'est pas conforme
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Il y a plusieurs points à relever ici. Premièrement, une ligne indique l'endroit où le crochet est appelé.

```
Vérification des règles..
(refs/heads/master) (fb8c72) (c56860)
```

Le script `update` affiche ces lignes sur stdout au tout début. Tout ce que le script écrit sur stdout sera transmis au client.

La ligne suivante à remarquer est le message d'erreur.

```
[REGLE] Le message de validation n'est pas conforme
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

La première ligne a été écrite par le script, les deux autres l'ont été par Git pour indiquer que le script `update` a rendu un code de sortie non nul, ce qui a causé l'échec de la poussée. Enfin, il y a ces lignes :

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Il y a un message d'échec distant pour chaque référence que le crochet a rejetée et une indication que l'échec est dû spécifiquement à un échec de crochet.

Par ailleurs, si quelqu'un cherche à modifier un fichier auquel il n'a pas les droits d'accès lors d'une poussée, il verra quelque chose de similaire. Par exemple, si un auteur de documentation essaie de pousser un *commit* qui modifie quelque chose dans le répertoire `lib`, il verra :

```
[ACL] Vous n'avez pas le droit de pousser sur lib/test.rb
```

À partir de maintenant, tant que le script `update` est en place et exécutable, votre dépôt ne peut plus subir de poussées hors avancée rapide, n'accepte plus de messages sans format et vos utilisateurs sont cloîtrés.

Crochets côté client

Le problème de cette approche, ce sont les plaintes des utilisateurs qui résulteront inévitablement des échecs de leurs poussées. Leur frustration et leur confusion devant le rejet à la dernière minute d'un travail minutieux est tout à fait compréhensible. De plus, la correction nécessitera une modification de leur historique, ce qui n'est pas une partie de plaisir.

Pour éviter ce scénario, il faut pouvoir fournir aux utilisateurs des crochets côté client qui leur permettront de vérifier que leurs validations seront effectivement acceptées par le serveur. Ainsi, ils pourront corriger les problèmes avant de valider et avant que ces difficultés ne deviennent des casse-têtes. Ces scripts n'étant pas diffusés lors du clonage du projet, il vous faudra les distribuer d'une autre manière, puis indiquer aux utilisateurs de les copier dans leur répertoire `.git/hooks` et de les rendre exécutables. Vous pouvez distribuer ces crochets au sein du projet ou dans un projet annexe mais il n'y a aucun moyen de les mettre en place automatiquement.

Premièrement, pour éviter le rejet du serveur au motif d'un mauvais format du message de validation, il faut vérifier celui-ci avant que chaque *commit* ne soit enregistré. Pour ce faire, utilisons le crochet `commit-msg`. En lisant le message à partir du fichier passé en premier argument et en le comparant au format attendu, on peut forcer Git à abandonner la validation en cas d'absence de correspondance :

```ruby
#!/usr/bin/env ruby
fichier_message = ARGV[0]
message = File.read(fichier_message)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[REGLE] Le message de validation ne suit pas le format"
  exit 1
end
```

Avec ce fichier exécutable et à sa place dans `.git/hooks/commit-msg`, si une validation avec un message incorrect est tentée, voici le résultat :

```
$ git commit -am 'test'
[REGLE] Le message de validation ne suit pas le format
```

La validation n'a pas abouti. Néanmoins, si le message contient la bonne forme, Git accepte la validation :

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Ensuite, il faut s'assurer des droits sur les fichiers modifiés. Si le répertoire `.git` du projet contient une copie du fichier d'ACL précédemment utilisé, alors le script `pre-commit` suivant appliquera ses règles :

```
#!/usr/bin/env ruby

$utilisateur    = ENV['USER']

# [ insérer la fonction acl_access_data method ci-dessus ]

# Ne permet qu'à certains utilisateurs de modifier certains sous-répertoires
def verif_perms_repertoire
  acces = get_acl_access_data('.git/acl')

  fichiers_modifies = `git diff-index --cached --name-only HEAD`.split("\n")
  fichiers_modifies.each do |chemin|
    next if chemin.size == 0
    acces_permis = false
    acces[$utilisateur].each do |chemin_acces|
    if !chemin_acces || (chemin.index(chemin_acces) == 0)
      acces_permis = true
    end
    if !acces_permis
      puts "[ACL] Vous n'avez pas le droit de pousser sur #{path}"
      exit 1
    end
  end
end

verif_perms_repertoire
```

C'est grossièrement le même script que celui côté serveur, mais avec deux différences majeures. Premièrement, le fichier ACL est à un endroit différent parce que le script s'exécute depuis la copie de travail et non depuis le répertoire Git. Il faut donc changer le chemin vers le fichier d'ACL de :

```
access = get_acl_access_data('acl')
```

en :

```
access = get_acl_access_data('.git/acl')
```

L'autre différence majeure réside dans la manière d'obtenir la liste des fichiers modifiés. La fonction sur le serveur la recherche dans le journal des *commits* mais comme dans le cas actuel, le *commit* n'a pas encore été enregistré, il faut chercher la liste dans la zone d'index. Donc au lieu de :

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

on utilise :

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Mais à ces deux différences près, le script fonctionne de manière identique. Ce script a aussi une autre limitation : il s'attend à ce que l'utilisateur qui le lance localement soit identique à celui sur le serveur distant. S'ils sont différents, il faudra positionner manuellement la variable `$utilisateur`.

La dernière action à réaliser consiste à vérifier que les références poussées sont bien en avance rapide, mais l'inverse est plutôt rare. Pour obtenir une référence qui n'est pas en avance rapide, il faut soit rebaser après un *commit* qui a déjà été poussé, soit essayer de pousser une branche locale différente vers la même branche distante.

Par hypothèse, le serveur est déjà configuré avec `receive.denyDeletes` et `receive.denyNonFastForwards`, donc la seule action accidentelle qu'il faut intercepter reste le rebasage de *commits* qui ont déjà été poussés.

Voici un exemple de script `pre-rebase` qui fait cette vérification. Ce script récupère une liste de tous les *commits* qu'on est sur le point de réécrire et vérifie s'ils existent dans une référence distante. S'il en trouve un accessible depuis une des références distantes, il interrompt le rebasage :

```
#!/usr/bin/env ruby

branche_base = ARGV[0]
if ARGV[1]
  branche_thematique = ARGV[1]
else
  branche_thematique = "HEAD"
end

sha_cibles = `git rev-list #{branche_base}..#{branche_thematique}`.split("\n")
refs_distantes = `git branch -r`.split("\n").map { |r| r.strip }

shas_cibles.each do |sha|
  refs_distantes.each do |ref_distante|
    shas_pousses = `git rev-list ^#{sha}^@ refs/remotes/#{ref_distante}`
    if shas_pousses.split("\n").include?(sha)
      puts "[REGLE] Le commit #{sha} a déjà été poussé sur #{ref_distante}"
```

```
      exit 1
    end
  end
end
```

Ce script utilise une syntaxe qui n'a pas été abordée à la section **"Sélection des versions"**. La liste des *commits* déjà poussés est obtenue avec cette commande :

```
`git rev-list ^#{sha}^@ refs/remotes/#{ref_distante}`
.
```

La syntaxe SHA^@ fait référence à tous les parents du *commit*. Les *commits* recherchés sont accessibles depuis le dernier *commit* distant et inaccessibles depuis n'importe quel parent de n'importe quel SHA qu'on cherche à pousser. C'est la définition d'avance rapide.

La limitation de cette approche reste qu'elle peut s'avérer très lente et non nécessaire. Si vous n'essayez pas de forcer à pousser avec l'option `-f`, le serveur vous avertira et n'acceptera pas la poussée. Cependant, cela reste un exercice intéressant qui peut aider théoriquement à éviter un rebasage qui devra être annulé plus tard.

Résumé

Nous avons traité la plupart des moyens principaux de personnaliser le client et le serveur Git pour mieux l'adapter à toutes les méthodes et les projets. Nous avons couvert toutes sortes de réglages de configurations, d'attributs dans des fichiers et de crochets d'événement et nous avons construit un exemple de politique de gestion de serveur. Vous voilà prêt à adapter Git à quasiment toutes les gestions dont vous avez rêvé.

Git et les autres systèmes 9

Le monde n'est pas parfait. Habituellement, vous ne pouvez pas basculer immédiatement sous Git tous les projets que vous pourriez rencontrer. Quelques fois, vous êtes bloqué sur un projet utilisant un autre VCS et vous regrettez que ce ne soit pas Git. Dans la première partie de ce chapitre, nous traiterons de la manière d'utiliser git comme client pour les projets utilisant un autre système.

À un moment, vous voudrez convertir votre projet à Git. La seconde partie de ce chapitre traite la migration de votre projet dans Git depuis certains systèmes spécifiques et enfin par un script d'import personnalisé pour les cas non-standards.

Git comme client

Git fournit de si bonnes sensations aux développeurs que de nombreuses personnes ont cherché à l'utiliser sur leur station de travail, même si le reste de leur équipe utilise un VCS complètement différent. Il existe un certain nombre d'adaptateurs appelés « passerelles ». Nous allons en décrire certains des plus communs.

Git et Subversion

Aujourd'hui, la majorité des projets de développement libre et un grand nombre de projets dans les sociétés utilisent Subversion pour gérer leur code source. Il a été le VCS libre le plus populaire depuis une bonne décennie et a été considéré comme le choix *de facto* pour les projets open-source. Il est aussi très similaire à CVS qui a été le grand chef des gestionnaires de source avant lui.

Une des grandes fonctionnalités de Git est sa passerelle vers Subversion, `git svn`. Cet outil vous permet d'utiliser Git comme un client valide d'un serveur Subversion pour que vous puissiez utiliser les capacités de Git en local puis poussez sur le serveur Subversion comme si vous utilisiez Subversion localement. Cela signifie que vous pouvez réaliser localement les embranchements et

les fusions, utiliser l'index, utiliser le rebasage et le picorage de *commits*, etc, tandis que vos collaborateurs continuent de travailler avec leurs méthodes ancestrales et obscures. C'est une bonne manière d'introduire Git dans un environnement professionnel et d'aider vos collègues développeurs à devenir plus efficaces tandis que vous ferez pression pour une modification de l'infrastructure vers l'utilisation massive de Git. La passerelle Subversion n'est que la première dose vers la drogue du monde des DVCS.

`GIT SVN`

La commande de base dans Git pour toutes les commandes de passerelle est `git svn`. Vous préfixerez tout avec cette paire de mots. Les possibilités étant nombreuses, nous traiterons des plus communes pendant que nous détaillerons quelques petits modes de gestion.

Il est important de noter que lorsque vous utilisez `git svn`, vous interagissez avec Subversion qui est un système fonctionnant très différemment de Git. Bien que vous **puissiez** réaliser des branches locales et les fusionner, il est généralement conseillé de conserver votre historique le plus linéaire possible en rebasant votre travail et en évitant des activités telles qu'interagir dans le même temps avec un dépôt Git distant.

Ne réécrivez pas votre historique avant d'essayer de pousser à nouveau et ne poussez pas en parallèle dans un dépôt Git pour collaborer avec vos collègues développant avec Git. Subversion ne supporte qu'un historique linéaire et il est très facile de l'égarer. Si vous travaillez avec une équipe dont certains membres utilisent SVN et d'autres utilisent Git, assurez-vous que tout le monde n'utilise que le serveur SVN pour collaborer, cela vous rendra service.

INSTALLATION

Pour montrer cette fonctionnalité, il faut un serveur SVN sur lequel vous avez des droits en écriture. Pour copier ces exemples, faites une copie inscriptible de mon dépôt de test. Dans cette optique, vous pouvez utiliser un outil appelé `svnsync` qui est livré avec les versions les plus récentes de Subversion — il devrait être distribué avec les versions à partir de 1.4. Pour ces tests, j'ai créé sur Google code un nouveau dépôt Subversion qui était une copie partielle du projet `protobuf` qui est un outil qui encode les données structurées pour une transmission par réseau.

En préparation, créez un nouveau dépôt local Subversion :

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Ensuite, autorisez tous les utilisateurs à changer les revprops — le moyen le plus simple consiste à ajouter un script `pre-revprop-change` qui renvoie toujours 0 :

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Vous pouvez à présent synchroniser ce projet sur votre machine locale en lançant `svnsync init` avec les dépôts source et cible.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Cela initialise les propriétés nécessaires à la synchronisation. Vous pouvez ensuite cloner le code en lançant :

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[…]
```

Bien que cette opération ne dure que quelques minutes, si vous essayez de copier le dépôt original sur un autre dépôt distant au lieu d'un dépôt local, le processus durera près d'une heure, en dépit du fait qu'il y a moins de 100 *commits*. Subversion doit cloner révision par révision puis pousser vers un autre dépôt — c'est ridiculement inefficace mais c'est la seule possibilité.

DÉMARRAGE

Avec des droits en écriture sur un dépôt Subversion, vous voici prêt à expérimenter une méthode typique. Commençons par la commande `git svn clone` qui importe un dépôt Subversion complet dans un dépôt Git local. Souvenez-vous que si vous importez depuis un dépôt Subversion hébergé sur Internet, il faut remplacer l'URL `file://tmp/test-svn` ci-dessous par l'URL de votre dépôt Subversion :

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    A   m4/acx_pthread.m4
    A   m4/stl_hash.m4
    A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    A   java/src/test/java/com/google/protobuf/WireFormatTest.java
…
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/br
Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

Cela équivaut à lancer `git svn init` suivi de `git svn fetch` sur l'URL que vous avez fournie. Cela peut prendre un certain temps. Le projet de test ne contient que 75 *commits* et la taille du code n'est pas extraordinaire, ce qui prend juste quelques minutes. Cependant, Git doit extraire chaque version, une par une et les valider individuellement. Pour un projet contenant des centaines ou des milliers de *commits*, cela peut prendre littéralement des heures ou même des jours à terminer.

La partie `-T trunk -b branches -t tags` indique à Git que ce dépôt Subversion suit les conventions de base en matière d'embranchement et d'étiquetage. Si vous nommez votre trunk, vos branches ou vos étiquettes différemment, vous pouvez modifier ces options. Comme cette organisation est la plus commune, ces options peuvent être simplement remplacées par `-s` qui signifie structure standard. La commande suivante est équivalente :

```
$ git svn clone file:///tmp/test-svn -s
```

À présent, vous disposez d'un dépôt Git valide qui a importé vos branches et vos étiquettes :

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
```

```
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Il est important de remarquer comment cet outil sous-classe vos références distantes différemment. Voyons de plus près avec la commande Git de plomberie `show-ref` :

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git ne fait pas cela quand il clone depuis un serveur Git ; voici à quoi ressemble un dépôt avec des étiquettes juste après le clonage :

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git entrepose les étiquettes directement dans `refs/tags`, plutôt que de les traiter comme des branches distantes.

VALIDER EN RETOUR SUR LE SERVEUR SUBVERSION

Comme vous disposez d'un dépôt en état de marche, vous pouvez commencer à travailler sur le projet et pousser vos *commits* en utilisant efficacement Git comme client SVN. Si vous éditez un des fichiers et le validez, vous créez un *commit* qui existe localement dans Git mais qui n'existe pas sur le serveur Subversion :

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Ensuite, vous avez besoin de pousser vos modifications en amont. Remarquez que cela modifie la manière de travailler par rapport à Subversion — vous pouvez réaliser plusieurs validations en mode déconnecté pour ensuite les pousser toutes en une fois sur le serveur Subversion. Pour pousser sur un serveur Subversion, il faut lancer la commande `git svn dcommit` :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origi
Resetting to the latest refs/remotes/origin/trunk
```

Cette commande rassemble tous les *commits* que vous avez validés par-dessus le code du serveur Subversion et réalise un *commit* sur le serveur pour chacun, puis réécrit l'historique Git local pour y ajouter un identifiant unique. Cette étape est à souligner car elle signifie que toutes les sommes de contrôle SHA-1 de vos *commits* locaux ont changé. C'est en partie pour cette raison que c'est une idée très périlleuse de vouloir travailler dans le même temps avec des serveurs Git distants. L'examen du dernier *commit* montre que le nouveau `git-svn-id` a été ajouté :

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Remarquez que la somme de contrôle SHA qui commençait par `4af61fd` quand vous avez validé commence à présent par `95e0222`. Si vous souhaitez pousser à la fois sur un serveur Git et un serveur Subversion, il faut obligatoirement pousser (`dcommit`) sur le serveur Subversion en premier, car cette action va modifier vos données des *commits*.

TIRER DES MODIFICATIONS

Quand vous travaillez avec d'autres développeurs, il arrive à certains moments que ce qu'un développeur a poussé provoque un conflit lorsqu'un autre voudra pousser à son tour. Cette modification sera rejetée jusqu'à ce qu'elle soit fusionnée. Dans `git svn`, cela ressemble à ceci :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ, using reba
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145 c80b6127dd04f5fcda218730ddf3a2da4eb3
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Pour résoudre cette situation, vous pouvez lancer la commande `git svn rebase` qui tire depuis le serveur toute modification apparue entre temps et rebase votre travail sur le sommet de l'historique du serveur :

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ, using reba
:100644 100644 65536c6e30d263495c17d781962cfff12422693a b34372b25ccf4945fe5658fa381b075045e7
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M       README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

À présent, tout votre travail se trouve au-delà de l'historique du serveur et vous pouvez effectivement réaliser un `dcommit` :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r85
    M   README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/origi
Resetting to the latest refs/remotes/origin/trunk
```

Il est important de se souvenir qu'à la différence de Git qui requiert une fusion avec les modifications distantes non présentes localement avant de pouvoir pousser, `git svn` ne vous y contraint que si vos modifications provoquent un conflit (de la même manière que `svn`). Si une autre personne pousse une modification à un fichier et que vous poussez une modification à un autre fichier, votre `dcommit` passera sans problème :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   configure.ac
Committed r87
    M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ,
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb6
First, rewinding head to replay your work on top of it...
```

Il faut s'en souvenir car le résultat de ces actions est un état du dépôt qui n'existait pas sur aucun des ordinateurs quand vous avez poussé. Si les modifications sont incompatibles mais ne créent pas de conflits, vous pouvez créer des défauts qui seront très difficiles à diagnostiquer. C'est une grande différence avec un serveur Git — dans Git, vous pouvez tester complètement l'état du projet sur votre système client avant de le publier, tandis qu'avec SVN, vous ne pouvez jamais être totalement certain que les états avant et après validation sont identiques.

Vous devrez aussi lancer cette commande pour tirer les modifications depuis le serveur Subversion, même si vous n'êtes pas encore prêt à valider. Vous pouvez lancer `git svn fetch` pour tirer les nouveaux *commits*, mais `git svn rebase` tire non seulement les *commits* distants mais rebase aussi vos *commits* locaux.

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Lancer `git svn rebase` de temps en temps vous assure que votre travail est toujours synchronisé avec le serveur. Vous devrez cependant vous assurer que votre copie de travail est propre quand vous la lancez. Si vous avez des modifications locales, il vous faudra soit remiser votre travail, soit valider temporairement vos modifications avant de lancer `git svn rebase`, sinon la commande s'arrêtera si elle détecte que le rebasage provoquerait un conflit de fusion.

LE PROBLÈME AVEC LES BRANCHES GIT

Après vous être habitué à la manière de faire avec Git, vous souhaiterez sûrement créer des branches thématiques, travailler dessus, puis les fusionner. Si vous poussez sur un serveur Subversion via `git svn`, vous souhaiterez à chaque fois rebaser votre travail sur une branche unique au lieu de fusionner les branches ensemble. La raison principale en est que Subversion gère un historique linéaire et ne gère pas les fusions comme Git y excelle. De ce fait, `git svn` suit seulement le premier parent lorsqu'il convertit les instantanés en *commits* Subversion.

Supposons que votre historique ressemble à ce qui suit. Vous avez créé une branche `experience`, avez réalisé deux validations puis les avez fusionnées dans `master`. Lors du `dcommit`, vous voyez le résultat suivant :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   CHANGES.txt
Committed r89
    M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    M   COPYING.txt
    M   INSTALL.txt
Committed r90
    M   INSTALL.txt
    M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Lancer `dcommit` sur une branche avec un historique fusionné fonctionne correctement, à l'exception que l'examen de l'historique du projet Git indique qu'il n'a réécrit aucun des *commits* réalisés sur la branche `experience`, mais que toutes les modifications introduites apparaissent dans la version SVN de l'unique *commit* de fusion.

Quand quelqu'un d'autre clone ce travail, tout ce qu'il voit, c'est le *commit* de la fusion avec toutes les modifications injectées en une fois, comme si vous aviez lancé `git merge --squash`. Il ne voit aucune information sur son origine ni sur sa date de validation.

LES EMBRANCHEMENTS DANS SUBVERSION

La gestion de branches dans Subversion n'a rien à voir avec celle de Git. Évitez de l'utiliser autant que possible. Cependant vous pouvez créer des branches et valider dessus dans Subversion en utilisant `git svn`.

CRÉER UNE NOUVELLE BRANCHE SVN

Pour créer une nouvelle branche dans Subversion, vous pouvez utiliser la commande `git svn branch [nom de la branche]` :

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/br
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f672
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Cela est équivalent à la commande Subversion `svn copy trunk branches/opera` et réalise l'opération sur le serveur Subversion. Remarquez que cette commande ne vous bascule pas sur cette branche ; si vous validez, le *commit* s'appliquera à `trunk` et non à la branche `opera`.

BASCULER DE BRANCHE ACTIVE

Git devine la branche cible des `dcommits` en se référant au sommet des branches Subversion dans votre historique — vous ne devriez en avoir qu'un et celui-ci devrait être le dernier possédant un `git-svn-id` dans l'historique actuel de votre branche.

Si vous souhaitez travailler simultanément sur plusieurs branches, vous pouvez régler vos branches locales pour que le `dcommit` arrive sur une branche

Subversion spécifique en les démarrant sur le *commit* de cette branche importée depuis Subversion. Si vous voulez une branche `opera` sur laquelle travailler séparément, vous pouvez lancer :

```
$ git branch opera remotes/origin/opera
```

À présent, si vous voulez fusionner votre branche `opera` dans `trunk` (votre branche `master`), vous pouvez le faire en réalisant un `git merge` normal. Mais vous devez préciser un message de validation descriptif (via `-m`), ou la fusion indiquera simplement « Merge branch opera » au lieu d'un message plus informatif.

Souvenez-vous que bien que vous utilisez `git merge` qui facilitera l'opération de fusion par rapport à Subversion (Git détectera automatiquement l'ancêtre commun pour la fusion), ce n'est pas un *commit* de fusion normal de Git. Vous devrez pousser ces données finalement sur le serveur Subversion qui ne sait pas tracer les *commits* possédant plusieurs parents. Donc, ce sera un *commit* unique qui englobera toutes les modifications de l'autre branche. Après avoir fusionné une branche dans une autre, il est difficile de continuer à travailler sur cette branche, comme vous le feriez normalement dans Git. La commande `dcommit` qui a été lancée efface toute information sur la branche qui a été fusionnée, ce qui rend faux tout calcul d'antériorité pour la fusion. `dcommit` fait ressembler le résultat de `git merge` à celui de `git merge --squash`. Malheureusement, il n'y a pas de moyen efficace de remédier à ce problème — Subversion ne stocke pas cette information et vous serez toujours contraints par ses limitations si vous l'utilisez comme serveur. Pour éviter ces problèmes, le mieux reste d'effacer la branche locale (dans notre cas, `opera`) dès qu'elle a été fusionnée dans `trunk`.

COMMANDES SUBVERSION

La boîte à outil `git svn` fournit des commandes de nature à faciliter la transition vers Git en mimant certaines commandes disponibles avec Subversion. Voici quelques commandes qui vous fournissent les mêmes services que Subversion.

L'historique dans le style Subversion

Si vous êtes habitué à Subversion, vous pouvez lancer `git svn log` pour visualiser votre historique dans un format SVN :

```
$ git svn log
------------------------------------------------------------------------
```

```
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

Deux choses importantes à connaître sur `git svn log`. Premièrement, à la différence de la véritable commande `svn log` qui interroge le serveur, cette commande fonctionne hors connexion. Deuxièmement, elle ne montre que les *commits* qui ont été effectivement remontés sur le serveur Subversion. Les *commits* locaux qui n'ont pas encore été remontés via `dcommit` n'apparaissent pas, pas plus que ceux qui auraient été poussés sur le serveur par des tiers entre-temps. Cela donne plutôt le dernier état connu des *commits* sur le serveur Subversion.

Annotations SVN

De la même manière que `git svn log` simule une commande `svn log` déconnectée, vous pouvez obtenir l'équivalent de `svn annotate` en lançant `git svn blame [fichier]`. Le résultat ressemble à ceci :

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protoco
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Ici aussi, tous les *commits* locaux dans Git ou ceux poussés sur Subversion dans l'intervalle n'apparaissent pas.

Information sur le serveur SVN

Vous pouvez aussi obtenir le même genre d'information que celle fournie par `svn info` en lançant `git svn info` :

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Comme `blame` et `log`, cette commande travaille hors connexion et n'est à jour qu'à la dernière date à laquelle vous avez communiqué avec le serveur Subversion.

Ignorer ce que Subversion ignore

Si vous clonez un dépôt Subversion contenant des propriétés `svn:ignore`, vous souhaiterez sûrement paramétrer les fichiers `.gitignore` en correspondance pour vous éviter de valider accidentellement des fichiers qui ne devraient pas l'être. `git svn` dispose de deux commandes pour le faire.

La première est `git svn create-ignore` qui crée automatiquement pour vous les fichiers `.gitignore` prêts pour l'inclusion dans votre prochaine validation.

La seconde commande est `git svn show-ignore` qui affiche sur `stdout` les lignes nécessaires à un fichier `.gitignore` qu'il suffira de rediriger dans votre fichier d'exclusion de projet :

```
$ git svn show-ignore > .git/info/exclude
```

De cette manière, vous ne parsemez pas le projet de fichiers `.gitignore`. C'est une option optimale si vous êtes le seul utilisateur de Git dans une équipe Subversion et que vos coéquipiers ne veulent pas voir de fichiers `.gitignore` dans le projet.

RÉSUMÉ SUR GIT-SVN

Les outils `git svn` sont utiles si vous êtes bloqué avec un serveur Subversion pour le moment ou si vous devez travailler dans un environnement de développement qui nécessite un serveur Subversion. Il faut cependant les considérer comme une version estropiée de Git ou vous pourriez rencontrer des problèmes de conversion qui vous embrouilleront vous et vos collaborateurs. Pour éviter tout problème, essayez de suivre les principes suivants :

- Gardez un historique Git linéaire qui ne contient pas de *commits* de fusion issus de `git merge`.
- Rebasez tout travail réalisé en dehors de la branche principale sur celle-ci ; ne la fusionnez pas.
- Ne mettez pas en place et ne travaillez pas en parallèle sur un serveur Git. Si nécessaire, montez-en un pour accélérer les clones pour de nouveaux développeurs mais n'y poussez rien qui n'ait déjà une entrée `git-svn-id`. Vous devriez même y ajouter un crochet `pre-receive` qui vérifie la présence de `git-svn-id` dans chaque message de validation et rejette les remontées dont un des *commits* n'en contiendrait pas.

Si vous suivez ces principes, le travail avec un serveur Subversion peut être supportable. Cependant, si le basculement sur un vrai serveur Git est possible, votre équipe y gagnera beaucoup.

Git et Mercurial

L'univers des systèmes de gestion de version distribués ne se limite pas à Git. En fait, il existe de nombreux autres systèmes, chacun avec sa propre approche sur la gestion distribuée des versions. À part Git, le plus populaire est Mercurial, et ces deux-ci sont très ressemblants par de nombreux aspects.

La bonne nouvelle si vous préférez le comportement de Git côté client mais que vous devez travailler sur un projet géré sous Mercurial, c'est que l'on peut utiliser Git avec un dépôt géré sous Mercurial. Du fait que Git parle avec les dépôts distants au moyen de greffons de protocole distant, il n'est pas surprenant que cette passerelle prenne la forme d'un greffon de protocole distant. Le projet s'appelle git-remote-hg et peut être trouvé à l'adresse ***https://github.com/felipec/git-remote-hg***.

GIT-REMOTE-HG

Premièrement, vous devez installer git-remote-hg. Cela revient simplement à copier ce fichier quelque part dans votre chemin de recherche, comme ceci :

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

…en supposant que `~/bin` est présent dans votre $PATH. git-remote-hg est aussi dépendant de la bibliothèque `Mercurial` pour Python. Si Python est déjà installé, c'est aussi simple que :

```
$ pip install mercurial
```

Si Python n'est pas déjà installé, visitez ***https://www.python.org/*** et récupérez-le.

La dernière dépendance est le client Mercurial. Rendez-vous sur ***http://mercurial.selenic.com/*** et installez-le si ce n'est pas déjà fait.

Maintenant, vous voilà prêt. Vous n'avez besoin que d'un dépôt Mercurial où pousser. Heureusement, tous les dépôts Mercurial peuvent servir et nous allons donc simplement utiliser le dépôt "hello world" dont tout le monde se sert pour apprendre Mercurial :

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

DÉMARRAGE

Avec un dépôt « côté serveur » maintenant disponible, détaillons un flux de travail typique. Comme vous le verrez, ces deux systèmes sont suffisamment similaires pour qu'il y ait peu de friction.

Comme toujours avec Git, commençons par cloner :

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branche
* 65bb417 Create a standard "hello, world" program
```

Notez bien que pour travailler avec un dépôt Mercurial, on utilise la commande standard `git clone`. C'est dû au fait que git-remote-hg travaille à un niveau assez bas, en utilisant un mécanisme similaire à celui du protocole HTTP/S de Git. Comme Git et Mercurial sont tous les deux organisés pour que chaque client récupère une copie complète de l'historique du dépôt, cette com-

mande réalise rapidement un clone complet, incluant tout l'historique du projet.

La commande log montre deux *commits*, dont le dernier est pointé par une ribambelle de refs. En fait, certaines d'entre elles n'existent par vraiment. Jetons un œil à ce qui est réellement présent dans le répertoire `.git` :

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

9 directories, 5 files
```

Git-remote-hg essaie de rendre les choses plus idiomatiquement Git-esques, mais sous le capot, il gère la correspondance conceptuelle entre deux systèmes légèrement différents. Par exemple, le fichier `refs/hg/origin/branches/default` est un fichier Git de références, qui contient le SHA-1 commençant par « ac7955c », qui est le *commit* pointé par `master`. Donc le répertoire `refs/hg` est en quelque sorte un faux `refs/remotes/origin`, mais il contient la distinction entre les marque-pages et les branches.

Le fichier `notes/hg` est le point de départ pour comprendre comment git-remote-hg fait correspondre les empreintes des *commits* Git avec les IDs de modification de Mercurial. Explorons-le un peu :

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800
```

```
Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...  65bb417...
100644 blob 485e178...  ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Donc, `refs/notes/hg` pointe sur un arbre qui correspond dans la base de données des objets de Git à une liste des autres objets avec des noms. `git-ls-tree` affiche le mode, le type, l'empreinte de l'objet et le nom de fichier des articles d'un arbre. Quand nous creusons un de ces articles, nous trouvons à l'intérieur un blob appelé « ac9117f » (l'empreinte SHA-1 du *commit* pointé par `master`), avec le contenu « 0a04b98 » (qui est l'ID de la modification Mercurial au sommet de la branche `default`).

La bonne nouvelle est que nous n'avons quasiment pas à nous soucier de tout ceci. Le mode de travail ne sera pas très différent de celui avec un serveur distant Git.

Il reste une chose à gérer avant de passer à la suite : les fichiers `ignore`. Mercurial et Git utilisent un mécanisme très similaire pour cette fonctionnalité, mais il est très probable que vous ne souhaitez pas valider un fichier `.gitignore` dans un dépôt Mercurial. Heureusement, Git dispose d'un moyen d'ignorer les fichiers d'un dépôt local et le format Mercurial est compatible avec Git. Il suffit donc de le copier :

```
$ cp .hgignore .git/info/exclude
```

Le fichier `.git/info/exclude` se comporte simplement comme un fichier `.gitignore`, mais n'est pas inclus dans les *commits*.

DÉROULEMENT

Supposons que nous avons travaillé et validé quelques *commits* sur la branche `master` et que nous sommes prêt à pousser ce travail sur un dépôt distant. Notre dépôt ressemble actuellement à ceci :

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
```

```
* ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/bra
* 65bb417 Create a standard "hello, world" program
```

Notre branche `master` est en avance de deux *commits* par rapport à `origin/master`, mais ces deux *commits* n'existent que sur notre machine locale. Voyons si quelqu'un d'autre a poussé son travail dans le même temps :

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/bra
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Comme nous avons utilisé l'option `--all`, nous voyons les références « notes » qui sont utilisés en interne par git-remote-hg, mais nous pouvons les ignorer. Le reste était attendu ; `origin/master` a avancé d'un *commit* et notre historique a divergé. À la différence d'autres systèmes que nous décrivons dans ce chapitre, Mercurial est capable de gérer les fusions, donc ce que nous allons faire n'a rien d'extraordinaire.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/b
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Parfait. Nous lançons les tests et tout passe, et nous voilà prêts à partager notre travail avec l'équipe :

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

C'est fini ! Si vous inspectez le dépôt Mercurial, vous verrez que le résultat se présente comme attendu :

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

La modification numérotée *2* a été faite par Mercurial et celles numérotées *3* et *4* ont été faites par git-remote-hg, en poussant les *commits* réalisés avec Git.

BRANCHES ET MARQUE-PAGES

Git n'a qu'un seul type de branche : une référence qui se déplace quand des *commits* sont ajoutés. Dans Mercurial, ce type de référence est appelé « marque-page » et se comporte de la même manière qu'une branche Git.

Le concept de « branche » dans Mercurial est plus contraignant. La branche sur laquelle une modification est réalisée est enregistrée *avec la modification*, ce qui signifie que cette dernière sera toujours présente dans l'historique du dépôt. Voici un exemple d'un *commit* ajouté à la branche `develop` :

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

Notez la ligne qui commence par « branch ». Git ne peut pas vraiment répliquer ce comportement (il n'en a pas besoin ; les deux types de branches peuvent être représentés par une ref Git), mais git-remote-hg a besoin de comprendre cette différence, puisque qu'elle a du sens pour Mercurial.

La création de marque-pages Mercurial est aussi simple que la création de branches Git. Du côté Git :

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

C'est tout ce qui est nécessaire. Du côté Mercurial, cela ressemble à ceci :

```
$ hg bookmarks
   featureA                 5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
```

```
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Remarquez la nouvelle étiquette `[featureA]` sur la révision 5. Elle se comporte exactement comme une branche Git du côté Git, avec une exception : vous ne pouvez pas effacer un marque-page depuis le côté Git (c'est une limitation des greffons de gestion distante).

Vous pouvez travailler aussi sur une branche « lourde » Mercurial : placez une branche dans l'espace de nom `branches` :

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

Voici à quoi ça ressemble du côté Mercurial :

```
$ hg branches
permanent                      7:a4529d07aad4
develop                        6:8f65e5e02793
default                        5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
```

```
| |  summary:     Merge remote-tracking branch 'origin/master'
[...]
```

Le nom de branche « permanent » a été enregistré avec la modification marquée *7*.

Du côté Git, travailler avec les deux styles de branches revient au même : employez les commandes `checkout`, `commit`, `fetch`, `merge`, `pull` et `push` comme vous feriez normalement. Une chose à savoir cependant est que Mercurial ne supporte pas la réécriture de l'historique mais seulement les ajouts. Voici à quoi ressemble le dépôt Mercurial après un rebasage interactif et une poussée forcée :

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Les modifications *8*, *9* et *10* ont été créées et appartiennent à la branche `permanent` mais les anciennes modifications sont toujours présentes. Ça a toutes les chances de perdre vos collègues qui utilisent Mercurial, donc c'est à éviter à tout prix.

RÉSUMÉ MERCURIAL

Git et Mercurial sont suffisamment similaires pour que le travail pendulaire entre les deux se passe sans accroc. Si vous évitez de modifier l'historique qui a déjà quitté votre machine (comme il l'est recommandé), vous pouvez tout simplement ignorer que le dépôt distant fonctionne avec Mercurial.

Git et Perforce

Perforce est un système de version très populaire dans les environnements professionnels. Il existe depuis 1995, ce qui en fait le système le plus ancien abordé dans ce chapitre. Avec cette information en tête, il apparaît construit avec les contraintes de cette époque ; il considère que vous êtes toujours connecté à un serveur central et une seule version est conservée sur le disque dur local. C'est certain, ses fonctionnalités et ses contraintes correspondent à quelques problèmes spécifiques, mais de nombreux projets utilisent Perforce là où Git fonctionnerait réellement mieux.

Il y a deux options pour mélanger l'utilisation de Perforce et de Git. La première que nous traiterons est le pont « Git Fusion » créé par les développeurs de Perforce, qui vous permet d'exposer en lecture-écriture des sous-arbres de votre dépôt Perforce en tant que dépôts Git. La seconde s'appelle git-p4, un pont côté client qui permet d'utiliser Git comme un client Perforce, sans besoin de reconfigurer le serveur Perforce.

GIT FUSION

Perforce fournit un produit appelé Git Fusion (disponible sur ***http://www.perforce.com/git-fusion***), qui synchronise un serveur Perforce avec des dépôts Git du côté serveur.

Installation

Pour nos exemples, nous utiliserons la méthode d'installation de Git Fusion la plus facile qui consiste à télécharger une machine virtuelle qui embarque le *daemon* Perforce et Git Fusion. Vous pouvez obtenir la machine virtuelle depuis ***http://www.perforce.com/downloads/Perforce/20-User***, et une fois téléchargée, importez-la dans votre logiciel favori de virtualisation (nous utiliserons VirtualBox).

Au premier lancement de la machine, il vous sera demandé de personnaliser quelques mots de passe pour trois utilisateurs Linux (`root`, `perforce` et `git`), et de fournir un nom d'instance qui peut être utilisé pour distinguer cette installation des autres sur le même réseau. Quand tout est terminé, vous verrez ceci :

FIGURE 9-1

L'écran de démarrage de la machine virtuelle Git Fusion.

Prenez note de l'adresse IP qui est indiquée ici, car nous en aurons besoin plus tard. Ensuite, nous allons créer l'utilisateur Perforce. Sélectionnez l'option « Login » en bas de l'écran et appuyez sur *Entrée* (ou connectez-vous en SSH à la machine), puis identifiez-vous comme `root`. Ensuite, utilisez ces commandes pour créer un utilisateur :

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

La première commande va ouvrir un éditeur VI pour personnaliser l'utilisateur, mais vous pouvez accepter les valeurs par défaut en tapant `:wq` et en ap-

puyant sur *Entrée*. La seconde vous demandera d'entrer le mot de passe deux fois. C'est tout ce qu'il faut faire depuis une invite de commande, et on peut quitter la session.

L'action suivante consiste à indiquer à Git de ne pas vérifier les certificats SSL. L'image Git Fusion contient un certificat, mais celui-ci ne correspond pas au domaine de l'adresse IP de votre machine virtuelle, donc Git va rejeter la connexion HTTPS. Pour une installation permanente, consultez le manuel Perforce Git Fusion pour installer un certificat différent ; pour l'objet de notre exemple, ceci suffira :

```
$ export GIT_SSL_NO_VERIFY=true
```

Maintenant, nous pouvons tester que tout fonctionne correctement.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

La machine virtuelle contient un projet exemple que vous pouvez cloner. Ici, nous clonons via HTTPS, avec l'utilisateur `john` que nous avons créé auparavant ; Git demande le mot de passe pour cette connexion, mais le cache d'identifiant permettra de sauter cette étape par la suite.

Configuration de Fusion

Une fois que Git Fusion est installé, vous désirerez sûrement modifier la configuration. C'est assez facile à faire via votre client Perforce favori ; rapatriez simplement le répertoire `//.git-fusion` du serveur Perforce dans votre espace de travail. La structure du fichier ressemble à ceci :

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
```

```
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

Le répertoire `objects` est utilisé en interne par Git Fusion pour faire correspondre les objets Perforce avec Git et vice versa et il n'y a pas lieu d'y toucher. Il y a un fichier `p4gf_config` global dans ce répertoire, ainsi qu'un fichier pour chaque dépôt. Ce sont les fichiers de configuration qui déterminent comment Git Fusion se comporte. Examinons le fichier à la racine :

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

Nous ne nous étendrons pas sur les significations des différents paramètres, mais on voit que c'est un simple fichier INI, du même style que ceux utilisés par Git. Ce fichier spécifie les options globales, qui peuvent être surchargées par chaque fichier de configuration spécifique à un dépôt, tel que `repos/Talk-`

house/p4gf_config. Si vous ouvrez ce fichier, vous verrez une section [@repo] contenant des paramétrages différents des paramètres globaux par défaut. Vous verrez aussi des sections ressemblant à ceci :

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

C'est la correspondance entre une branche Perforce et une branche Git. Le nom de la section est libre, du moment qu'il est unique. git-branch-name vous permet de convertir un chemin du dépôt qui serait encombrant sous Git en quelque chose de plus utilisable. L'entrée view contrôle comment les fichiers Perforce sont transformés en dépôts Git, en utilisant la syntaxe standard de description de vue. Des correspondances multiples peuvent être indiquées, comme dans cet exemple :

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

De cette manière, si votre montage d'espace de travail normal change de structure de répertoires, vous pouvez répliquer cette modification dans le dépôt Git.

Le dernier fichier que nous examinerons est users/p4gf_usermap, qui fait correspondre les utilisateurs Perforce avec les utilisateurs Git, et qui n'est même pas nécessaire. Quand une modification Perforce est convertie en *commit* Git, le comportement par défaut de Git Fusion consiste à rechercher l'utilisateur Perforce et à utiliser son adresse de courriel et son nom complet comme champs d'auteur/validateur dans Git. Dans l'autre sens, le comportement par défaut consiste à rechercher l'utilisateur Perforce correspondant à l'adresse de courriel stockée dans le champ auteur du *commit* Git et de soumettre une modification avec cet identifiant (si les permissions l'accordent). Dans la plupart des cas, ce comportement suffira, mais considérons tout de même le fichier de correspondance suivant :

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Chaque ligne est de la forme <utilisateur> <courriel> <nom complet> et crée une correspondance unique. Les deux premières lignes font correspondre deux adresses de courriel distinctes avec le même utilisateur Per-

force. C'est utile si vous avez créé des *commits* Git sous plusieurs adresses de courriel (ou modifié votre adresse de courriel), mais que vous voulez les faire correspondre au même utilisateur Perforce. À la création d'un *commit* Git depuis une modification Perforce, la première ligne correspondant à l'utilisateur Perforce est utilisée pour fournir l'information d'auteur à Git.

Les deux dernières lignes masquent les noms réels de Bob et Joe dans les *commits* Git créés. C'est très utile si vous souhaitez ouvrir les sources d'un projet interne, mais que vous ne souhaitez pas rendre public le répertoire de vos employés. Notez que les adresses de courriel et les noms complets devraient être uniques, à moins que vous ne souhaitiez publier tous les *commits* Git avec un auteur unique fictif.

Utilisation

Perforce Git Fusion est une passerelle à double-sens entre les contrôles de version Perforce et Git. Voyons comment cela se passe du côté Git. Nous supposerons que nous avons monté le projet « Jam » en utilisant le fichier de configuration ci-dessus, et que nous pouvons le cloner comme ceci :

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerk
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

La première fois que vous le faites, cela peut durer un certain temps. Ce qui se passe, c'est que Git Fusion convertit toutes les modifications concernées de l'historique Perforce en *commits* Git. Cela se passe localement sur le serveur, donc c'est plutôt rapide, mais si votre historique est long, ce n'est pas immédi-

at. Les récupérations subséquentes ne lancent que des conversions incrémentales, ce qui devrait correspondre à la vitesse native de Git.

Comme vous pouvez le voir, notre dépôt ressemble complètement à un autre dépôt Git. Il y a trois branches et Git a utilement créé une branche `master` locale qui suit la branche `origin/master`. Travaillons un peu et créons une paire de *commits* :

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the Intel one
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Nous avons deux nouveaux *commits*. Maintenant, vérifions si quelqu'un d'autre a aussi travaillé :

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Il semble bien ! Ça n'apparaît pas sur cette vue, mais le *commit* `6afeb15` a en fait été créé en utilisant un client Perforce. Il ressemble juste à un commit normal du point de vue de Git, ce qui est exactement l'effet recherché. Voyons comment le serveur Perforce gère le *commit* de fusion :

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git pense que ça a marché. Voyons l'historique du fichier `README` du point de vue de Perforce, en utilisant la fonctionnalité de graphe de révision de `p4v` :

FIGURE 9-2

Graphe de révision de Perforce résultant d'une poussée depuis Git.

Si vous n'avez jamais vu ceci auparavant, cela peut dérouter, mais c'est une vue similaire à la vue graphique de l'historique Git. Nous visualisons l'historique du fichier `README`, donc l'arbre de répertoire en haut à gauche ne montre que ce fichier, aux endroits où il apparaît dans différentes branches. En haut à droite, nous avons le graphe visuel des relations entre les différentes révisions du fichier et la vue en grand du graphe en bas à droite. Le reste de l'écran concerne la visualisation des détails pour la révision sélectionnée (2 dans ce cas).

Une chose à noter est que le graphe ressemble exactement à celui de l'historique Git. Perforce n'avait pas de branche nommée pour stocker les *commits* 1 et 2, il a donc créé une branche « anonymous » dans le répertoire `.git-fusion` pour la gérer. Cela arrivera aussi pour des branches Git nommées qui ne correspondent pas à une branche Perforce nommée (et que vous pouvez plus tard faire correspondre à une branche Perforce en utilisant le fichier de configuration).

Tout ceci se passe en coulisse, mais le résultat final est qu'une personne dans l'équipe peut utiliser Git, une autre Perforce et aucune des deux n'a à se soucier du choix de l'autre.

Résumé Git-Fusion

Si vous avez accès (ou pouvez avoir accès) à un votre serveur Perforce, Git Fusion est un excellent moyen de faire parler Git et Perforce ensemble. Cela nécessite un peu de configuration, mais la courbe d'apprentissage n'est pas très raide. C'est une des rares sections de ce chapitre où il est inutile de faire spécifiquement attention à ne pas utiliser toute la puissance de Git. Cela ne signifie pas que Perforce sera ravi de tout ce que vous lui enverrez — si vous réécrivez l'historique qui a déjà été poussé, Git Fusion va le rejeter — Git Fusion cherche vraiment à sembler naturel. Vous pouvez même utiliser les sous-modules Git (bien qu'ils paraîtront étranges pour les utilisateurs Perforce), et fusionner les branches (ce qui sera enregistré comme une intégration du côté Perforce).

Si vous ne pouvez pas convaincre un administrateur de votre serveur d'installer Git Fusion, il existe encore un moyen d'utiliser ces outils ensemble.

GIT-P4

Git-p4 est une passerelle à double sens entre Git et Perforce. Il fonctionne intégralement au sein de votre dépôt Git, donc vous n'avez besoin d'aucun accès au serveur Perforce (autre que les autorisations d'utilisateur, bien sûr). Git-p4 n'est pas une solution aussi flexible ou complète que Git Fusion, mais il permet tout de même de réaliser la plupart des activités sans être invasif dans l'environnement serveur.

Vous aurez besoin de l'outil `p4` dans votre de chemin de recherche pour travailler avec `git-p4`. À la date d'écriture du livre, il est disponible à *http://www.perforce.com/downloads/Perforce/20-User*.

Installation

Pour l'exemple, nous allons lancer le serveur Perforce depuis l'image Git Fusion, comme indiqué ci-dessus, mais nous n'utiliserons pas le serveur Git Fusion et nous dialoguerons avec la gestion de version Perforce directement.

Pour utiliser le client p4 en ligne de commande (dont `git-p4` dépend), vous devrez définir quelques variables d'environnement :

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Démarrage

Comme d'habitude avec Git, la première commande est un clonage :

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into refs/remotes/p4
```

Cela crée ce qui en parlé Git s'appelle un clone « superficiel » (**shallow**) ; seule la toute dernière révision Perforce est importée dans Git ; souvenez-vous que Perforce n'a pas été pensé pour fournir toutes les révisions à chaque utilisateur. C'est suffisant pour utiliser Git comme client Perforce, mais pour d'autres utilisations, ce n'est pas assez.

Une fois que c'est terminé, nous avons un dépôt Git complètement fonctionnel.

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ f
```

Notez le dépôt p4 distant pour le serveur Perforce, mais tout le reste ressemble à un clone standard. En fait, c'est trompeur ; ce n'est pas réellement dépôt distant.

```
$ git remote -v
```

Il n'y a pas du tout de dépôt distant. Git-p4 a créé des références qui représentent l'état du serveur et celles-ci ressemblent à des références de dépôts distants dans `git log`, mais elles ne sont pas gérées par Git lui-même et vous ne pouvez pas pousser dessus.

Utilisation

Donc, travaillons un peu. Supposons que vous avez progressé sur une fonctionnalité très importante et que vous êtes prêt à la montrer au reste de votre équipe.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at revisio
```

Nous avons réalisé deux nouveaux *commits* qui sont prêts à être soumis au serveur Perforce. Vérifions si quelqu'un d'autre a poussé son travail entre temps.

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Il semblerait que ce soit le cas, et `master` et `p4/master` ont divergé. Le système de branchement de Perforce ne ressemble en rien à celui de Git, donc soumettre des *commits* de fusion n'a aucun sens. Git-p4 recommande de rebaser vos *commits* et fournit même un raccourci pour le faire :

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Vous pouvez déjà le deviner aux messages affichés, mais `git p4 rebase` est un raccourci pour `git p4 sync` suivi de `git rebase p4/master`. C'est légèrement plus intelligent que cela, spécifiquement lors de la gestion de branches multiples, mais ça correspond bien.

À présent, notre historique est linéaire à nouveau et nous sommes prêts à remonter nos modifications sur Perforce. La commande `git p4 submit` va essayer de créer une nouvelle révision Perforce pour chaque *commit* Git entre `p4/master` et `master`. Son lancement ouvre notre éditeur favori et le contenu du fichier ouvert ressemble à ceci :

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists only.)
#  Files:       What opened files from the default changelist are to be added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html   2
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
```

```
 Jam/MR</a>,
 a software build tool.
 </td>
```

C'est quasiment le même contenu qu'on verrait en lançant `p4 submit`, mis à part le bloc à la fin que git-p4 a utilement inclus. Git-p4 essaye d'honorer vos réglages Git et Perforce individuellement quand il doit fournir un nom pour un *commit* ou une modification, mais dans certains cas, vous voudrez le modifier. Par exemple, si le *commit* Git que vous importez a été écrit par un contributeur qui n'a pas de compte utilisateur dans Perforce, vous voudrez tout de même que la modification résultante ait l'air d'avoir été écrite par lui, et non par vous.

Git-p4 a importé le message du *commit* Git comme contenu de la modification Perforce, donc tout ce qu'il nous reste à faire et de sauvegarder et de quitter, deux fois (une fois par *commit*). La sortie qui en résulte ressemble à ceci :

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

À la sortie, c'est comme si nous avions fait un `git push`, ce qui est l'analogie la plus proche avec ce qui s'est réellement passé.

Notez aussi que durant ce processus, les *commits* Git sont transformés en modifications Perforce ; si vous voulez les comprimer en une seule modification, vous pouvez le faire avec un rebasage interactif avant de lancer `git p4 submit`. Notez aussi que les empreintes SHA-1 de tous les *commits* qui ont été soumis comme modifications ont changé ; c'est parce que git-p4 ajoute une ligne à la fin de chaque message de validation qu'il convertit :

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

Que se passe-t-il si vous essayez de soumettre un *commit* de fusion ? Essayons pour voir. Voici la situation dans laquelle nous sommes :

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Les historiques Git et Perforce divergent à partir de 775a46f. Le côté Git contient deux *commits*, puis un *commit* de fusion avec la tête Perforce, puis un autre *commit*. Nous allons essayer de les soumettre au sommet d'un seul changement côté Perforce. Voyons ce qui se passerait si nous essayions de soumettre cet historique maintenant :

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/
```

```
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

L'option `-n` est un raccourci pour `--dry-run` qui essaie d'afficher ce qui se passerait si la commande `submit` était réellement lancée. Dans ce cas, il semble que nous créerions trois modifications Perforce, ce qui correspond aux trois *commits* sans fusion qui n'existent pas encore sur le serveur Perforce. Cela ressemble exactement à ce que nous souhaitons, voyons comment cela se passe :

```
$ git p4 submit
[…]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Notre historique est devenu linéaire, comme si nous avions rebasé avant de soumettre (ce qui est exactement ce qui s'est passé). Cela signifie que vous êtes libre de créer, modifier, jeter et fusionner les branches du côté Git sans crainte que votre historique deviennent à un moment incompatible avec Perforce. Si vous pouvez le rebaser, vous pourrez le reporter dans le serveur Perforce.

Branche

Si votre projet Perforce a de multiples branches, vous n'êtes pas malchanceux ; `git-p4` peut gérer cette configuration d'une manière similaire à Git. Supposons que votre dépôt Perforce ait la forme suivante :

```
//depot
  └── project
      ├── main
      └── dev
```

Et supposons que vous ayez une branche `dev` qui contient une *view spec* qui ressemble à ceci :

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 peut détecter automatiquement cette situation et faire ce qu'il faut :

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Notez le déterminant « `@all` » ; il indique à `git-p4` de cloner non seulement la dernière modification pour ce sous-arbre, mais aussi toutes les modifications qui ont déjà touché à ces chemins. C'est plus proche du concept de clone dans Git, mais si vous travaillez sur un projet avec un long historique, cela peut prendre du temps à se terminer.

L'option `--detect-branches` indique à `git-p4` d'utiliser les spécifications de branche de Perforce pour faire correspondre aux références Git. Si ces correspondances ne sont pas présentes sur le serveur Perforce (ce qui est une manière tout à fait valide d'utiliser Perforce), vous pouvez dire à `git-p4` ce que sont les correspondances de branches, et vous obtiendrez le même résultat :

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Renseigner la variable de configuration `git-p4.branchList` à `main:dev` indique à `git-p4` que `main` et `dev` sont toutes deux des branches et que la seconde est la fille de la première.

Si nous lançons maintenant `git checkout -b dev p4/project/dev` et ajoutons quelques *commits*, git-p4 est assez intelligent pour cibler la bonne branche quand nous lançons `git-p4 submit`. Malheureusement, `git-p4` ne peut pas mélanger les clones superficiels et les branches multiples ; si vous avez un projet gigantesque et que vous voulez travailler sur plus d'une branche,

vous devrez lancer `git p4 clone` une fois pour chaque branche à laquelle vous souhaitez soumettre.

Pour créer ou intégrer des branches, vous devrez utiliser un client Perforce. Git-p4 ne peut synchroniser et soumettre que sur des branches préexistantes, et il ne peut le faire qu'avec une modification linéaire à la fois. Si vous fusionnez deux branches dans Git et que vous essayez de soumettre la nouvelle modification, tout ce qui sera enregistré sera une série de modifications de fichiers ; les métadonnées relatives aux branches impliquées dans cette intégration seront perdues.

RÉSUMÉ GIT ET PERFORCE

Git-p4 rend possible l'usage des modes d'utilisation de Git avec un serveur Perforce, et ce, de manière plutôt réussie. Cependant, il est important de se souvenir que Perforce gère les sources et qu'on ne travaille avec Git que localement. Il faut rester vraiment attentif au partage de *commits* Git ; si vous avez un dépôt distant que d'autres personnes utilisent, ne poussez aucun *commit* qui n'a pas déjà été soumis au serveur Perforce.

Si vous souhaitez mélanger l'utilisation de Git et de Perforce comme clients pour la gestion de source sans restriction et si vous arrivez à convaincre un administrateur de l'installer, Git Fusion fait de Git un client de premier choix pour un serveur Perforce.

Git et TFS

Git est en train de devenir populaire chez les développeurs Windows et si vous écrivez du code pour Windows, il y a de fortes chances que vous utilisiez *Team Foundation Server* (TFS) de Microsoft. TFS est une suite collaborative qui inclut le suivi de tickets et de tâches, le support de modes de développement Scrum et autres, la revue de code et la gestion de version. Pour éviter toute confusion ultérieure, **TFS** est en fait le serveur, qui supporte la gestion de version de sources en utilisant à la fois Git et son propre gestionnaire de version, appelé **TFVC** (*Team Fundation Version Control*). Le support de Git est une fonctionnalité assez nouvelle pour TFS (introduite dans la version 2013), donc tous les outils plus anciens font référence à la partie gestion de version comme « TFS », même s'ils ne fonctionnent réellement qu'avec TFVC.

Si vous vous trouvez au sein d'une équipe qui utilise TFVC mais que vous préférez utiliser Git comme client de gestionnaire de version, il y a un projet pour votre cas.

QUEL OUTIL

En fait, il y en a deux : git-tf et git-tfs.

Git-tfs (qu'on peut trouver à ***http://git-tfs.com***) est un projet .NET et ne fonctionne que sous Windows (à l'heure de la rédaction du livre). Pour travailler avec des dépôts Git, il utilise les liaisons .NET pour libgit2, une implémentation de Git orientée bibliothèque, qui est très performante et qui permet de manipuler avec beaucoup de flexibilité un dépôt Git à bas niveau. Libgit2 n'est pas une implantation complète de Git, donc pour couvrir la différence, git-tfs va en fait appeler directement le client Git en ligne de commande pour certaines opérations de manière à éliminer les limites artificielles de ce qui est réalisable sur des dépôts Git. Son support des fonctionnalités de TFVC est très mature, puisqu'il utilise les assemblages de Visual Studio pour les opérations avec les serveurs. Cela implique que vous devez avoir accès à ces assemblages, ce qui signifie que vous devez installer une version récente de Visual Studio (n'importe quelle version depuis la version 2010, y compris la version Express depuis la version 2012), ou le SDK (Software Development Kit) Visual Studio.

Git-tf (dont le site est ***https://gittfs.codeplex.com***) est un projet Java et en tant que tel peut fonctionner sur tout ordinateur supportant l'environnement d'exécution Java. Il s'interface avec les dépôts Git à travers JGit (une implantation sur JVM de Git), ce qui signifie qu'il n'y a virtuellement aucune limitation en termes de fonctionnalités Git. Cependant, le support pour TFVC est plus limité comparé à git-tfs - il ne supporte pas les branches par exemple.

Donc chaque outil a ses avantages et ses défauts, et de nombreuses situations favorisent l'un par rapport à l'autre. Nous décrirons l'utilisation de base de chaque outil.

Vous aurez besoin d'un accès à un dépôt TFVC pour pouvoir suivre les instructions qui vont suivre. Il n'y en a pas beaucoup disponibles sur internet comme Git ou Subversion, et il se peut que vous deviez en créer un par vous-même. Codeplex (***https://www.codeplex.com***) ou Visual Studio Online (***http://www.visualstudio.com***) sont tous deux de bons choix.

DÉMARRAGE : `GIT-TF`

La première chose à faire, comme toujours avec Git, c'est de cloner. Voici à quoi cela ressemble avec `git-tf` :

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

Le premier argument est l'URL de la collection TFVC, le deuxième est de la forme `/projet/branche` et le troisième est le chemin vers le dépôt local Git à créer (celui-ci est optionnel). Git-tf ne peut fonctionner qu'avec une branche à la fois ; si vous voulez valider sur une branche TFVC différente, vous devrez faire un nouveau clone de cette branche.

Cela crée un dépôt Git complètement fonctionnel :

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Ceci s'appelle un clone *superficiel*, ce qui signifie que seule la dernière révision a été téléchargée. TFVC n'est pas conçu pour que chaque client ait une copie complète de l'historique, donc git-tf ne récupère que la dernière révision par défaut, ce qui est plus rapide.

Si vous avez du temps, il vaut peut-être le coup de cloner l'intégralité de l'historique du projet, en utilisant l'option `--deep` :

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
        Team Project Creation Wizard
```

Remarquez les étiquettes comprenant des noms tels que `TFS_C35189` ; c'est une fonctionnalité qui vous aide à reconnaître quels *commits* Git sont associés à des modifications TFVC. C'est une façon élégante de les représenter, puisque vous pouvez voir avec une simple commande `log` quels *commits* sont associés avec un instantané qui existe aussi dans TFVC. Elles ne sont pas nécessaires (en fait, on peut les désactiver avec `git config git-tf.tag false`) – git-tf conserve les correspondances *commit*-modification dans le fichier `.git/git-tf`.

DÉMARRAGE : `GIT-TFS`

Le clonage via Git-tfs se comporte légèrement différemment. Observons :

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeefb961958b674
```

Notez l'option `--with-branches`. Git-tfs est capable de faire correspondre les branches de TFVC et Git, et cette option indique de créer une branche Git locale pour chaque branche TFVC. C'est hautement recommandé si vous avez déjà fait des branches et des fusions dans TFS, mais cela ne fonctionnera pas avec un serveur plus ancien que TFS 2010 – avant cette version, les « branches » n'étaient que des répertoires et git-tfs ne peut pas les différencier de répertoires normaux.

Visitons le dépôt Git résultant :

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id: [https://username.visualstudio.com/DefaultCollection]$/myproject/Tru
```

Il y a deux branches locales, `master` et `featureA`, ce qui correspond au point de départ du clone (`Trunk` dans TFVC) et à une branche enfant (`featureA` dans TFVC). Vous pouvez voir que le « dépôt distant » `tfs` contient aussi des références : `default` et `featureA` qui représentent les branches TFVC. Git-tfs fait correspondre la branche que vous avez clonée depuis `tfs/default`, et les autres récupèrent leur propre nom.

Une autre chose à noter concerne les lignes `git-tfs-id:` dans les messages de validation. Au lieu d'étiquettes, git-tfs utilise ces marqueurs pour faire le lien entre les modifications TFVC et les *commits* Git. Cela implique que les *commits* Git vont avoir une empreinte SHA-1 différente entre avant et après avoir été poussés sur TFVC.

TRAVAIL AVEC GIT-TF[S]

Indépendamment de chaque outil que vous utilisez, vous devriez renseigner quelques paramètres de configuration Git pour éviter les ennuis.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

Evidemment, vous souhaitez ensuite travailler sur le projet. TFVC et TFS ont des caractéristiques qui peuvent complexifier votre travail :

1. Les branches thématiques qui ne sont pas représentées dans TFVC ajoutent un peu de complexité. Cela est dû à la manière **très** différente dont TFVC et Git représentent les branches.
2. Soyez conscient que TFVC permet aux utilisateurs d'« extraire » des fichiers depuis le serveur en les verrouillant pour qu'aucun autre utilisateur ne puisse les éditer. Cela ne vous empêchera évidemment pas de les éditer dans votre dépôt local, mais cela pourrait être un obstacle au moment de pousser vos modifications sur le serveur TFVC.
3. TFS a le concept de validations « gardées », où un cycle de compilation/test TFS doit se terminer avec succès pour que la validation soit acceptée. Cela utilise la fonction « enterrement » (*shelve*) dans TFVC, que nous ne détaillons pas en détail ici. Vous pouvez simuler ceci manuellement avec git-tf, et git-tfs fournit la commande `checkintool` qui connaît le concept de garde.

Pour abréger, nous n'allons traiter que le cas sans erreur, qui contourne ou évite quasiment tous les problèmes.

TRAVAIL AVEC `GIT-TF`

Supposons que vous ayez travaillé et validé quelques *commits* sur `master` et que vous êtes prêt à partager votre progression sur le serveur TFVC. Voici notre dépôt Git :

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Nous voulons prendre l'instantané qui est dans le *commit* 4178a82 et le pousser sur le serveur TFVC.

Tout d'abord, vérifions si un de nos collègues a fait quelque chose depuis notre dernière connexion :

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Il semble que c'est le cas et nous avons maintenant un historique divergent. C'est là où Git brille, mais nous avons deux options :

1. Faire un *commit* de fusion semble naturel pour un utilisateur Git (après tout, c'est ce que `git pull` réalise), et git-tf peut faire cela pour vous avec un simple `git tf pull`. Gardez cependant à l'esprit que TFVC n'est pas conçu de cette manière et si vous poussez des *commits* de fusion, votre historique va commencer à être différent entre les deux côtés, ce qui peut être déroutant. Cependant, si vous comptez soumettre tout votre travail comme une modification unique, c'est sûrement le choix le plus simple.
2. Rebaser pour rendre votre historique linéaire, ce qui signifie que nous avons l'option de convertir chaque *commit* Git en modification TFVC.

Comme c'est l'option qui laisse le plus de possibilités ouvertes, c'est la méthode recommandée ; `git-tf` facilite même cette méthode avec la commande `` `git tf pull --rebase`.

Le choix reste le vôtre. Pour cet exemple, nous rebaserons :

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

À présent, nous sommes prêt à valider dans le serveur TFVC. Git-tf vous laisse le choix de faire un changement unique qui représente toutes les modifications depuis le dernier réalisé (`--shallow`, par défaut) ou de créer une nouvelle modification pour chaque *commit* Git (`--deep`). Pour cet exemple, nous allons créer une modification unique :

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Il y a une nouvelle étiquette `TFS_C35348` qui indique que TFVC stocke le même instantané que le *commit* `5a0e25e`. Il est important de noter que chaque

commit Git n'a pas besoin d'avoir une contrepartie exacte dans TFVC ; le *commit* `6eb3eb5`, par exemple, n'existe pas sur le serveur.

C'est le style de gestion principal. Gardez en tête les quelques autres aspects de cette utilisation :

- Il est impossible d'utiliser les branches. Git-tf ne peut créer des dépôts Git qu'à partir d'une branche de TFVC à la fois.
- Le serveur central est TFVC ou Git, pas les deux. Différents clones git-tf du même dépôt TFVC peuvent avoir des empreintes SHA-1 différentes, ce qui sera un casse-tête sans fin.
- Si la gestion dans votre équipe consiste à collaborer par Git et à synchroniser périodiquement avec TFVC, ne connectez TFVC qu'à un seul dépôt Git.

TRAVAILLER AVEC `GIT-TFS`

Déroulons le même scénario en utilisant `git-tfs`. Voici les nouveaux *commits* que nous avons ajoutés à la branche `master` dans notre dépôt Git :

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

Maintenant, voyons si quelqu'un a avancé pendant que nous travaillions de notre côté :

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Oui, un collègue a ajouté une nouvelle modification TFVC, qui prend la forme du nouveau *commit* `aea74a0`, ce qui a fait avancer la branche `tfs/default`.

De la même manière qu'avec `git-tf`, nous avons deux options pour résoudre l'historique divergent :

1. Rebaser pour préserver l'historique linéaire,
2. fusionner pour préserver ce qui s'est réellement passé.

Dans cet exemple, nous allons réaliser une validation « profonde » où chaque *commit* Git devient une modification TFVC, ce qui implique que nous rebasions.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Nous voici prêts à réintégrer notre code dans le serveur TFVC. Nous utiliserons la commande `rcheckin` pour créer des modifications TFVC pour chaque *commit* Git dans le parcours entre HEAD et la première branche distante `tfs` trouvée (la commande `checkin` ne créerait qu'une modification, comme si on compressait tous les *commits*).

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-commit...
Rebase done successfully.
No more to rcheckin.
```

```
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Remarquez comment après chaque enregistrement réussi dans le serveur TFVC, git-tfs rebase le travail restant sur ce qui vient d'être intégré. C'est dû à l'addition du champ `git-tfs-id` au bas du message de validation, qui modifie l'empreinte SHA-1 du *commit* dernièrement enregistré. Cela se passe comme prévu et il n'y a pas lieu de s'en inquiéter, mais il faut garder à l'esprit cette transformation, spécialement si vous partagez des *commits* Git avec d'autres développeurs.

TFS a de nombreuses fonctionnalités intégrées avec le système de gestion de version, telles que les tâches, les revues, les enregistrements gardés, etc. Travailler avec ces fonctionnalités à partir de la ligne de commande peut être lourd mais heureusement, git-tfs permet de lancer très facilement un outil d'enregistrement graphique :

```
PS> git tfs checkintool
PS> git tfs ct
```

L'outil ressemble à ceci :

FIGURE 9-3

L'outil d'enregistrement git-tfs.

Les utilisateurs de TFS le connaissent, puisque c'est la même boîte de dialogue que celle lancée depuis Visual Studio.

Git-tfs vous laisse aussi gérer vos branches TFVC depuis votre dépôt Git. Par exemple, nous allons en créer une :

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git lga
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Créer une branche dans TFVC signifie ajouter une modification où cette branche existe à présent, ce qui se traduit par un *commit* Git. Notez aussi que git-tfs a **créé** la branche distante `tfs/featureBee`, mais HEAD pointe toujours sur `master`. Si vous voulez travailler sur la toute nouvelle branche, vous souhaiterez baser vos nouveaux *commits* sur `1d54865`, peut-être en créant une branche thématique sur ce *commit*.

RÉSUMÉ GIT ET TFS

Git-tf et Git-tfs sont tous deux des grands outils pour s'interfacer avec un serveur TFVC. Ils vous permettent d'utiliser la puissance de Git localement, vous évitant d'avoir sans arrêt à faire des aller-retours avec le serveur central TFVC et simplifie votre vie de développeur, sans forcer toute l'équipe à passer sous Git. Si vous travaillez sous Windows (ce qui est très probable si votre équipe utilise TFS), vous souhaiterez utiliser git-tfs car ses fonctionnalités sont les plus complètes, mais si vous travaillez avec une autre plate-forme, vous utiliserez git-tf qui est plus limité. Comme avec la plupart des outils vus dans ce chapitre, vous avez intérêt à vous standardiser sur un de ces systèmes de gestion de version et à utiliser l'autre comme miroir – soit Git, soit TFVC doivent être le centre de collaboration, pas les deux.

Migration vers Git

Si vous avez une base de code existant dans un autre Système de Contrôle de Version (SCV) mais que vous avez décidé de commencer à utiliser Git, vous devez migrer votre projet d'une manière ou d'une autre. Cette section passe en revue quelques importateurs pour des systèmes communs, et ensuite démontre comment développer votre propre importateur personnalisé. Vous apprendrez comment importer les données depuis plusieurs des plus gros systèmes de gestion de configuration logicielle (*SCM*, *Software Configuration Management*) utilisés professionnellement, parce qu'ils comportent la majorité des utilisateurs qui basculent, et parce que des outils de haute qualité dédiés sont faciles à se procurer.

Subversion

Si vous avez lu la section précédente concernant l'utilisation de `git svn`, vous pouvez utiliser facilement ces instructions pour `git svn clone` un dépôt ; ensuite, vous pouvez arrêter d'utiliser le serveur Subversion, pousser vers un nouveau serveur Git, et commencer à l'utiliser. Si vous voulez l'historique, vous pouvez obtenir cela aussi rapidement que vous pouvez tirer les données hors du serveur Subversion (ce qui peut prendre un bout de temps).

Cependant, l'import n'est pas parfait ; et comme ça prendra tant de temps, autant le faire correctement. Le premier problème est l'information d'auteur. Dans Subversion, chaque personne qui crée un *commit* a un utilisateur sur le système qui est enregistré dans l'information de *commit*. Les exemples dans la section précédente montrent `schacon` à quelques endroits, comme la sortie de `blame` et `git svn log`. Si vous voulez faire correspondre cela à une meilleure donnée d'auteur Git, vous avez besoin d'une transposition des utilisateurs Subversion vers les auteurs Git. Créez un fichier appelé `users.txt` qui a cette correspondance dans un format tel que celui-ci :

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Pour obtenir une liste des noms d'auteur que SVN utilise, vous pouvez lancer ceci :

```
$ svn log --xml | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

Cela génère la sortie log dans le format XML, puis garde seulement les lignes avec l'information d'auteur, rejette les doublons, enlève les étiquettes XML. (Bien sûr, cela ne marche que sur une machine ayant `grep`, `sort` et `perl` installés.) Ensuite, redirigez cette sortie dans votre fichier users.txt afin que vous puissiez ajouter l'information d'utilisateur Git équivalente près de chaque entrée.

Vous pouvez fournir ce fichier à `git svn` pour l'aider à faire correspondre la donnée d'auteur plus précisément. Vous pouvez aussi demander à `git svn` de ne pas inclure les metadonnées que Subversion importe normalement, en passant `--no-metadata` à la commande `clone` ou `init`. Ceci fait ressembler votre commande `import` à ceci :

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

Maintenant vous devriez avoir un import Subversion plus joli dans votre dossier `my_project`. Au lieu de *commits* qui ressemblent à ceci

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

ils ressemblent à ceci :

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Non seulement le champ Auteur a l'air beaucoup mieux, mais le `git-svn-id` n'est plus là non plus.

Vous devriez aussi faire un peu de ménage post-import. D'abord, vous devriez nettoyer les références bizarres que `git svn` a installées. Premièrement vous déplacerez les étiquettes afin qu'elles soient de véritables étiquettes plutôt que d'étranges branches distantes, et ensuite vous déplacerez le reste des branches afin qu'elles soient locales.

Pour déplacer les étiquettes pour qu'elles soient des étiquettes Git propres, lancez

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

Ceci prend les références qui étaient des branches distantes qui commençaient par `remotes/origin/tags` et en fait de vraies étiquettes (légères).

Ensuite, déplacez le reste des références sous `refs/remotes` pour qu'elles soient des branches locales :

```
$ cp -Rf .git/refs/remotes/origin/* .git/refs/heads/
$ rm -Rf .git/refs/remotes/origin
```

Il peut arriver que vous voyiez quelques autres branches qui sont suffixées par `@xxx` (où xxx est un nombre), alors que dans Subversion vous ne voyez qu'une seule branche. C'est en fait une fonctionnalité Subversion appelée « peg-revisions », qui est quelque chose pour laquelle Git n'a tout simplement pas d'équivalent syntaxique. Donc, `git svn` ajoute simplement le numéro de version svn au nom de la branche de la même façon que vous l'auriez écrit dans svn pour adresser la « peg-revision » de cette branche. Si vous ne vous souciez plus des « peg-revisions », supprimez-les simplement en utilisant `git branch -d`.

Maintenant toutes les vieilles branches sont de vraies branches Git et toutes les vieilles étiquettes sont de vraies étiquettes Git.

Il y a une dernière chose à nettoyer. Malheureusement, `git svn` crée une branche supplémentaire appelée `trunk`, qui correspond à la branche par défaut de Subversion, mais la ref `trunk` pointe au même endroit que `master`. Comme `master` est plus idiomatiquement Git, voici comment supprimer la branche supplémentaire :

```
$ git branch -d trunk
```

La dernière chose à faire est d'ajouter votre nouveau serveur Git en tant que serveur distant et pousser vers lui. Voici un exemple d'ajout de votre serveur en tant que serveur distant :

```
$ git remote add origin git@my-git-server:myrepository.git
```

Puisque vous voulez que vos branches et étiquettes montent, vous pouvez maintenant lancer :

```
$ git push origin --all
$ git push origin --tags
```

Toutes vos branches et étiquettes devraient être sur votre nouveau serveur Git dans un import joli et propre.

Mercurial

Puisque Mercurial et Git ont des modèles assez similaires pour représenter les versions, et puisque Git est un peu plus flexible, convertir un dépôt depuis Mercurial vers Git est assez simple, en utilisant un outil appelé “hg-fast-export”, duquel vous aurez besoin d’une copie :

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

La première étape dans la conversion est d’obtenir un clone complet du dépôt Mercurial que vous voulez convertir :

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

L’étape suivante est de créer un fichier d’association d’auteur. Mercurial est un peu plus indulgent que Git pour ce qu’il mettra dans le champ auteur pour les modifications, donc c’est le bon moment pour faire le ménage. La génération de ceci tient en une ligne de commande dans un shell `bash` :

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Cela prendra quelques secondes, en fonction de la longueur de l’historique de votre projet, et ensuite le fichier `/tmp/authors` ressemblera à quelque chose comme ceci :

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
```

```
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

Dans cet exemple, la même personne (Bob) a créé des modifications sous différents noms, dont l'un est correct, et dont un autre est complètement invalide pour un *commit* Git. Hg-fast-import nous laisse régler cela en ajoutant `={nouveau nom et adresse de courriel}` à la fin de chaque ligne que l'on veut changer, et en enlevant les lignes pour les noms d'utilisateur auxquels on ne veut pas toucher. Si tous les noms d'utilisateur ont l'air bien, nous n'aurons pas du tout besoin de ce fichier. Dans cet exemple, nous voulons que notre fichier ressemble à cela :

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

L'étape suivante consiste à créer notre nouveau dépôt Git, et à lancer le script d'export :

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

L'option `-r` indique à hg-fast-export où trouver le dépôt Mercurial que l'on veut convertir, et l'option `-A` lui indique où trouver le fichier de correspondance d'auteur. Le script analyse les modifications Mercurial et les convertit en un script pour la fonctionnalité "fast-import" de Git (que nous détaillerons un peu plus tard). Cela prend un peu de temps (bien que ce soit *beaucoup plus* rapide que si c'était à travers le réseau), et la sortie est assez verbeuse :

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed f
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed f
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/remov
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/remov
master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/r
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
```

```
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602 attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599 attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0 attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

C'est à peu près tout ce qu'il y a. Toutes les étiquettes Mercurial ont été converties en étiquettes Git, et les branches et marques-page Mercurial ont été convertis en branches Git. Maintenant vous êtes prêt à pousser le dépôt vers son nouveau serveur d'hébergement :

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Perforce

Le système suivant dont vous allez voir l'importation est Perforce. Ainsi que nous l'avons dit plus haut, il y a deux façons de permettre de faire parler Git et Perforce l'un avec l'autre : git-p4 et Perforce Git Fusion.

PERFORCE GIT FUSION

Git Fusion rend ce processus assez indolore. Configurez les paramètres de votre projet, les correspondances utilisateur et les branches en utilisant un fichier de configuration (comme discuté dans **“Git Fusion”**), et clonez le dépôt. Git Fusion vous laisse avec ce qui ressemble à un dépôt Git natif, qui est alors prêt à être poussé vers un hôte Git natif si vous le désirez. Vous pouvez même utiliser Perforce comme hôte Git si vous ça vous plaît.

GIT-P4

Git-p4 peut aussi agir comme outil d’import. Comme exemple, nous importerons le projet Jam depuis le Dépôt Public Perforce. Pour définir votre client, vous devez exporter la variable d’environnement P4PORT pour pointer vers le dépôt Perforce :

```
$ export P4PORT=public.perforce.com:1666
```

> Pour suivre tout le long, vous aurez besoin d’un dépôt Perforce auquel vous connecter. Nous utiliserons le dépôt public à public.perforce.com pour nos exemples, mais vous pouvez utiliser n’importe quel dépôt auquel vous avez accès.

Lancez la commande `git p4 clone` pour importer le projet Jam depuis le serveur Perforce, en fournissant le chemin vers le dépôt et le projet dans lequel vous voulez importer le projet :

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

Ce projet particulier a seulement une branche, mais si vous avez des branches configurées avec des vues de branche (ou juste un ensemble de dossiers), vous pouvez utiliser l’option `--detect-branches` avec `git p4 clone` pour importer aussi toutes les branches du projet. Voyez **“Branche”** pour plus de détails sur ceci.

A ce point, vous avez presque terminé. Si vous allez dans le dossier `p4import` et lancez `git log`, vous pouvez voir le travail importé :

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Vous pouvez voir que `git-p4` a laissé un identifiant dans chaque message de *commit*. C'est bien de garder cet identifiant-là, au cas où vous auriez besoin de référencer le numéro de changement Perforce plus tard. Cependant, si vous souhaitez enlever l'identifiant, c'est maintenant le moment de le faire – avant que vous ne commenciez à travailler sur le nouveau dépôt. Vous pouvez utiliser `git filter-branch` pour enlever en masse les chaînes d'identifiant :

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Si vous lancez `git log`, vous pouvez voir que toutes les sommes de vérification SHA-1 pour les *commits* ont changé, mais les chaînes `git-p4` ne sont plus dans les messages de *commit* :

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800
```

```
    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

Votre import est prêt à être poussé vers votre nouveau serveur Git.

TFS

Si votre équipe est en train de convertir son code source de TFVC à Git, vous voudrez la conversion de la plus haute fidélité que vous puissiez obtenir. Cela signifie que, tandis que nous couvrions à la fois git-tfs et git-tf pour la section interop, nous couvrirons seulement git-tfs dans cette partie, parce que git-tfs supporte les branches, et c'est excessivement difficile en utilisant git-tf.

> Ceci est une conversion à sens unique. Le dépôt Git résultant ne pourra pas se connecter au projet TFVC original.

La première chose à faire est d'associer les noms d'utilisateur. TFC est assez permissif pour ce qui va dans le champ auteur pour les changements, mais Git veut un nom et une adresse de courriel lisibles par un humain. Vous pouvez obtenir cette information depuis la ligne de commande client `tf`, comme ceci :

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

Cela récupère toutes les modifications de l'historique du projet et les insère dans le fichier AUTHORS_TMP que nous traiterons pour en extraire la donnée de la colonne *Utilisateur* (la deuxième). Ouvrez le fichier et trouvez à quels caractères commence et finit la colonne et remplacez, dans la ligne de commande suivante, les paramètres `11-20` de la commande `cut` par ceux que vous avez trouvés :

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

La commande `cut` ignore tout sauf les caractères 11-20 de chaque ligne. La commande `tail` saute les deux premières lignes, qui sont des champs d'en-tête et des soulignés dans le style ASCII. Le résultat de tout ceci est envoyé à `uniq` pour éliminer les doublons, et sauvé dans un fichier nommé `AUTHORS`. L'étape suivante est manuelle ; afin que git-tfs fasse un usage effectif de ce fichier, chaque ligne doit être dans ce format :

```
DOMAIN\username = User Name <email@address.com>
```

La partie gauche est le champ “utilisateur” de TFVC, et la partie droite du signe égal est le nom d’utilisateur qui sera utilisé pour les *commits* Git.

Une fois que vous avez ce fichier, la chose suivante à faire est de faire un clone complet du projet TFVC par lequel vous êtes intéressé :

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.com/DefaultC
```

Ensuite vous voudrez nettoyer les sections `git-tfs-id` du bas des messages de commit. La commande suivante le fera :

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' '--' --all
```

Cela utilise la commande `sed` de l’environnement Git-bash pour remplacer n’importe quelle ligne commençant par “git-tfs-id:” par du vide que Git ignorera ensuite.

Une fois cela fait, vous êtes prêt à ajouter un nouveau serveur distant, y pousser toutes les branches, et vous avez votre équipe prête à commencer à travailler depuis Git.

Un importateur personnalisé

Si votre système n’est pas un de ceux ci-dessus, vous devriez chercher un importateur en ligne – des importateurs de qualité sont disponibles pour plein d’autres systèmes, incluant CVS, Clear Case, Visual Source Safe, même un dossier d’archives. Si aucun de ces outils ne fonctionne pour vous, vous avez un outil plus obscur, ou alors vous avez besoin d’un procédé d’importation personnalisé, vous devriez utiliser `git fast-import`. Cette commande lit des instructions simples depuis l’entrée standard pour écrire des données Git spécifiques. Il est bien plus facile de créer des objets Git de cette façon que de lancer des commandes Git brutes ou que d’essayer d’écrire les objets bruts (voir **Chapter 10** pour plus d’informations). De cette façon, vous pouvez écrire un script d’importation qui lit l’information nécessaire hors du système duquel vous importez et qui affiche les instructions directement dans la sortie standard. Vous pouvez alors lancer ce programme et envoyer sa sortie à travers un tube dans `git fast-import`.

Pour démontrer rapidement, vous écrirez un importateur simple. Supposez que vous travaillez dans `current`, vous sauvegardez votre projet en copiant occasionnellement le dossier dans un dossier de sauvegarde estampillé de la date `back_YYYY_MM_DD`, et vous voulez importer cela dans Git. Votre structure de dossier ressemble à ceci :

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Pour importer un dossier Git, vous devez passer en revue comment Git stocke ses données. Comme vous vous le rappelez, Git est fondamentalement une liste liée d'objets *commit* qui pointent sur un instantané de contenu. Tout ce que vous avez à faire est de dire à `fast-import` ce que sont les instantanés de contenu, quelles données de *commit* pointent sur eux, et l'ordre dans lequel ils vont. Votre stratégie sera d'explorer les instantanés un à un et créer les *commits* avec les contenus dans chaque dossier, en liant chaque *commit* avec le précédent.

Comme nous l'avons fait dans **"Exemple de politique gérée par Git"**, nous écrirons ceci en Ruby, parce que c'est ce avec quoi nous travaillons généralement et ça a tendance à être facile à lire. Vous pouvez écrire cet exemple assez facilement avec n'importe quel langage de programmation auquel vous êtes familier – il faut seulement afficher l'information appropriée dans `stdout`. Et, si vous travaillez sous Windows, cela signifie que vous devrez prendre un soin particulier à ne pas introduire de retour chariot (carriage return, CR) à la fin de vos lignes – `git fast-import` est très exigeant ; il accepte seulement la fin de ligne (Line Feed, LF) et pas le retour chariot fin de ligne (CRLF) que Windows utilise.

Pour commencer, vous vous placerez dans le dossier cible et identifierez chaque sous-dossier, chacun étant un instantané que vous voulez importer en tant que *commit*. Vous vous placerez dans chaque sous-dossier et afficherez les commandes nécessaires pour l'exporter. Votre boucle basique principale ressemble à ceci :

```ruby
last_mark = nil

# boucle sur les dossiers
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # rentre dans chaque dossier cible
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Vous lancez `print_export` à l'intérieur de chaque dossier, qui prend le manifeste et la marque de l'instantané précédent et retourne la marque et l'empreinte de celui-ci ; de cette façon, vous pouvez les lier proprement. "Marque" est le terme de `fast-import` pour un identifiant que vous donnez à un *commit* ; au fur et à mesure que vous créez des *commits*, vous donnez à chacun une marque que vous pouvez utiliser pour le lier aux autres *commits*. Donc, la première chose à faire dans votre méthode `print_export` est de générer une marque à partir du nom du dossier :

```
mark = convert_dir_to_mark(dir)
```

Vous ferez ceci en créant un tableau de dossiers et en utilisant la valeur d'index comme marque, car une marque doit être un nombre entier. Votre méthode ressemble à ceci :

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Maintenant que vous avez une représentation par un entier de votre *commit*, vous avez besoin d'une date pour les métadonnées du *commit*. Puisque la date est exprimée dans le nom du dossier, vous l'analyserez. La ligne suivante dans votre fichier `print_export` est

```
date = convert_dir_to_date(dir)
```

où `convert_dir_to_date` est définie comme

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

Cela retourne une valeur entière pour la date de chaque dossier. Le dernier bout de méta-information dont vous avez besoin pour chaque *commit* est la donnée de l'auteur, que vous codez en dur dans une variable globale :

```
$author = 'John Doe <john@example.com>'
```

Maintenant vous êtes prêt à commencer à publier l'information de *commit* pour votre importateur. L'information initiale déclare que vous êtes en train de définir un objet *commit* et sur quelle branche il est, suivi de la marque que vous avez générée, l'information d'auteur et le message de *commit*, et ensuite le précédent *commit*, s'il y en a un. Le code ressemble à ceci :

```
# affiche l'information d'import
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Vous codez en dur le fuseau horaire (-0700) parce que c'est facile de faire ainsi. Si vous importez depuis un autre système, vous devez spécifier le fuseau horaire comme décalage. Le message de *commit* doit être exprimé dans un format spécial :

```
data (taille)\n(contenu)
```

Le format est constitué du mot data, de la taille de la donnée à lire, d'une nouvelle ligne et finalement de la donnée. Comme vous avez besoin d'utiliser le même format pour spécifier le contenu du fichier plus tard, vous créez une méthode assistante, `export_data` :

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Tout ce qui reste à faire est de spécifier le contenu du fichier pour chaque instantané. C'est facile, car vous les avez dans un dossier – vous pouvez imprimer la commande `deleteall` suivie par le contenu de chaque fichier du dossier. Git enregistrera ensuite chaque instantané de manière appropriée :

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note : Comme beaucoup de systèmes conçoivent leurs révisions comme des changements d'un *commit* à l'autre, fast-import peut aussi prendre des commandes avec chaque *commit* pour spécifier quels fichiers ont été ajoutés, sup-

primés ou modifiés et ce qu'est le nouveau contenu. Vous pourriez calculer les différences entre instantanés et fournir seulement cette donnée, mais faire ainsi est plus complexe – vous pouvez aussi bien donner à Git toutes les données et le laisser faire. Si cela convient mieux pour vos données, référez-vous à la page de manuel `fast-import` pour les détails sur la manière de fournir les données de cette façon.

Le format pour lister le contenu d'un nouveau fichier ou pour spécifier un fichier modifié avec le nouveau contenu est le suivant :

```
M 644 inline path/to/file
data (taille)
(contenu du fichier)
```

Ici, 644 est le mode (si vous avez des fichiers exécutables, vous devez le détecter et spécifier 755 à la place), et `inline` dit que vous listerez le contenu immédiatement après cette ligne. Votre méthode `inline_data` ressemble à ceci :

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Vous réutilisez la méthode `export_data` que vous avez définie plus tôt, parce que c'est de la même façon que vous avez spécifié vos données du message de *commit*.

La dernière chose que vous avez besoin de faire est de retourner la marque courante pour qu'elle soit passée à la prochaine itération :

```
return mark
```

Si vous êtes sous Windows, vous devrez vous assurer d'ajouter une étape supplémentaire. Comme mentionné précédemment, Windows utilise CRLF comme caractères de fin de ligne alors que `git fast-import` ne s'attend qu'à LF. Pour contourner ce problème et satisfaire `git fast-import`, vous devez indiquer à Ruby d'utiliser LF au lieu de CRLF :

```
$stdout.binmode
```

Et voilà. Voici le script dans son intégralité :

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"
```

```
$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end


def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
    puts "mark :#{mark}"
    puts "committer #{$author} #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
    end
    mark
end


# explore les dossiers
```

```ruby
last_mark = nil
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # move into the target directory
        Dir.chdir(dir) do
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

Si vous lancez ce script, vous obtiendrez un contenu qui ressemble à peu près à ceci :

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

Pour lancer l'importateur, envoyez à travers un tube cette sortie à `git fast-import` pendant que vous êtes dans le dossier Git dans lequel vous voulez importer. Vous pouvez créer un nouveau dossier et ensuite exécuter `git init` à l'intérieur de celui-ci comme point de départ, et ensuite exécuter votre script :

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5
      trees  :            4 (         1 duplicates          0 deltas of          4
      commits:            4 (         1 duplicates          0 deltas of          0
      tags   :            0 (         0 duplicates          0 deltas of          0
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

Comme vous pouvez le voir, lorsque c’est terminé avec succès, il vous donne un lot de statistiques sur ce qu’il a fait. Dans ce cas-ci, vous avez importé un total de 13 objets pour 4 *commits* dans une branche. Maintenant, vous pouvez lancer `git log` pour voir votre nouvel historique :

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

Vous y voilà — un dépôt Git beau et propre. Il est important de noter que rien n'est extrait – vous n'avez d'abord aucun fichier dans votre répertoire de travail. Pour les obtenir, vous devez réinitialiser votre branche là où `master` est maintenant :

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Vous pouvez faire beaucoup plus avec l'outil `fast-import` – manipuler différents modes, les données binaires, les branches multiples et la fusion, les étiquettes, les indicateurs de progression, et plus encore. Nombre d'exemples de scénarios plus complexes sont disponibles dans le dossier `contrib/fast-import` du code source Git.

Résumé

Vous devriez être à l'aise en utilisant Git comme client pour d'autres systèmes de contrôle de version, ou en important presque n'importe quel dépôt existant dans Git sans perdre de donnée. Dans le prochain chapitre, nous couvrirons les tripes brutes de Git afin que vous puissiez travailler chaque octet, si besoin est.

Les tripes de Git 10

Vous êtes peut-être arrivé à ce chapitre en en sautant certains autres ou après avoir parcouru tout le reste du livre. Dans tous les cas, c'est ici que le fonctionnement interne et la mise en œuvre de Git sont abordés. Pour nous, leur apprentissage a été fondamental pour comprendre à quel point Git est utile et puissant, mais d'autres soutiennent que cela peut être source de confusion et peut être trop complexe pour les débutants. Nous en avons donc fait le dernier chapitre de ce livre pour que vous puissiez le lire tôt ou tard lors de votre apprentissage. Nous vous laissons le choix.

Maintenant que vous êtes ici, commençons. Tout d'abord, si ce n'est pas encore clair, Git est fondamentalement un système de fichiers adressable par contenu avec l'interface utilisateur d'un VCS au-dessus. Vous en apprendrez plus sur ce que cela signifie dans quelques instants.

Aux premiers jours de Git (surtout avant la version 1.5), l'interface utilisateur était beaucoup plus complexe, car elle était centrée sur le système de fichiers plutôt que sur l'aspect VCS. Ces dernières années, l'interface utilisateur a été peaufinée jusqu'à devenir aussi cohérente et facile à utiliser que n'importe quel autre système. Pour beaucoup, l'image du Git des débuts avec son interface utilisateur complexe et difficile à apprendre est toujours présente.

La couche système de fichiers adressable par contenu est vraiment géniale et nous l'aborderons dans ce chapitre. Ensuite, vous apprendrez les mécanismes de transfert ainsi que les tâches que vous serez amené à accomplir pour maintenir un dépôt.

Plomberie et porcelaine

Ce livre couvre l'utilisation de Git avec une trentaine de verbes comme `checkout`, `branch`, `remote`… Mais, puisque Git était initialement une boîte à outils (*toolkit*) pour VCS, plutôt qu'un VCS complet et convivial, il dispose de tout un ensemble d'actions pour les tâches bas niveau qui étaient conçues pour être liées dans le style UNIX ou appelées depuis des scripts. Ces commandes sont

dites commandes de « plomberie » (*plumbing*) et les autres, plus conviviales sont appelées « la porcelaine » (*porcelain*).

Les neuf premiers chapitres du livre concernent presque exclusivement les commandes de porcelaine. Par contre, dans ce chapitre, vous serez principalement confronté aux commandes de plomberie bas niveau, car elles vous donnent accès au fonctionnement interne de Git et aident à montrer comment et pourquoi Git fonctionne comme il le fait. Beaucoup de ces commandes ne sont pas faites pour être utilisées à la main sur une ligne de commande, mais sont plutôt utilisées comme briques de base pour écrire de nouveaux outils et scripts personnalisés.

Quand vous exécutez `git init` dans un nouveau répertoire ou un répertoire existant, Git crée un répertoire `.git` qui contient presque tout ce que Git stocke et manipule. Si vous voulez sauvegarder ou cloner votre dépôt, copier ce seul répertoire suffirait presque. Ce chapitre traite principalement de ce que contient ce répertoire. Voici à quoi il ressemble :

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Vous y verrez sans doute d'autres fichiers, mais ceci est un dépôt qui vient d'être créé avec `git init` et c'est ce que vous verrez par défaut. Le fichier `description` est utilisé uniquement par le programme GitWeb, il ne faut donc pas s'en soucier. Le fichier `config` contient les options de configuration spécifiques à votre projet et le répertoire `info` contient un fichier d'exclusions listant les motifs que vous souhaitez ignorer et que vous ne voulez pas mettre dans un fichier `.gitignore`. Le répertoire `hooks` contient les scripts de procédures automatiques côté client ou serveur, ils sont décrits en détail dans **"Crochets Git"**.

Il reste quatre éléments importants : les fichiers HEAD et (pas encore créé) `index`, ainsi que les répertoires `objects` et `refs`. Ce sont les composants principaux d'un dépôt Git. Le répertoire `objects` stocke le contenu de votre base de données, le répertoire `refs` stocke les pointeurs vers les objets *commit* de ces données (branches), le fichier HEAD pointe sur la branche qui est en cours dans votre répertoire de travail et le fichier `index` est l'endroit où Git stocke les

informations sur la zone d'attente. Vous allez maintenant plonger en détail dans chacune de ces sections et voir comment Git fonctionne.

Les objets de Git

Git est un système de fichier adressables par contenu. Super ! Mais qu'est-ce que ça veut dire ? Ça veut dire que le cœur de Git est une simple base de paires clé/valeur. Vous pouvez y insérer n'importe quelle sorte de données et il vous retournera une clé que vous pourrez utiliser à n'importe quel moment pour récupérer ces données. Pour illustrer cela, vous pouvez utiliser la commande de plomberie `hash-object`, qui prend des données, les stocke dans votre répertoire `.git`, puis retourne la clé sous laquelle les données sont stockées. Tout d'abord, créez un nouveau dépôt Git et vérifiez que rien ne se trouve dans le répertoire `objects` :

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git a initialisé le répertoire `objects` et y a créé les sous-répertoires `pack` et `info`, mais ils ne contiennent pas de fichier régulier. Maintenant, stockez du texte dans votre base de données Git :

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

L'option `-w` spécifie à `hash-object` de stocker l'objet, sinon la commande répondrait seulement quelle serait la clé. `--stdin` spécifie à la commande de lire le contenu depuis l'entrée standard, sinon `hash-object` s'attend à trouver un chemin vers un fichier. La sortie de la commande est une empreinte de 40 caractères. C'est l'empreinte SHA-1 – une somme de contrôle du contenu du fichier que vous stockez plus un en-tête, que vous apprendrez sous peu. Voyez maintenant comment Git a stocké vos données :

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Vous pouvez voir un fichier dans le répertoire `objects`. C'est comme cela que Git stocke initialement du contenu – un fichier par contenu, nommé d'après la somme de contrôle SHA-1 du contenu et de son en-tête. Le sous-répertoire est nommé d'après les 2 premiers caractères de l'empreinte et le fichier d'après les 38 caractères restants.

Vous pouvez récupérer le contenu avec la commande `cat-file`. Cette commande est un peu le couteau suisse pour l'inspection des objets Git. Lui passer l'option `-p` ordonne à la commande `cat-file` de déterminer le type de contenu et de vous l'afficher joliment :

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Vous pouvez maintenant ajouter du contenu à Git et le récupérer à nouveau. Vous pouvez faire de même avec le contenu de fichiers. Par exemple, vous pouvez mettre en œuvre une gestion de version simple d'un fichier. D'abord, créez un nouveau fichier et enregistrez son contenu dans la base de données :

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Puis, modifiez le contenu du fichier et enregistrez-le à nouveau :

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Votre base de données contient les 2 versions du fichier, ainsi que le premier contenu que vous avez stocké ici :

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Vous pouvez maintenant restaurer le fichier à sa première version :

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

ou à sa seconde version :

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Mais se rappeler de la clé SHA-1 de chaque version de votre fichier n'est pas pratique. En plus, vous ne stockez pas le nom du fichier dans votre système – seulement le contenu. Ce type d'objet est appelé un blob (*Binary Large OBject*, soit en français : Gros Objet Binaire). Git peut vous donner le type d'objet de n'importe quel objet Git, étant donné sa clé SHA-1, avec `cat-file -t` :

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Les objets arbres

Le prochain type que nous allons étudier est l'arbre (*tree*) qui résout le problème de stockage du nom du fichier et vous permet d'enregistrer un groupe de fichiers ensemble. Git stocke du contenu de la même manière, mais plus simplement, qu'un système de fichier UNIX. Tout le contenu est stocké comme des objets de type arbre ou blob : un arbre correspondant à un répertoire UNIX et un blob correspond à peu près aux inodes ou au contenu d'un fichier. Un unique arbre contient une ou plusieurs entrées de type arbre, chacune incluant un pointeur SHA-1 vers un blob ou un sous-arbre (*sub-tree*) avec ses droits d'accès (*mode*), son type et son nom de fichier associés. L'arbre le plus récent d'un projet pourrait ressembler, par exemple, à ceci :

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

La syntaxe `master^{tree}` signifie l'objet arbre qui est pointé par le dernier *commit* de la branche `master`. Remarquez que le sous-répertoire `lib` n'est pas un blob, mais un pointeur vers un autre arbre :

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptuellement, les données que Git stocke ressemblent ceci :

FIGURE 10-1

Une version simple du modèle de données Git.

Vous pouvez facilement créer votre propre arbre. Git crée habituellement un arbre à partir de l'état de la zone d'attente ou index et écrit une série d'objets arbre à partir de là. Donc, pour créer un objet arbre, vous devez d'abord mettre en place un index en mettant quelques fichiers en attente. Pour créer un index contenant une entrée, la première version de votre fichier `test.txt` par exemple, utilisons la commande de plomberie `update-index`. Vous pouvez utiliser cette commande pour ajouter artificiellement une version plus ancienne à une nouvelle zone d'attente. Vous devez utiliser les options `--add` car le fichier n'existe pas encore dans votre zone d'attente (vous n'avez même pas encore mis en place une zone d'attente) et `--cacheinfo` car le fichier que vous ajoutez n'est

pas dans votre répertoire, mais dans la base de données. Vous pouvez ensuite préciser le mode, SHA-1 et le nom de fichier :

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Dans ce cas, vous précisez le mode `100644`, qui signifie que c'est un fichier normal. Les alternatives sont `100755`, qui signifie que c'est un exécutable, et `120000`, qui précise que c'est un lien symbolique. Le concept de « mode » a été repris des mode UNIX, mais est beaucoup moins flexible : ces trois modes sont les seuls valides pour Git, pour les fichiers (blobs) dans Git (bien que d'autres modes soient utilisés pour les répertoires et sous-modules).

Vous pouvez maintenant utiliser la commande `write-tree` pour écrire la zone d'attente dans un objet arbre. L'option `-w` est inutile (appeler `write-tree` crée automatiquement un objet arbre à partir de l'état de l'index si cet arbre n'existe pas) :

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Vous pouvez également vérifier que c'est un objet arbre :

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Vous allez créer maintenant un nouvel arbre avec la seconde version de `test.txt` et aussi un nouveau fichier :

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Votre zone d'attente contient maintenant la nouvelle version de `test.txt` ainsi que le nouveau fichier `new.txt`. Enregistrez cet arbre (c'est-à-dire enregistrez l'état de la zone d'attente ou index dans un objet arbre) et voyez à quoi il ressemble :

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Remarquez que cet arbre contient des entrées pour les deux fichiers et que l'empreinte SHA-1 de `test.txt` est l'empreinte de la « version 2 » de tout à l'heure (`1f7a7a`). Pour le plaisir, ajoutez le premier arbre à celui-ci, en tant que sous-répertoire. Vous pouvez récupérer un arbre de votre zone d'attente en exécutant `read-tree`. Dans ce cas, vous pouvez récupérer un arbre existant dans votre zone d'attente comme étant un sous-arbre en utilisant l'option `--prefix` de `read-tree` :

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Si vous créiez un répertoire de travail à partir du nouvel arbre que vous venez d'enregistrer, vous auriez deux fichiers à la racine du répertoire de travail, ainsi qu'un sous-répertoire appelé bak qui contiendrait la première version du fichier `test.txt`. Vous pouvez vous représenter les données que Git utilise pour ces structures comme ceci :

FIGURE 10-2

Structure du contenu de vos données Git actuelles.

Les objets *commit*

Vous avez trois arbres qui définissent différents instantanés du projet que vous suivez, mais le problème précédent persiste : vous devez vous souvenir des valeurs des trois empreintes SHA-1 pour accéder aux instantanés. Vous n'avez pas non plus d'information sur qui a enregistré les instantanés, quand et pourquoi. Ce sont les informations élémentaires qu'un objet *commit* stocke pour vous.

Pour créer un objet *commit*, il suffit d'exécuter `commit-tree` et de préciser l'empreinte SHA-1 d'un seul arbre et quels objets *commit*, s'il y en a, le précèdent directement. Commencez avec le premier arbre que vous avez créé :

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Vous pouvez voir votre nouvel objet *commit* avec `cat-file` :

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

```
first commit
```

Le format d'un objet *commit* est simple : il contient l'arbre racine de l'instantané du projet à ce moment, les informations sur l'auteur et le validateur (qui utilisent vos variables de configuration `user.name` et `user.email` et un horodatage); une ligne vide et le message de validation.

Ensuite, vous enregistrez les deux autres objets *commit*, chacun référençant le *commit* dont il est issu :

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Chacun des trois objets *commit* pointe sur un des trois arbres d'instantané que vous avez créés. Curieusement, vous disposez maintenant d'un historique Git complet que vous pouvez visualiser avec la commande `git log`, si vous la lancez sur le SHA-1 du dernier *commit* :

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

        third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

        second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit
```

```
test.txt | 1 +
1 file changed, 1 insertion(+)
```

Fantastique. Vous venez d'effectuer les opérations bas niveau pour construire un historique Git sans avoir utilisé une seule des commandes de haut niveau. C'est l'essence de ce que fait Git quand vous exécutez les commandes `git add` et `git commit`. Il stocke les blobs correspondant aux fichiers modifiés, met à jour l'index, écrit les arbres et ajoute les objets *commit* qui référencent les arbres racines venant juste avant eux. Ces trois objets principaux (le blob, l'arbre et le *commit*) sont initialement stockés dans des fichiers séparés du répertoire `.git/objects`. Voici tous les objets contenus dans le répertoire exemple, commentés d'après leur contenu :

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Si vous suivez les pointeurs internes de ces objets, vous obtenez un graphe comme celui-ci :

FIGURE 10-3

Tous les objets de votre répertoire Git.

Stockage des objets

Nous avons parlé plus tôt de l'en-tête présent avec le contenu. Prenons un moment pour étudier la façon dont Git stocke les objets. On verra comment stocker interactivement un objet blob (ici, la chaîne "what is up, doc?") avec le langage Ruby.

Vous pouvez démarrer Ruby en mode interactif avec la commande `irb` :

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git construit un en-tête qui commence avec le type de l'objet, ici un blob. Ensuite, il ajoute un espace suivi de taille du contenu et enfin un octet nul :

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git concatène l'en-tête avec le contenu original et calcule l'empreinte SHA-1 du nouveau contenu. En Ruby, vous pouvez calculer l'empreinte SHA-1 d'une

chaîne en incluant la bibliothèque « digest/SHA-1 » via la commande `require`, puis en appelant `Digest::SHA1.hexdigest()` sur la chaîne :

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresse le nouveau contenu avec zlib, ce que vous pouvez faire avec la bibliothèque zlib de Ruby. D'abord, vous devez inclure la bibliothèque et ensuite exécuter `Zlib::Deflate.deflate()` sur le contenu :

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

Finalement, vous enregistrerez le contenu compressé dans un objet sur le disque. Vous déterminerez le chemin de l'objet que vous voulez enregistrer (les deux premiers caractères de l'empreinte SHA-1 formeront le nom du sous-répertoire et les 38 derniers formeront le nom du fichier dans ce répertoire). En Ruby, on peut utiliser la fonction `FileUtils.mkdir_p()` pour créer un sous-répertoire s'il n'existe pas. Ensuite, ouvrez le fichier avec `File.open()` et enregistrez le contenu compressé en appelant la fonction `write()` sur la référence du fichier :

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

C'est tout ! Vous venez juste de créer un objet blob valide. Tout les objets Git sont stockés de la même façon, mais avec des types différents : l'en-tête commencera par « commit » ou « tree » au lieu de la chaîne « blob ». De plus, alors

que le contenu d'un blob peut être à peu près n'importe quoi, le contenu d'un *commit* ou d'un arbre est formaté de façon très précise.

Références Git

On peut exécuter quelque chose comme `git log 1a410e` pour visualiser tout l'historique, mais il faut se souvenir que `1a410e` est le dernier *commit* afin de parcourir l'historique et trouver tous ces objets. Vous avez besoin d'un fichier dans lequel vous pouvez stocker l'empreinte SHA-1 sous un nom simple afin d'utiliser ce pointeur plutôt que l'empreinte SHA-1 elle-même.

Git appelle ces pointeurs des « références », ou « refs ». On trouve les fichiers contenant des empreintes SHA-1 dans le répertoire `git/refs`. Dans le projet actuel, ce répertoire ne contient aucun fichier, mais possède une structure simple :

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Pour créer une nouvelle référence servant à se souvenir du dernier *commit*, vous pouvez simplement faire ceci :

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Vous pouvez maintenant utiliser la référence principale que vous venez de créer à la place de l'empreinte SHA-1 dans vos commandes Git :

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Il n'est pas conseillé d'éditer directement les fichiers des références. Git propose une manière sûre de mettre à jour une référence, c'est la commande `update-ref` :

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

C'est simplement ce qu'est une branche dans Git : un simple pointeur ou référence sur le dernier état d'une suite de travaux. Pour créer une branche à partir du deuxième *commit*, vous pouvez faire ceci :

```
$ git update-ref refs/heads/test cac0ca
```

Cette branche contiendra seulement le travail effectué jusqu'à ce *commit* :

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

La base de donnée Git ressemble maintenant à quelque chose comme ceci :

FIGURE 10-4

Le répertoire d'objets de Git avec les références de branches incluses.

Quand vous exécutez une commande comme `git branch (nomdebranche)`, Git exécute simplement la commande `update-ref` pour ajouter l'empreinte SHA-1 du dernier *commit* de la branche sur laquelle vous êtes quelle que soit la nouvelle référence que vous voulez créer.

La branche HEAD

On peut se poser la question : « Comment Git peut avoir connaissance de l'empreinte SHA-1 du dernier *commit* quand on exécute `git branch (branch-`

`name)` ? » La réponse est dans le fichier HEAD (qui veut dire tête en français, soit, ici, l'état courant).

Le fichier HEAD est une référence symbolique à la branche courante. Par référence symbolique, j'entends que contrairement à une référence normale, elle ne contient pas une empreinte SHA-1, mais plutôt un pointeur vers une autre référence. Si vous regardez ce fichier, vous devriez voir quelque chose comme ceci :

```
$ cat .git/HEAD
ref: refs/heads/master
```

Si vous exécutez `git checkout test`, Git met à jour ce fichier, qui ressemblera à ceci :

```
$ cat .git/HEAD
ref: refs/heads/test
```

Quand vous exécutez `git commit`, il crée l'objet *commit* en spécifiant le parent de cet objet *commit* quelle que soit l'empreinte SHA-1 pointée par la référence de HEAD.

On peut éditer manuellement ce fichier, mais encore une fois, il existe une commande plus sûre pour le faire : `symbolic-ref`. Vous pouvez lire le contenu de votre fichier HEAD avec cette commande :

```
$ git symbolic-ref HEAD
refs/heads/master
```

Vous pouvez aussi définir la valeur de HEAD :

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Vous ne pouvez pas définir une référence symbolique à une valeur non contenu dans refs :

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Étiquettes

Nous venons de parcourir les trois types d'objets utilisés par Git, mais il en existe un quatrième. L'objet étiquette (*tag* en anglais) ressemble beaucoup à un objet *commit*. Il contient un étiqueteur, une date, un message et un pointeur. La principale différence est que l'étiquette pointe en général vers un *commit* plutôt qu'un arbre. C'est comme une référence à une branche, mais elle ne bouge jamais : elle pointe toujours vers le même *commit*, lui donnant un nom plus sympathique.

Comme présenté au **Chapter 2**, il existe deux types d'étiquettes : annotée et légère. Vous pouvez créer une étiquette légère comme ceci :

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

C'est tout ce qu'est une étiquette légère : une référence qui n'est jamais modifiée. Une étiquette annotée est plus complexe. Quand on crée une étiquette annotée, Git crée un objet étiquette, puis enregistre une référence qui pointe vers lui plutôt que directement vers le *commit*. Vous pouvez voir ceci en créant une étiquette annotée (`-a` spécifie que c'est une étiquette annotée) :

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Voici l'empreinte SHA-1 de l'objet créé :

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Maintenant, exécutez la commande `cat-file` sur cette empreinte SHA-1 :

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Remarquez que le contenu de l'objet pointe vers l'empreinte SHA-1 du *commit* que vous avez étiqueté. Remarquez qu'il n'est pas nécessaire qu'il pointe vers un *commit*. On peut étiqueter n'importe quel objet. Par exemple, dans le

code source de Git, le mainteneur a ajouté sa clé publique GPG dans un blob et a étiqueté ce blob. Vous pouvez voir la clé publique en exécutant ceci sur un clone du dépôt Git :

```
$ git cat-file blob junio-gpg-pub
```

Le noyau Linux contient aussi une étiquette ne pointant pas vers un *commit* : la première étiquette créée pointe vers l'arbre initial lors de l'importation du code source.

Références distantes

Le troisième type de références que l'on étudiera sont les références distantes (*remotes*). Si l'on ajoute une référence distante et que l'on pousse des objets vers elle, Git stocke la valeur que vous avez poussée en dernier vers cette référence pour chaque branche dans le répertoire `refs/remotes`. Vous pouvez par exemple ajouter une référence distante nommée `origin` et y pousser votre branche `master` :

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Ensuite, vous pouvez voir l'état de la branche `master` dans la référence distante `origin` la dernière fois que vous avez communiqué avec le serveur en regardant le fichier `refs/remotes/origin/master` :

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Les références distantes diffèrent des branches (références `refs/heads`) principalement parce qu'on ne peut y accéder qu'en lecture seule. Vous pouvez éxécuter `git checkout` sur l'une d'entre elles, mais Git ne fera jamais pointer HEAD sur l'une d'elles, donc vous ne pourrez jamais en mettre une à jour en uti-

lisant une commande `commit`. Git les gère comme des marque-pages du dernier état connu de vers quoi ces branches pointent sur le serveur.

Fichiers groupés

Revenons à la base de donnée d'objets de notre dépôt Git de test. Pour l'instant, elle contient 11 objets : 4 blobs, 3 arbres, 3 *commits* et 1 étiquette :

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresse le contenu de ces fichiers avec zlib et on ne stocke pas grand chose ; au final, tous ces fichiers occupent seulement 925 octets. Ajoutons de plus gros contenu au dépôt pour montrer une fonctionnalité intéressante de Git. Pour la démonstration, nous allons ajouter le fichier `repo.rb` de la bibliothèque Grit. Il représente environ 22 ko de code source :

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Si vous observez l'arbre qui en résulte, vous verrez l'empreinte SHA-1 du blob contenant le fichier `repo.rb` :

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
```

```
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Vous pouvez vérifier la taille de l'objet sur disque à l'aide de `git cat-file` :

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Maintenant, modifiez légèrement le fichier et voyez ce qui arrive :

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Regardez l'arbre créé par ce *commit* et vous verrez quelque chose d'intéressant :

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Ce blob est un blob différent. Bien que l'on ait ajouté une seule ligne à la fin d'un fichier en faisant 400, Git enregistre ce nouveau contenu dans un objet totalement différent :

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Il y a donc deux objets de 22 ko quasiment identiques sur le disque. Ne serait-ce pas charmant si Git pouvait n'enregistrer qu'un objet en entier, le deuxième n'étant qu'un delta (une différence) avec le premier ?

Il se trouve que c'est possible. Le format initial dans lequel Git enregistre les objets sur le disque est appelé le format brut (*loose object*). De temps en temps, Git compacte plusieurs de ces objets en un seul fichier binaire appelé *packfile* (fichier groupé), afin d'économiser de l'espace et d'être plus efficace. Git effectue cette opération quand il y a trop d'objets au format brut, ou si l'on exécute manuellement la commande `git gc`, ou encore quand on pousse vers un ser-

veur distant. Pour voir cela en action, vous pouvez demander manuellement à Git de compacter les objets en exécutant la commande `git gc` :

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

Si l'on jette un œil dans le répertoire des objets, on constatera que la plupart des objets ne sont plus là et qu'un couple de fichiers est apparu :

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Les objets restant sont des blobs qui ne sont pointés par aucun *commit*. Dans notre cas, il s'agit des blobs « what is up, doc? » et « test content » créés plus tôt comme exemple. Puisqu'ils n'ont été ajoutés à aucun *commit*, ils sont considérés en suspend et ne sont pas compactés dans le nouveau fichier groupé.

Les autres fichiers sont le nouveau fichier groupé et un index. Le fichier groupé est un fichier unique rassemblant le contenu de tous les objets venant d'être supprimés du système de fichier. L'index est un fichier contenant les emplacements dans le fichier groupé, pour que l'on puisse accéder rapidement à un objet particulier. Ce qui est vraiment bien, c'est que les objets occupaient environ 22 ko d'espace disque avant `gc` et que le nouveau fichier groupé en occupe seulement 7. On a réduit l'occupation du disque de ⅔ en regroupant les objets.

Comment Git réalise-t-il cela ? Quand Git compacte des objets, il recherche les fichiers qui ont des noms et des tailles similaires, puis enregistre seulement les deltas entre une version du fichier et la suivante. On peut regarder à l'intérieur du fichier groupé et voir l'espace économisé par Git. La commande de plomberie `git verify-pack` vous permet de voir ce qui a été compacté :

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
```

```
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Ici, le blob `033b4`, qui, si on se souvient bien, était la première version du fichier `repo.rb`, référence le blob `b042a`, qui est la seconde version du fichier. La troisième colonne de l'affichage est la taille de l'objet dans le fichier compact et on peut voir que `b042a` occupe 22 ko dans le fichier, mais que `033b4` occupe seulement 9 octets. Ce qui est aussi intéressant est que la seconde version du fichier est celle qui est enregistrée telle quelle, tandis que la version originale est enregistrée sous forme d'un delta. La raison en est que vous aurez sans doute besoin d'accéder rapidement aux versions les plus récentes du fichier.

Une chose intéressante à propos de ceci est que l'on peut recompacter à tout moment. Git recompacte votre base de donnée occasionnellement, en essayant d'économiser de la place. Vous pouvez aussi recompacter à la main, en exécutant la commande `git gc` vous-même.

La *refspec*

Tout au long de ce livre, nous avons utilisé des associations simples entre les branches distantes et les références locales. Elles peuvent être plus complexes. Supposons que vous ajoutiez un dépôt distant comme ceci :

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Cela ajoute une section au fichier `.git/config`, contenant le nom du dépôt distant (`origin`), l'URL de ce dépôt et la *refspec* pour la récupération :

```
[remote "origin"]
        url = https://github.com/schacon/simplegit-progit
        fetch = +refs/heads/*:refs/remotes/origin/*
```

Le format de la *refspec* est un + facultatif, suivi de `<src>:<dst>`, où `<src>` est le motif des références du côté distant et `<dst>` est l'emplacement local où les références seront enregistrées. Le + précise à Git de mettre à jour la référence même si ce n'est pas une avance rapide.

Dans le cas par défaut, qui est celui d'un enregistrement automatique par la commande `git remote add`, Git récupère toutes les références de `refs/heads/` sur le serveur et les enregistre localement dans `refs/remotes/origin/`. Ainsi, s'il y a une branche `master` sur le serveur, vous pouvez accéder localement à l'historique de cette branche via :

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Ces syntaxes sont toutes équivalentes, car Git les développe en `refs/remotes/origin/master`.

Si vous préférez que Git récupère seulement la branche `master` et non chacune des branches du serveur distant, vous pouvez remplacer la ligne fetch par :

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

C'est la *refspec* par défaut de `git fetch` pour ce dépôt distant. Si l'on veut effectuer une action particulière une seule fois, la *refspec* peut aussi être précisée en ligne de commande. Pour tirer la branche `master` du dépôt distant vers la branche locale `origin/mymaster`, vous pouvez exécuter :

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Vous pouvez indiquer plusieurs *refspecs*. En ligne de commande, vous pouvez tirer plusieurs branches de cette façon :

```
$ git fetch origin master:refs/remotes/origin/mymaster \
         topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

Dans ce cas, la récupération (*pull*) de la branche `master` a été refusée car ce n'était pas une avance rapide. On peut surcharger ce comportement en précisant un + devant la *refspec*.

On peut aussi indiquer plusieurs *refspecs* pour la récupération, dans le fichier de configuration. Si vous voulez toujours récupérer les branches `master` et `experiment`, ajoutez ces deux lignes :

```
[remote "origin"]
        url = https://github.com/schacon/simplegit-progit
        fetch = +refs/heads/master:refs/remotes/origin/master
        fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Vous ne pouvez pas utiliser des jokers partiels, ce qui suit est donc invalide :

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

On peut toutefois utiliser des espaces de noms (*namespaces*) ou des répertoires pour accomplir cela. S'il existe une équipe qualité (QA) qui publie une série de branches et que l'on veut la branche `master`, les branches de l'équipe qualité et rien d'autre, on peut utiliser la configuration suivante :

```
[remote "origin"]
        url = https://github.com/schacon/simplegit-progit
        fetch = +refs/heads/master:refs/remotes/origin/master
        fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Si vous utilisez des processus complexes impliquant une équipe qualité, des développeurs et des intégrateurs qui publient des branches et qui collaborent sur des branches distantes, vous pouvez facilement utiliser des espaces de noms de cette façon.

Pousser des *refspecs*

Il est pratique de pouvoir récupérer des références issues d'espace de nom de cette façon, mais comment l'équipe qualité insère-t-elle ces branches dans l'espace de nom `qa/` en premier lieu ? On peut accomplir cela en utilisant les spécifications de références pour la publication.

Si l'équipe qualité veut publier sa branche `master` vers `qa/master` sur le serveur distant, elle peut exécuter :

```
$ git push origin master:refs/heads/qa/master
```

Si elle veut que Git le fasse automatiquement à chaque exécution de `git push origin`, elle peut ajouter une entrée `push` au fichier de configuration :

```
[remote "origin"]
        url = https://github.com/schacon/simplegit-progit
        fetch = +refs/heads/*:refs/remotes/origin/*
        push = refs/heads/master:refs/heads/qa/master
```

De même, cela fera que, par défaut, `git push origin` publiera la branche locale `master` sur la branche distante `qa/master`.

Supprimer des références

Vous pouvez aussi utiliser les *refspecs* pour supprimer des références sur le serveur distant en exécutant une commande comme :

```
$ git push origin :topic
```

La *refspec* ressemble à `<src>:<dst>`, mais en laissant vide la partie `<src>`, cela signifie une création de la branche à partir de rien et donc sa suppression.

Les protocoles de transfert

Git peut transférer des données entre deux dépôts de deux façons principales : le protocole « stupide » et le protocole « intelligent ».

Cette section fait un tour d'horizon du fonctionnement de ces deux protocoles.

Le protocole stupide

Si vous mettez en place un dépôt à accéder en lecture seule sur HTTP, c'est vraisemblablement le protocole stupide qui sera utilisé.

Ce protocole est dit « stupide », car il ne nécessite aucun code spécifique à Git côté serveur durant le transfert ; le processus de récupération est une série

de requêtes GET, où le client devine la structure du dépôt Git présent sur le serveur.

> Le protocole stupide est rarement utilisé ces derniers temps. Il est difficile de le rendre sécurisé ou privé, et donc la plupart des hébergeurs Git (sur le *cloud* ou sur serveur dédié) refusent de l'utiliser. On conseille généralement d'utiliser le protocole intelligent, qui est décrit plus loin.

Suivons le processus `http-fetch` pour la bibliothèque simplegit :

```
$ git clone http://server/simplegit-progit.git
```

La première chose que fait cette commande est de récupérer le fichier `info/refs`. Ce fichier est écrit par la commande `update-server-info` et c'est pour cela qu'il faut activer le crochet `post-receive`, sinon le transfert HTTP ne fonctionnera pas correctement :

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

On possède maintenant une liste des références distantes et empreintes SHA-1. Ensuite, on regarde vers quoi pointe HEAD, pour savoir sur quelle branche se placer quand on aura fini :

```
=> GET HEAD
ref: refs/heads/master
```

On aura besoin de se placer sur la branche `master`, quand le processus sera terminé. On est maintenant prêt à démarrer le processus de parcours. Puisque votre point de départ est l'objet *commit* `ca82a6` que vous avez vu dans le fichier `info/refs`, vous commencez par le récupérer :

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Vous obtenez un objet, cet objet est dans le format brut sur le serveur et vous l'avez récupéré à travers une requête HTTP GET statique. Vous pouvez le décompresser avec zlib, ignorer l'en-tête et regarder le contenu du *commit* :

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
```

```
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Puis, vous avez deux autres objets supplémentaires à récupérer : `cfda3b` qui est l'arbre du contenu sur lequel pointe le *commit* que nous venons de récupérer et `085bb3` qui est le *commit* parent :

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Cela vous donne le prochain objet *commit*. Récupérez l'objet arbre :

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oups, on dirait que l'objet arbre n'est pas au format brut sur le serveur, vous obtenez donc une réponse 404. On peut en déduire certaines raisons : l'objet peut être dans un dépôt suppléant ou il peut être dans un fichier groupé de ce dépôt. Git vérifie la liste des dépôts suppléants d'abord :

```
=> GET objects/info/http-alternates
(empty file)
```

Si la réponse contenait une liste d'URL suppléantes, Git aurait cherché les fichiers bruts et les fichiers groupés à ces emplacements, c'est un mécanisme sympathique pour les projets qui ont dérivé d'un autre pour partager les objets sur le disque. Cependant, puisqu'il n'y a pas de suppléants listés dans ce cas, votre objet doit se trouver dans un fichier groupé. Pour voir quels fichiers groupés sont disponibles sur le serveur, vous avez besoin de récupérer le fichier `objects/info/packs`, qui en contient la liste (générée également par `update-server-info`) :

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Il n'existe qu'un seul fichier groupé sur le serveur, votre objet se trouve évidemment dedans, mais vous allez tout de même vérifier l'index pour être sûr. C'est également utile lorsque vous avez plusieurs fichiers groupés sur le serveur, vous pouvez donc voir quel fichier groupé contient l'objet dont vous avez besoin :

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Maintenant que vous avez l'index du fichier groupé, vous pouvez vérifier si votre objet est bien dedans car l'index liste les empreintes SHA-1 des objets contenus dans ce fichier groupé et des emplacements de ces objets. Votre objet est là, allez donc récupérer le fichier groupé complet :

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Vous avez votre objet arbre, vous continuez donc le chemin des *commits*. Ils sont également tous contenus dans votre fichier groupé que vous venez de télécharger, vous n'avez donc pas d'autres requêtes à faire au serveur. Git récupère une copie de travail de votre branche `master` qui été référencée par HEAD que vous avez téléchargé au début.

Le protocole intelligent

Le protocole stupide est simple mais un peu inefficace, et il ne permet pas l'écriture de données du client au serveur. Le protocole intelligent est une méthode plus habituelle pour transférer des données, mais elle nécessite l'exécution sur le serveur d'un processus qui connaît Git : il peut lire les données locales et déterminer ce que le client a ou ce dont il a besoin pour générer un fichier groupé personnalisé pour lui. Il y a deux ensembles d'exécutables pour transférer les données : une paire pour téléverser des données et une paire pour en télécharger.

TÉLÉVERSER DES DONNÉES

Pour téléverser des données vers un exécutable distant, Git utilise les exécutables `send-pack` et `receive-pack`. L'exécutable `send-pack` tourne sur le client et se connecte à l'exécutable `receive-pack` du côté serveur.

SSH

Par exemple, disons que vous exécutez `git push origin master` dans votre projet et `origin` est défini comme une URL qui utilise le protocole SSH. Git appelle l'exécutable `send-pack`, qui initialise une connexion à travers SSH vers votre serveur. Il essaye d'exécuter une commande sur le serveur distant via un appel SSH qui ressemble à :

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
```

```
        delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

La commande `git-receive-pack` répond immédiatement avec une ligne pour chaque référence qu'elle connaît actuellement, dans ce cas, uniquement la branche `master` et son empreinte SHA-1. La première ligne contient également une liste des compétences du serveur (ici : `report-status`, `delete-refs` et quelques autres, dont l'identifiant du client).

Chaque ligne commence avec une valeur hexadécimale sur 4 caractères, spécifiant le reste de la longueur de la ligne. La première ligne, ici, commence avec `00a5`, soit 165 en hexadécimal, ce qui signifie qu'il y a 165 octets restants sur cette ligne. La ligne d'après est `0000`, signifiant que le serveur a fini de lister ses références.

Maintenant qu'il connait l'état du serveur, votre exécutable `send-pack` détermine quels *commits* il a de plus que le serveur. L'exécutable `send-pack` envoie alors à l'exécutable `receive-pack` les informations concernant chaque référence que cette commande `push` va mettre à jour. Par exemple, si vous mettez à jour la branche `master` et ajoutez la branche `experiment`, la réponse de `send-pack` ressemblera à quelque chose comme :

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 \
        refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d \
        refs/heads/experiment
0000
```

Git envoie une ligne pour chaque référence que l'on met à jour avec l'ancien SHA-1, le nouveau SHA-1 et la référence en train d'être mise à jour. La première ligne contient également les compétences du client. La valeur SHA-1 remplie de *0* signifie qu'il n'y avait rien à cet endroit avant, car vous êtes en train d'ajouter la référence `experiment`. Si vous étiez en train de supprimer une référence, vous verriez l'opposé : que des *0* du côté droit.

Puis, le client téléverse un fichier groupé de tous les objets que le serveur n'a pas encore.

Finalement, le serveur répond avec une indication de succès (ou d'échec) :

```
000eunpack ok
```

HTTP(S)

Le processus est quasiment le même avec HTTP, à une différence près lors de l'établissement de la liaison (*handshaking*). La connection est amorcée avec cette requête :

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master \
        report-status delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

Ceci est la fin du premier échange client-serveur. Le client fait alors une nouvelle requête, qui est cette fois un POST, avec les données fournies par `git-upload-pack`.

```
=> POST http://server/simplegit-progit.git/git-receive/pack
```

La requête POST contient la sortie de `send-pack` et le fichier groupé. Enfin, le serveur indique le succès ou l'échec dans sa réponse HTTP.

TÉLÉCHARGEMENT DES DONNÉES

Lorsque vous téléchargez des données, les exécutables `fetch-pack` et `upload-pack` entrent en jeu. Le client démarre un processus `fetch-pack` qui se connecte à un processus `upload-pack` du côté serveur pour négocier les données qui seront téléchargées.

SSH

Si vous téléchargez par SSH, `fetch-pack` fait quelque chose comme ceci :

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Une fois `fetch-pack` connecté, `upload-pack` lui répond quelque chose du style :

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
        side-band side-band-64k ofs-delta shallow no-progress include-tag \
        multi_ack_detailed symref=HEAD:refs/heads/master \
        agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Ceci est très proche de la réponse de `receive-pack` mais les compétences sont différentes. En plus, il envoie ce qui est pointé par HEAD (`symref=HEAD:refs/heads/master`), afin que le client sache ce qu'il doit récupérer dans le cas d'un clone.

À ce moment, `fetch-pack` regarde les objets qu'il a et répond avec la liste des objets dont il a besoin en envoyant « want » (vouloir) suivi du SHA-1 qu'il veut. Il envoie tous les objets qu'il a déjà avec « have » suivi du SHA-1. À la fin de la liste, il écrit « done » (fait) pour inciter l'exécutable `upload-pack` à commencer à envoyer le fichier groupé des données demandées :

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

L'établissement de la liaison pour une opération de téléchargement nécessite deux requêtes HTTP. La première est un `GET` vers le même point que dans le protocole stupide :

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
        side-band side-band-64k ofs-delta shallow no-progress include-tag \
        multi_ack_detailed no-done symref=HEAD:refs/heads/master \
        agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Ceci ressemble beaucoup à un appel à `git-upload-pack` par une connection SSH, mais le deuxième échange est fait dans une requête séparée :

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Une fois de plus, ce format est le même que plus haut. La réponse à cette requête indique le succès ou l'échec, et contient le fichier groupé.

Résumé sur les protocoles

Cette section contient un survol basique des protocoles de transfert. Les protocoles contiennent de nombreuses autres fonctionalités, comme les compéten-

ces `multi_ack` ou `side-band`, mais leur étude est hors du sujet de ce livre. Nous avons essayé de vous donner une idée générale des échanges entre client et serveur. Si vous souhaitez en connaître davantage, vous devrez probablement jeter un œil sur le code source de Git.

Maintenance et récupération de données

Parfois, vous aurez besoin de faire un peu de ménage : rendre un dépôt plus compact, nettoyer les dépôts importés, ou récupérer du travail perdu. Cette section couvrira certains de ces scénarios.

Maintenance

De temps en temps, Git exécute automatiquement une commande appelée « auto gc ». La plupart du temps, cette commande ne fait rien. Cependant, s'il y a trop d'objets bruts (des objets qui ne sont pas dans des fichiers groupés), ou trop de fichiers groupés, Git lance une commande `git gc` à part entière. « gc » est l'abréviation de « garbage collect » (ramasse-miettes) et la commande fait plusieurs choses : elle rassemble plusieurs objets bruts et les place dans des fichiers groupés, elle rassemble des fichiers groupés en un gros fichier groupé et elle supprime des objets qui ne sont plus accessibles depuis aucun *commit* et qui sont vieux de plusieurs mois.

Vous pouvez exécuter `auto gc` manuellement :

```
$ git gc --auto
```

Encore une fois, cela ne fait généralement rien. Vous devez avoir environ 7 000 objets bruts ou plus de 50 fichiers groupés pour que Git appelle une vraie commande `gc`. Vous pouvez modifier ces limites avec les propriétés de configuration `gc.auto` et `gc.autoPackLimit`, respectivement.

`gc` regroupera aussi vos références dans un seul fichier. Supposons que votre dépôt contienne les branches et étiquettes suivantes :

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Si vous exécutez `git gc`, vous n'aurez plus ces fichiers dans votre répertoire `refs`. Git les déplacera pour plus d'efficacité dans un fichier nommé `.git/packed-refs` qui ressemble à ceci :

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Si vous mettez à jour une référence, Git ne modifiera pas ce fichier, mais enregistrera plutôt un nouveau fichier dans `refs/heads`. Pour obtenir l'empreinte SHA-1 appropriée pour une référence donnée, Git cherche d'abord cette référence dans le répertoire `refs`, puis dans le fichier `packed-refs` si non trouvée. Si vous ne pouvez pas trouver une référence dans votre répertoire `refs`, elle est probablement dans votre fichier `packed-refs`.

Remarquez la dernière ligne du fichier, celle commençant par `^`. Cela signifie que l'étiquette directement au-dessus est une étiquette annotée et que cette ligne est le *commit* que l'étiquette annotée référence.

Récupération de données

À un moment quelconque de votre vie avec Git, vous pouvez accidentellement perdre un *commit*. Généralement, cela arrive parce que vous avez forcé la suppression d'une branche contenant du travail et il se trouve que vous vouliez cette branche finalement ; ou vous avez réinitialisé une branche avec suppression, en abandonnant des *commits* dont vous vouliez des informations. Supposons que cela arrive, comment pouvez-vous récupérer vos *commits* ?

Voici un exemple qui réinitialise la branche `master` avec suppression dans votre dépôt de test vers un ancien *commit* et qui récupère les *commits* perdus. Premièrement, vérifions dans quel état est votre dépôt en ce moment :

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Maintenant, déplaçons la branche `master` vers le *commit* du milieu :

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Vous avez effectivement perdu les deux *commits* du haut, vous n'avez pas de branche depuis laquelle ces *commits* seraient accessibles. Vous avez besoin de trouver le SHA du dernier *commit* et d'ajouter une branche s'y référant. Le problème est de trouver ce SHA, ce n'est pas comme si vous l'aviez mémorisé, hein ?

Souvent, la manière la plus rapide est d'utiliser l'outil `git reflog`. Pendant que vous travaillez, Git enregistre l'emplacement de votre HEAD chaque fois que vous le changez. À chaque *commit* ou commutation de branche, le journal des références (*reflog*) est mis à jour. Le journal des références est aussi mis à jour par la commande `git update-ref`, ce qui est une autre raison de l'utiliser plutôt que de simplement écrire votre valeur SHA dans vos fichiers de références, comme mentionné dans la section **"Références Git"** plus haut dans ce chapitre. Vous pouvez voir où vous étiez à n'importe quel moment en exécutant `git reflog` :

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Ici, nous pouvons voir deux *commits* que nous avons récupérés, cependant, il n'y a pas plus d'information ici. Pour voir, les mêmes informations d'une manière plus utile, nous pouvons exécuter `git log -g`, qui nous donnera une sortie normalisée pour votre journal de références :

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

                third commit
```

```
commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       modified repo.rb a bit
```

On dirait que le *commit* du bas est celui que vous avez perdu, vous pouvez donc le récupérer en créant une nouvelle branche sur ce *commit*. Par exemple, vous créez une branche nommée `recover-branch` sur ce *commit* (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool. Maintenant vous avez une nouvelle branche appelée `recover-branch` à l'emplacement où votre branche `master` se trouvait, rendant les deux premiers *commits* à nouveau accessibles. Pour poursuivre, nous supposerons que vos pertes ne sont pas dans le journal des références pour une raison quelconque. On peut simuler cela en supprimant `recover-branch` et le journal des références. Maintenant, les deux premiers *commits* ne sont plus accessibles :

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Comme les données du journal de référence sont sauvegardées dans le répertoire `.git/logs/`, vous n'avez effectivement plus de journal de références. Comment pouvez-vous récupérer ces *commits* maintenant ? Une manière de faire est d'utiliser l'outil `git fsck`, qui vérifie l'intégrité de votre base de données. Si vous l'exécutez avec l'option `--full`, il vous montre tous les objets qui ne sont pas référencés par d'autres objets :

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
```

```
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

Dans ce cas, vous pouvez voir votre *commit* manquant après « dangling commit ». Vous pouvez le restaurer de la même manière que précédemment, en créant une branche qui référence cette empreinte SHA-1.

Suppression d'objets

Il y a beaucoup de choses dans Git qui sont géniales, mais une fonctionnalité qui peut poser problème est le fait que `git clone` télécharge l'historique entier du projet, incluant chaque version de chaque fichier. C'est très bien lorsque le tout est du code source, parce que Git est hautement optimisé pour compresser les données efficacement. Cependant, si quelqu'un à un moment donné de l'historique de votre projet a ajouté un énorme fichier, chaque clone sera forcé de télécharger cet énorme fichier, même s'il a été supprimé du projet dans le *commit* suivant. Puisqu'il est accessible depuis l'historique, il sera toujours là.

Cela peut être un énorme problème, lorsque vous convertissez un dépôt Subversion ou Perforce en un dépôt Git. Comme comme vous ne téléchargez pas l'historique entier dans ces systèmes, ce genre d'ajout n'a que peu de conséquences. Si vous avez importé depuis un autre système ou que votre dépôt est beaucoup plus gros que ce qu'il devrait être, voici comment vous pouvez trouver et supprimer des gros objets.

Soyez prévenu : cette technique détruit votre historique de *commit*. Elle réécrit chaque objet *commit* depuis le premier objet arbre que vous modifiez pour supprimer une référence d'un gros fichier. Si vous faites cela immédiatement après un import, avant que quiconque n'ait eu le temps de commencer à travailler sur ce *commit*, tout va bien. Sinon, vous devez alerter tous les contributeurs qu'ils doivent rebaser leur travail sur vos nouveaux *commits*.

Pour la démonstration, nous allons ajouter un gros fichier dans votre dépôt de test, le supprimer dans le *commit* suivant, le trouver et le supprimer de manière permanente du dépôt. Premièrement, ajoutons un gros objet à votre historique :

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oups, vous ne vouliez pas ajouter une énorme archive à votre projet. Il vaut mieux s'en débarrasser :

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Maintenant, faites un `gc` sur votre base de données, pour voir combien d'espace disque vous utilisez :

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Vous pouvez exécuter la commande `count-objects` pour voir rapidement combien d'espace disque vous utilisez :

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

L'entrée `size-pack` est la taille de vos fichiers groupés en kilo-octet, vous utilisez donc presque 5 Mo. Avant votre dernier *commit*, vous utilisiez environ 2 ko ; clairement, supprimer le fichier avec le *commit* précédent ne l'a pas enlevé de votre historique. À chaque fois que quelqu'un clonera votre dépôt, il aura à cloner les 5 Mo pour récupérer votre tout petit projet, parce que vous avez accidentellement rajouté un gros fichier. Débarrassons-nous en.

Premièrement, vous devez le trouver. Dans ce cas, vous savez déjà de quel fichier il s'agit. Mais supposons que vous ne le sachiez pas, comment identifieriez-vous quel(s) fichier(s) prennent trop de place ? Si vous exécutez

`git gc`, tous les objets sont dans des fichiers groupés ; vous pouvez identifier les gros objets en utilisant une autre commande de plomberie appelée `git verify-pack` et en triant sur le troisième champ de la sortie qui est la taille des fichiers. Vous pouvez également le faire suivre à la commande `tail` car vous ne vous intéressez qu'aux fichiers les plus gros :

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

Le gros objet est à la fin : 5 Mio. Pour trouver quel fichier c'est, vous allez utiliser la commande `rev-list`, que vous avez utilisée brièvement dans **"Application d'une politique de format du message de validation"**. Si vous mettez l'option `--objects` à `rev-list`, elle listera tous les SHA des *commits* et des blobs avec le chemin du fichier associé. Vous pouvez utilisez cette commande pour trouver le nom de votre blob :

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Maintenant, vous voulez supprimer ce fichier de toutes les arborescences passées. Vous pouvez facilement voir quels *commits* ont modifié ce fichier :

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

Vous devez réécrire tous les *commits* en descendant depuis `7b30847` pour supprimer totalement ce fichier de votre historique Git. Pour cela, utilisez `filter-branch`, que vous avez utilisée dans le chapitre **"Réécrire l'historique"** :

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

L'option `--index-filter` est similaire à l'option `--tree-filter` utilisée dans le chapitre **"Réécrire l'historique"**, sauf qu'au lieu de modifier les fichiers sur le disque, vous modifiez votre index.

Plutôt que de supprimer un fichier spécifique avec une commande comme `rm file`, vous devez le supprimer avec `git rm --cached` ; vous devez le supprimer de l'index, pas du disque. La raison de faire cela de cette manière est la rapidité, car Git n'ayant pas besoin de récupérer chaque révision sur disque avant votre filtre, la procédure peut être beaucoup, beaucoup plus rapide. Vous pouvez faire la même chose avec `--tree-filter` si vous voulez. L'option `--ignore-unmatch` de `git rm` lui dit que ce n'est pas une erreur si le motif que vous voulez supprimez n'existe pas. Finalement, vous demandez à `filter-branch` de réécrire votre historique seulement depuis le parent du *commit* `7b30847`, car vous savez que c'est de là que le problème a commencé. Sinon, il aurait démarré du début et serait plus long inutilement.

Votre historique ne contient plus de référence à ce fichier. Cependant, votre journal de révision et un nouvel ensemble de références que Git a ajouté lors de votre `filter-branch` dans `.git/refs/original` en contiennent encore, vous devez donc les supprimer puis regrouper votre base de données. Vous devez vous débarrasser de tout ce qui fait référence à ces vieux *commits* avant de regrouper :

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Voyons combien d'espace vous avez récupéré :

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

La taille du dépôt regroupé est retombée à 8 ko, ce qui est beaucoup mieux que 5 Mo. Vous pouvez voir dans la valeur « size » que votre gros objet est toujours dans vos objets bruts, il n'est donc pas parti ; mais il ne sera plus transféré lors d'une poussée vers un serveur ou un clone, ce qui est l'important dans l'histoire. Si vous le voulez réellement, vous pouvez supprimer complètement l'objet en exécutant `git prune` avec l'option `--expire` :

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Les variables d'environnement

Git s'exécute toujours dans un shell `bash`, et utilise un certain nombre de variables d'environnement pour savoir comment se comporter. Il est parfois pratique de savoir lesquelles, et la façon de les utiliser pour que Git se comporte comme vous le souhaitez. Ceci n'est pas une liste exhaustive de toutes les variables d'environnement que Git utilise, mais nous allons voir les plus utiles.

Comportement général

Certains aspects du comportement général de Git en tant que programme dépend de variables d'environnement.

`GIT_EXEC_PATH` détermine l'endroit où Git va chercher ses sous-programmes (comme `git-commit`, `git-diff`, et d'autres). Vous pouvez vérifier le réglage actuel en lançant `git --exec-path`.

`HOME` n'est pas en général considérée comme modifiable (trop d'autres choses en dépendent), mais c'est l'endroit où Git va chercher le fichier de configuration général (*global*). Si vous voulez une installation de Git vraiment portable, complète du point de vue de la configuration générale, vous pouvez surcharger `HOME` dans le profil (*profile*).

`PREFIX` est l'équivalent pour la configuration au niveau du système. Git va chercher le fichier `$PREFIX/etc/gitconfig`.

GIT_CONFIG_NOSYSTEM, si elle est définie, invalide l'utilisation du fichier de configuration au niveau du système. Cette variable est utile si la configuration système interfère avec vos commandes et que vous n'avez pas les privilèges pour la changer ou la supprimer.

GIT_PAGER contrôle le programme que vous utilisez pour afficher les résultats sur plusieurs pages à la ligne de commande. Si elle n'est pas définie, Git utilisera PAGER à la place.

GIT_EDITOR est l'éditeur lancé par Git quand l'utilisateur doit taper du texte (un message de *commit* par exemple). Si elle n'est pas définie, Git utilisera EDITOR.

Les emplacements du dépôt

Git utilise plusieurs variables d'environnement pour déterminer comment interagir avec le dépôt courant.

GIT_DIR est l'emplacement du répertoire `.git`. S'il n'est pas spécifié, Git remonte l'arbre des répertoires jusqu'à ce qu'il arrive à ~ ou bien /, en cherchant un répertoire `.git` à chaque étape.

GIT_CEILING_DIRECTORIES contrôle le comportement de Git pendant la recherche d'un répertoire `.git`. Si vous êtes sur des répertoires qui se chargent lentement (par exemple sur une bande magnétique ou à travers une connexion réseau lente), vous pouvez souhaiter que Git s'arrête plus tôt qu'il ne le ferait habituellemnt, surtout si Git est appelé à la construction de votre appel shell (*prompt*).

GIT_WORK_TREE est l'emplacement de la racine du répertoire de travail pour un dépôt non nu. Si cette variable n'est pas spécifiée, c'est le répertoire parent de $GIT_DIR qui est utilisé.

GIT_INDEX_FILE est le chemin du fichier d'index (uniquement pour les dépôts non nus).

GIT_OBJECT_DIRECTORY peut être utilisé pour spécifier l'emplacement du répertoire qui se trouve habituellement à `.git/objects`.

GIT_ALTERNATE_OBJECT_DIRECTORIES est une liste séparée par des « : » (formattée comme ceci : `/rep/un:/rep/deux:…`) qui dit à Git où trouver les objets s'ils ne sont pas dans GIT_OBJECT_DIRECTORY. S'il vous arrive d'avoir beaucoup de projets avec des gros fichiers ayant exactement le même contenu, cette variable peut vous éviter d'en garder trop de copies.

Pathspecs

Une "*pathspec*" fait référence à la façon dont on spécifie les chemins dans Git, y compris l'utilisation des jokers. Ils sont utilisés dans le fichier `.gitignore`, mais également à la ligne de commande (`git add *.c`).

`GIT_GLOB_PATHSPECS` et `GIT_NOGLOB_PATHSPECS` contrôlent le comportement par défaut des jokers dans les *pathspecs*. Si `GIT_GLOB_PATHSPECS` vaut 1, les caractères jokers agissent comme des jokers (ce qui est le comportement par défaut) ; si `GIT_NOGLOB_PATHSPECS` vaut 1, les caractères jokers ne correspondent qu'à eux-même, ce qui veut dire que quelque chose comme `*.c` ne correspondrait qu'à un fichier *nommé* « *.c », et non pas tout fichier dont le nom se termine par `.c`. Vous pouvez surcharger ce comportement pour certains cas en faisant commencer la *pathspec* par `:(glob)` pour utiliser le joker, ou bien `:(literal)` pour une correspondance stricte, comme dans `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` empêche ces deux comportements ; aucun joker ne fonctionnera, et les préfixes de surcharge seront également inopérants.

`GIT_ICASE_PATHSPECS` rend toutes les *pathspecs* insensibles à la casse.

Création de *commits*

La création finale d'un objet Git *commit* est habituellement faite par `git-commit-tree`, qui utilise les variables d'environnement suivantes comme première source d'information, se repliant sur les valeurs de configuration seulement si celles-ci ne sont pas présentes :

`GIT_AUTHOR_NAME` est le nom lisible par un humain dans le champ « Auteur » (*author*).

`GIT_AUTHOR_EMAIL` est l'adresse de courriel pour le champ « Auteur ».

`GIT_AUTHOR_DATE` est l'horodatage utilisé pourle champ « Auteur ».

`GIT_COMMITTER_NAME` définit le nom humain pour le champ « Validateur » (*commiter*).

`GIT_COMMITTER_EMAIL` est l'adresse de courriel pour le champ « Validateur ».

`GIT_COMMITTER_DATE` est utilisé pour l'horodatage dans le champ « Validateur ».

`EMAIL` est l'adresse de courriel de repli pour le cas où la valeur de configuration `user.email` n'est pas définie. Si *celle-ci* n'est pas définie, Git se replie sur les noms d'utilisateur système et d'hôte.

Travail sur le réseau

Git utilise la bibliothèque `curl` pour effectuer des opérations sur HTTP, ainsi **GIT_CURL_VERBOSE** demande à Git d'émettre tous les messages générés par cette bibliothèque. C'est similaire à `curl -v` en ligne de commande.

GIT_SSL_NO_VERIFY demande à Git de ne pas vérifier les certificats SSL. Cela peut être parfois nécessaire si vous utilisez des certificats auto-signés pour servir des dépôts Git sur HTTPS, ou si vous êtes au milieu de l'installation d'un serveur Git mais n'avez pas encore installé un certificat complet.

Si le taux de données d'une opération HTTP est plus basse que **GIT_HTTP_LOW_SPEED_LIMIT** octets par seconde pendant plus longtemps que **GIT_HTTP_LOW_SPEED_TIME** secondes, Git annulera cette opération. Ces valeurs surchargent les valeurs de configuration `http.lowSpeedLimit` et `http.lowSpeedTime`.

GIT_HTTP_USER_AGENT définit la chaîne d'agent utilisateur utilisée par Git quand il communique sur HTTP. La valeur par défaut est quelque chose comme `git/2.0.0`.

Visualisation des différences et Fusion

GIT_DIFF_OPTS est un terme un peu inapproprié. Les seules valeurs valides sont `-u<n>` ou `--unified=<n>`, qui contrôlent le nombre de lignes de contexte affichées dans une commande `git diff`.

GIT_EXTERNAL_DIFF est utilisée comme une surcharge de la valeur de configuration `diff.external`. Si elle est définie, Git invoquera ce programme quand `git diff` sera invoquée.

GIT_DIFF_PATH_COUNTER et **GIT_DIFF_PATH_TOTAL** sont utiles à l'intérieur du programme spécifié par `GIT_EXTERNAL_DIFF` ou `diff.external`. Le premier represente le fichier de la série dont on est en train de visualiser les différences (en commençant par 1), et le dernier est le nombre total de fichiers dans le lot.

GIT_MERGE_VERBOSITY contrôle la sortie pour la stratégie de fusion récursive. Les valeurs admises sont les suivantes :

- 0 ne sort rien, sauf éventuellement un seul message d'erreur.
- 1 ne montre que les conflits.
- 2 montre aussi les modifications de fichier.
- 3 montre quand les fichiers sont sautés parce qu'ils n'ont pas changé.
- 4 montre tous les chemins qui sont en train d'être traités.
- 5 et au-delà montrent des informations détaillées de débogage.

La valeur par défaut est 2.

Débogage

Vous voulez *vraiment* savoir de quoi Git est capable ? Git comprend un ensemble de traces assez complet, et tout ce que vous avez à faire est de les activer. Les valeurs possibles de ces variables sont les suivantes :

- « true », « 1 » ou « 2 » – la catégorie de trace est écrite sur la sortie d'erreur standard (stderr).
- Un chemin absolu commençant par / – la sortie de trace sera écrite dans ce fichier.

GIT_TRACE contrôle les traces générales, qui ne rentrent dans aucune catégorie spécifique. Cela inclut le développement des alias et la délégation aux autres sous-programmes.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga => 'log' '--gr
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--pr
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

GIT_TRACE_PACK_ACCESS contrôle le traçage d'accès aux fichiers groupés. Le premier champ est le fichier groupé auquel on est en train d'accéder, le second est le décalage dans ce fichier :

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 34
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack 56
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack 14
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

GIT_TRACE_PACKET permet le traçage au niveau paquet pour les opérations sur le réseau.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-upload-pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git< 97b8860c071898d9e162678ea1035a
20:15:14.867088 pkt-line.c:46           packet:          git< 0f20ae29889d61f2e93ae00fd34f1c
20:15:14.867094 pkt-line.c:46           packet:          git< 36dc827bc9d17f80ed4f326de21247
# […]
```

GIT_TRACE_PERFORMANCE contrôle la journalisation d'information de performance. La sortie montre combien de temps prend chaque invocation particulière de Git.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git command: 'git' 'pack
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git' 'refl
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414             performance: 3.715349000 s: git command: 'git' 'pack
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git command: 'git' 'prun
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git command: 'git' 'upda
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git command: 'git' 'repa
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git command: 'git' 'prun
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git command: 'git' 'rere
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git command: 'git' 'gc'
```

GIT_TRACE_SETUP montre des informations sur ce que Git découvre sur le dépôt et l'environnement avec lequel il interagit.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315             setup: git_dir: .git
20:19:47.087184 trace.c:316             setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317             setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318             setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Divers

GIT_SSH, si spécifié, est un programme qui est invoqué à la place de `ssh` quand Git essaie de se connecter à un hôte SSH. Il est invoqué comme `$GIT_SSH [username@]host [-p <port>] <command>`. Notez que ce n'est pas le moyen le plus facile de personnaliser la façon dont ssh est invoqué ; il ne prendra pas en compte des paramètres supplémentaires en ligne de commande, donc vous devriez écrire un script l'enveloppant et faire pointer `GIT_SSH` dessus. Il est sans doute plus facile d'utiliser le fichier `~/.ssh/config` pour cela.

GIT_ASKPASS est une surcharge pour la valeur de configuration `core.askpass`. C'est le programme invoqué lorsque Git à besoin de demander ses identifiants à l'utilisateur, qui peut s'attendre à un texte comme argument en ligne de commande, et qui devrait retourner la réponse sur la sortie standard (`stdout`). (Consultez **"Stockage des identifiants"** pour plus d'information sur ce sous-système.)

GIT_NAMESPACE contrôle l'accès des références cloisonnées dans des espaces de nom, et est équivalent à l'option `--namespace`. C'est surtout utile côté serveur, où vous pourriez vouloir stocker plusieurs bifurcations (*forks*) d'un seul dépôt dans un seul dépôt, en gardant seulement les références séparées.

GIT_FLUSH peut être utilisée pour forcer Git à utiliser des entrées/sorties non mises en mémoire tampon (*buffer*) quand il écrit progressivement dans la sortie standard. Une valeur de 1 fait que Git évacue (*flush*) plus souvent, une valeur de 0 fait que la sortie est mise en mémoire tampon. La valeur par défaut (si la variable n'est pas définie) est à choisir selon un plan approprié de mise en mémoire tampon en fonction de l'activité et du mode de sortie.

GIT_REFLOG_ACTION vous permet de spécifier le texte descriptif écrit dans le `reflog`. Voici un exemple :

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

Résumé

Vous devriez avoir une assez bonne compréhension de ce que Git fait en arrière-plan et, jusqu'à un certain niveau, comment il est implémenté. Ce chapitre a parcouru plusieurs commandes de plomberie, qui sont à un niveau plus bas et plus simple que les commandes de porcelaine que vous avez vues dans le reste du livre. Comprendre comment Git travaille à bas niveau devrait vous aider à

comprendre pourquoi il fait ce qu'il fait et à créer vos propres outils et scripts pour vous permettre de travailler comme vous l'entendez.

Git, en tant que système de fichiers adressable par contenu, est un outil puissant que vous pouvez utiliser pour des fonctionnalités au-delà d'un système de contrôle de version. Nous espérons que vous pourrez utiliser votre connaissance nouvellement acquise des tripes de Git pour implémenter votre propre super application avec cette technologie et que vous vous sentirez plus à l'aise pour utiliser Git de manière plus poussée.

Git dans d'autres environnements

Si vous avez lu le livre dans son intégralité, vous avez appris beaucoup sur son utilisation à partir de la ligne de commande. Vous pouvez travailler avec des fichiers locaux, connecter vos dépôts avec d'autres à travers un réseau et travailler efficacement avec d'autres personnes. Mais l'histoire ne s'arrête pas ici ; Git est habituellement utilisé comme sous-partie d'un écosystème plus grand et le terminal n'est pas toujours le meilleur moyen d'interagir dans ce cadre. Nous allons maintenant jeter un œil à certains des autres types d'environnements où Git s'avère utile ainsi que la manière dont d'autres applications (dont la vôtre) intègrent Git.

Interfaces graphiques

L'environnement natif de Git est le terminal. Les nouvelles fonctionnalités y apparaissent en premier et c'est seulement à la ligne de commande que vous disposez de tout le pouvoir de Git. Mais le texte pur n'est pas toujours le meilleur choix pour toutes les tâches ; quelques fois, une représentation visuelle est préférable et certains utilisateurs sont beaucoup plus à l'aise avec une interface pointer-cliquer.

Il est important de noter que différentes interfaces sont adaptées à différents modes de travail. Certains clients n'exposent qu'une partie soigneusement choisie des fonctionnalités de Git, pour supporter une façon spécifique de travailler que l'auteur considère efficace. Vu sous cet angle, aucun outil ne peut être qualifié de « meilleur » qu'un autre, il est simplement plus adapté à l'utilisation désirée. Il faut remarquer aussi qu'il n'y rien que ces outils graphiques ne fassent que l'interface en ligne de commande ne puisse faire ; la ligne de commande reste l'interface qui vous donne le plus de puissance et de contrôle sur vos dépôts.

`gitk` et `git-gui`

Quand vous installez Git, vous obtenez aussi ses outils visuels, `gitk` et `git-gui`.

`gitk` est l'outil de visualisation graphique d'historique. Voyez-le comme une interface GUI puissante par-dessus `git log` et `git grep`. C'est l'outil à utiliser lorsque vous essayez de trouver un événement passé ou de visualiser l'historique de votre projet.

Gitk est plus facile à invoquer depuis la ligne de commande. Positionnez-vous simplement dans le dépôt Git et tapez :

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

Gitk accepte de nombreuses options de ligne de commande, dont la plupart sont passées directement à la commande `git log` sous-jacente. L'une des plus intéressantes est probablement d'ajouter l'option `--all` qui indique à gitk de montrer tous les *commits* joignables depuis *n'importe quelle* référence, et pas seulement HEAD. L'interface de Gitk ressemble à ceci :

Figure 1-1.

Le visualisateur d'historique `gitk`.

Dans la partie supérieure, une zone ressemble à la sortie de `git log --graph`. Chaque point représente un *commit*, les lignes représentent les liens de parenté et les références apparaissent dans des rectangles colorés. Le point jaune représente HEAD et le point rouge représente les modifications qui ne sont pas validées. Dans la partie basse, on visualise le *commit* sélectionné : les commentaires et le *patch* sur la gauche et une vue en résumé sur la droite. Au milieu se trouve un ensemble de composants graphiques utilisés pour rechercher dans l'historique.

`git-gui`, par contre est un outil permettant de ciseler les *commits*. Lui-aussi est plus facile à invoquer en ligne de commande :

```
$ git gui
```

Et il ressemble à ceci :

Figure 1-2.

L'outil d'aide à la validation `git-gui`.

Sur la gauche, il y a l'index ; les modifications non indexées sont en haut, les modifications indexées en bas. Vous pouvez déplacer des fichiers entiers entre les deux états en cliquant sur leurs icônes ou vous pouvez sélectionner un fichier à visualiser en cliquant sur son nom.

La vue diff en haut à droite montre les modifications pour le fichier sélectionné. Vous pouvez indexer des sections individuelles (ou des lignes individuelles) en cliquant-droit dans cette zone.

La zone de message et d'action est en bas à droite. Tapez votre message dans la boîte à texte et cliquez « Commiter » pour réaliser une action similaire à `git commit`. Vous pouvez aussi choisir de corriger le *commit* précédent en sélectionnant le bouton radio « Corriger dernier commit », ce qui met à jour la zone « Modifs. indexées » avec le contenu du dernier *commit*. Ensuite, vous pouvez simplement indexer ou désindexer certaines modifications, modifier le message de validation et cliquer à nouveau sur le bouton « Commiter » pour remplacer l'ancien *commit* par le nouveau.

`gitk` et `git-gui` sont des exemples d'outils orientés tâche. Chacun est taillé pour un objectif spécifique (visualiser l'historique et créer des *commits*, respectivement) en omettant certaines fonctionnalités non-nécessaires à cette tâche.

GitHub pour Mac et Windows

GitHub a créé deux clients Git orientés flux de travail : un pour Windows et un pour Mac. Ces clients sont un bon exemple d'outils orientés flux de travail — plutôt que d'exposer *toutes* les fonctionnalités de Git, ils se focalisent sur un sous-ensemble sélectionné de fonctions couramment utilisées qui s'accordent. Ils ressemblent à ceci :

Figure 1-3.

GitHub pour Mac.

Figure 1-4.

GitHub pour Windows.

Ils sont pensés pour se ressembler et fonctionner de la même manière, donc nous les traiterons comme un seul produit dans ce chapitre. Nous n'allons pas explorer en profondeur ces outils (ils ont leur propre documentation), mais un tour rapide de la vue « changes » (qui est celle sur laquelle vous passerez le plus de temps) montre que tout est en ordre.

- Sur la gauche, il y a la liste des dépôts que le client suit ; vous pouvez ajouter un dépôt (soit en le clonant, soit en l'attachant localement) en cliquant sur l'icône « + » en haut de la zone.
- Au centre, il y a la zone d'entrée de *commit* qui vous permet d'entrer un message de validation et de sélectionner les fichiers qui devraient être inclus. Sous Windows, l'historique de validation est affiché directement en dessous ; sous Mac, c'est un onglet séparé.
- À droite, il y a une vue de diff qui montre ce qui a changé dans votre répertoire de travail ou les modifications qui ont été incluses dans le *commit* sélectionné.
- La dernière chose à noter est le bouton « Sync » en haut à droite qui est le moyen principal d'interagir via le réseau.

Vous n'avez pas besoin d'un compte GitHub pour utiliser ces outils. Bien qu'ils soient pensés pour mettre en lumière le service et la méthode de travail de GitHub, ils fonctionneront parfaitement avec n'importe quel dépôt et dialogueront sans problème avec n'importe quel serveur Git.

INSTALLATION

GitHub pour Windows peut être téléchargé depuis ***https://windows.github.com*** et GitHub pour Mac depuis ***https://mac.github.com***. Quand les applications sont lancées pour la première fois, elles vous guident à travers le paramétrage initial de Git, tel que la configuration de votre nom et de votre adresse de courriel, et toutes deux définissent un paramétrage par défaut sans danger concernant différentes options de configuration telles que les caches d'identification et le comportement pour les fins de ligne.

Les deux applications sont « permanentes » — les mises à jour sont téléchargées et appliquées en tâche de fond pendant que l'application est ouverte. Elles incluent une version autonome de Git, ce qui signifie que vous n'aurez probablement pas à vous soucier de le mettre à jour. Sous Windows, le client inclut un raccourci pour lancer Powershell avec Posh-git, dont nous parlerons plus loin.

L'étape suivante consiste à fournir à l'outil des dépôts avec lesquels travailler. Le client vous affiche une liste de dépôts auxquels vous avez accès sur GitHub et que vous pouvez cloner en une étape. Si vous avez déjà un dépôt local, glissez son répertoire depuis le Finder ou Windows Explorer dans la fenêtre de client GitHub et il sera inclus dans la liste des dépôts.

GESTION RECOMMANDÉE DU TRAVAIL

Une fois installé et configuré, le client GitHub peut être utilisé pour de nombreuses tâches usuelles. La manière de travailler avec cet outil est quelques fois appelée le « *GitHub Flow* ». Nous traitons ceci plus en détail dans **"Processus GitHub"**, mais l'esprit général est que a) vous validerez vos modifications dans une branche et b) vous synchroniserez avec un dépôt distant assez régulièrement.

La gestion de branche est un des domaines pour lesquels les deux outils divergent. Sous Mac, il y a un bouton en haut de la fenêtre pour créer de nouvelles branches :

Figure 1-5.

Le bouton « Create Branch » sous Mac.

Sous Windows, cela se fait en tapant le nom de la branche dans la zone de saisie de bascule de branche :

Figure 1-6.

Créer une branche sous Windows.

Une fois votre branche créée, ajouter des nouveaux *commits* est assez simple. Faites des modifications dans votre répertoire de travail et quand vous revenez dans la fenêtre du client GitHub, elle vous indiquera quels fichiers ont changé. Entrez un message de validation, sélectionnez les fichiers que vous souhaitez inclure et cliquez sur le bouton « Commit » (ctrl-entrée ou ⌘-entrée).

La manière principale d'interagir avec les autres dépôts à travers le réseau passe par la fonctionnalité « Sync ». Git dispose d'opérations différentes pour pousser, récupérer, fusionner et rebaser, mais les clients GitHub les réduisent en une seule fonctionnalité à plusieurs étapes. Voici ce qui se passe quand vous cliquez sur le bouton « *Sync* » :

1. `git pull --rebase`. Si cela échoue à cause d'un conflit de fusion, revenir à `git pull --no-rebase`.
2. `git push`.

C'est la séquence la plus habituelle pour les commandes de réseau quand vous travaillez dans ce mode, donc les réduire à une seule commande fait gagner du temps.

RÉSUMÉ

Ces outils sont très adaptés au mode de travail pour lequel ils sont pensés. Les développeurs, mais aussi les non-développeurs peuvent collaborer sur un projet en peu de temps et un grand nombre des meilleures pratiques pour ce genre de flux de travail sont saisies dans les outils. Cependant, si votre gestion du développement est différente ou si vous voulez plus de contrôle sur comment et

quand les opérations de réseau sont réalisées, nous vous recommandons d'utiliser un autre client ou la ligne de commande.

Autres GUIs

Il existe un certain nombre d'autres clients Git graphiques et ils s'étendent depuis les outils très spécialisés, à fonction unique jusqu'aux applications qui cherchent à exposer tout ce que Git peut faire. Le site officiel Git présente une liste sélectionnée des clients les plus populaires sur ***http://git-scm.com/downloads/guis***. Une liste plus longue est disponible sur le site du wiki Git sur ***https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces***.

Git dans Visual Studio

À partir de Visual Studio 2013 Update 1, les utilisateurs Visual Studio disposent d'un client Git directement intégré dans leur environnement de développement. Visual Studio a proposé la fonctionnalité d'intégration de gestion de source depuis un certain temps, mais celle-ci était principalement orientée vers les systèmes centralisés avec verrouillage de fichier, et Git ne correspondait pas à ce mode d'opération. Le support de Git dans Visual Studio 2013 a été séparé de l'ancienne fonctionnalité et cela aboutit à une bien meilleure intégration entre Studio et Git.

Pour localiser cette fonctionnalité, ouvrez une solution contrôlée par Git (ou lancez juste `git init` dans une solution existante) et sélectionnez *View* > *Team Explorer* depuis le menu. Vous voyez la perspective « *Connect* » qui ressemble à ceci :

Figure 1-7.

Connexion à un dépôt Git depuis Team Explorer.

Visual Studio se souvient de tous les projets gérés par Git que vous avez ouverts et ceux-ci sont disponibles dans la liste du bas. Si vous ne voyez pas celui que vous souhaitez, cliquez sur le lien « Ajouter » et tapez le chemin vers le répertoire de travail. Double-cliquer sur un dépôt Git local vous ouvre la perspective *Accueil* qui ressemble à **Figure A-8** C'est le concentrateur de toutes les actions Git ; quand vous *écrivez* du code, vous passerez probablement la majorité de votre temps dans la vue « Modifications » mais quand viendra le temps de tirer les modifications de vos équipiers, vous utiliserez les vues « Unsynced Commits » et « Branches ».

Figure 1-8.

La vue « Accueil » pour un dépôt Git dans Visual Studio.

Visual Studio propose une interface utilisateur orientée tâche puissante pour Git. Il inclut une visualisation linéaire d'historique, une visualisation de diff, des commandes distantes et de nombreuses autres capacités. Pour une documentation complète sur cette fonctionnalité (qui ne tiendrait pas ici), rendez-vous sur ***http://msdn.microsoft.com/en-us/library/hh850437.aspx***.

Git dans Eclipse

Eclipse est livré avec un greffon appelé Egit qui fournit une interface assez complète pour les opérations Git. On y accède en basculant vers la Perspective Git (*Window* > *Open Perspective* > *Other…* et sélectionner « Git »).

Figure 1-9.

L'environnement EGit de Eclipse.

EGit possède une documentation fournie et de qualité que vous pouvez trouver sous *Help* > *Help Contents* puis en choisissant le nœud « EGit Documentation » dans la liste de contenu.

Git dans Bash

Si vous êtes un utilisateur de Bash, vous pouvez utiliser une des fonctionnalités de l'invite pour rendre votre utilisation de Git plus agréable. Git est fourni avec des greffons pour différentes invites, mais ils ne sont pas activés par défaut.

Premièrement, vous devez vous procurer une copie du fichier `contrib/completion/git-completion.bash` depuis le code source Git. Copiez-le dans un endroit pratique, tel que votre répertoire personnel et ajoutez ceci à votre fichier `.bashrc` :

```
. ~/git-completion.bash
```

Une fois cela fait, déplacez-vous dans un répertoire git et tapez :

```
$ git chec<tab>
```

…et Bash va achever automatiquement en `git checkout`. Cela fonctionne avec toutes les sous-commandes de Git, le paramètres de la ligne de commande et les noms de références quand c'est approprié.

C'est aussi utile pour modifier l'invite pour qu'elle affiche des informations à propos du dépôt Git du répertoire courant. Elles peuvent être aussi simples ou complexes que vous le souhaitez, mais il y a généralement quelques informations clés qui intéressent tout le monde, telles que la branche actuelle et l'état du répertoire de travail. Pour ajouter celles-ci à votre invite, copiez juste le fichier `contrib/completion/git-prompt.sh` depuis le dépôt source de Git dans votre répertoire personnel, ajoutez quelque chose comme ceci à votre `.bashrc` :

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

Le symbole `\w` signifie affiche le répertoire de travail courant, `\$` affiche la partie `$` de l'invite et `__git_ps1 " (%s)"` appelle la fonction fournie par `git-prompt.sh` avec un argument de formatage Ainsi votre invite bash ressemblera à ceci quand vous êtes quelque part dans un projet géré par Git :

Figure 1-10.

Invite bash personnalisée.

Ces deux scripts ont une documentation ; jetez un œil au contenu de `git-completion.bash` et `git-prompt.sh` pour plus d'information.

Git dans Zsh

Git est aussi livré avec une bibliothèque d'achèvement sur tabulation pour Zsh. Copiez simplement `contrib/completion/git-completion.zsh` dans votre

répertoire personnel et sourcez-le depuis votre `.zshrc`. L'interface de Zsh est un peu plus puissante que celle de Bash :

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

Les achèvements possibles ne sont pas seulement listés ; ils sont complétés par des descriptions utiles et vous pouvez naviguer graphiquement dans la liste en répétant tab de manière répétée. Ceci fonctionne avec les commandes Git, les arguments et les noms des choses dans le dépôt (telles que les références et les serveurs distants), ainsi que les noms de fichiers et les autres choses que Zsh sait achever.

Zsh peut être assez compatible avec Bash en ce qui concerne la personnalisation de l'invite, mais elle vous permet d'avoir aussi une partie droite de l'invite. Pour inclure le nom de la branche sur la partie droite, ajoutez ces lignes à votre fichier `~/.zshrc` :

```
setopt prompt_subst
. ~/git-prompt.sh
export RPROMPT=$'$(__git_ps1 "%s")'
```

Cela aboutit à un affichage de la branche actuelle sur le côté droit de la fenêtre de terminal quand votre invite est dans un répertoire Git, ce qui ressemble à ceci :

Figure 1-11.

L'invite zsh personnalisée.

Zsh est assez puissant pour qu'il existe des extensions complètes pour le rendre encore meilleur. Une d'entre elles s'appelle "oh-my-zsh" et on peut la trouver sur ***https://github.com/robbyrussell/oh-my-zsh***. Le système de greffons de oh-my-zsh est doté d'un achèvement sur tabulation pour git puissant et dispose d'une série de thèmes d'invites, dont de nombreux affichent l'information de gestion de version. **Figure A-12** n'est qu'un exemple de ce qui peut être fait avec ce système.

Figure 1-12.

Un exemple d'un thème oh-my-zsh.

Git dans Powershell

Le terminal de ligne de commande standard sous Windows (`cmd.exe`) n'est pas vraiment capable de fournir une expérience de Git personnalisée, mais si vous

utilisez Powershell, vous avez de la chance. Un paquetage appelé Posh-Git (***https://github.com/dahlbyk/posh-git***) fournit un système d'achèvement sur tabulation, ainsi qu'une invite améliorée pour vous aider à rester informé de l'état de votre dépôt. Elle ressemble à ceci :

Figure 1-13.

Powershell avec Posh-git.

Si vous avez installé GitHub pour Windows, Posh-Git est inclus par défaut et tout ce que vous avez à faire est d'ajouter ces lignes à votre `profile.ps1` (qui est habituellement dans `C:\Users\<username>\Documents\WindowsPowerShell`) :

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

Si vous n'êtes pas un utilisateur de GitHub pour Windows, téléchargez simplement une distribution depuis (***https://github.com/dahlbyk/posh-git***) et décompressez-la dans le répertoire `WindowsPowershell`. Ensuite, ouvrez une invite Powershell en tant qu'administrateur et faites ceci :

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

Cela ajoutera la ligne correcte à votre fichier `profile.ps1` et posh-git sera actif la prochaine fois que vous ouvrirez une invite.

Résumé

Vous avez appris à utiliser la puissance de Git depuis les outils que vous utilisez habituellement, ainsi qu'à accéder aux dépôts Git depuis vos propres programmes.

Embarquer Git dans vos applications

B

Si votre application est destinée aux développeurs, il y a de grandes chances qu'elle bénéficierait de l'intégration avec une gestion de source. Même des applications pour non-développeurs, tels que les éditeurs de document, pourraient potentiellement bénéficier d'une fonctionnalité de gestion de version et le modèle de Git fonctionne très bien dans de nombreux scénarios.

Si vous devez intégrer Git avec votre application, vous avez essentiellement trois choix : démarrer une invite et utiliser la ligne de commande Git, Libgit2 et JGit.

Git en ligne de commande

Une possibilité consiste à démarrer un processus d'invite de commande et à utiliser la ligne de commande Git pour agir. Cette méthode a le bénéfice d'être canonique et toutes les fonctionnalités de Git sont supportées. Cela s'avère aussi assez facile, du fait que la plupart des environnements d'exécution disposent d'une interface relativement simple permettant d'invoquer un processus avec des arguments en ligne de commande. Cependant, cette approche a quelques inconvénients.

L'un est que toutes les sorties sont en pur texte. Cela signifie que vous aurez à analyser le format de sortie de Git qui peut occasionnellement changer pour lire l'information d'avancée et de résultat, ce qui peut être inefficace et introduire des erreurs.

Un autre est l'absence de récupération sur erreur. Si le dépôt est corrompu d'une manière quelconque ou si l'utilisateur a une valeur de configuration malformée, Git va simplement refuser d'opérer beaucoup d'opérations.

Un autre encore est la gestion de processus. Git vous oblige à maintenir un environnement de ligne de commande dans un processus séparé, ce qui peut ajouter une complexité indésirable. Essayer de coordonner un certain nombre

de tels processus est un problème épineux (spécialement quand on accède au même dépôt depuis plusieurs processus).

Libgit2

Une autre option à votre disposition consiste à utiliser Libgit2. Libgit2 est une mise en œuvre de Git sans dépendance externe, qui se focalise sur une interface de programmation agréable à utiliser depuis d'autres programmes. Vous pouvez la trouver sur ***http://libgit2.github.com***.

Voyons d'abord à quoi ressemble l'API C. En voici un tour rapide :

```
// Ouvrir un depot
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereferencer HEAD vers un commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// afficher quelques proprietes du commit
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Nettoyer
git_commit_free(commit);
git_repository_free(repo);
```

Les deux premières lignes ouvrent un dépôt Git. Le type `git_repository` représente un identificateur de dépôt avec un cache en mémoire. C'est la méthode la plus simple, quand vous connaissez le chemin exact vers le répertoire de travail ou le répertoire `.git` d'un dépôt. Il y a aussi `git_repository_open_ext` qui inclut des options pour chercher, `git_clone` et ses déclinaisons pour créer un clone local d'un dépôt distant et `git_repository_init` pour créer un dépôt entièrement nouveau.

Le second bloc de code utilise la syntaxe « rev-parse » (voir **"Références de branches"** pour plus de détails) pour obtenir le *commit* sur lequel HEAD peut pointer. Le type retourné est un pointeur sur `git_object` qui représente quelque chose qui existe dans la base de données des objets de Git pour un dépôt. `git_object` est en fait une type « parent » pour différentes sortes d'objets ; l'agencement en mémoire de chacun de ces types « enfants » est identique à celui de `git_object`, donc on peut forcer la conversion vers le type désiré en

toute sécurité. Dans notre cas, `git_object_type(commit)` retournerait `GIT_OBJ_COMMIT`, il est donc permis de le convertir en un pointeur de `git_commit`.

Le bloc suivant montre comment accéder aux propriétés d'un *commit*. La dernière ligne utilise un type `git_oid` ; c'est la représentation d'une empreinte SHA-1 dans Libgit2.

De cet exemple, une structure générale commence à émerger :

- Si vous déclarez un pointeur et que vous en passez une référence dans un appel à Libgit2, cet appel renverra vraisemblablement un code de retour entier. Une valeur `0` indique un succès ; toute valeur négative est une erreur.
- Si Libgit2 peuple un pointeur pour vous, vous êtes responsable de sa libération.
- Si Libgit2 retourne un pointeur `const` après un appel, vous n'avez pas besoin de le libérer mais il deviendra invalide quand l'objet qui le possède sera lui-même libéré.
- Écrire en C est un exercice plutôt douloureux.

Cette dernière remarque signifie qu'il est fort peu probable que vous écrirez du C pour utiliser Libgit2. Heureusement, il existe un certain nombre de liaisons vers d'autres langages qui rendent plus facile l'interaction avec des dépôts Git depuis votre environnement et votre langage spécifiques. Voyons l'exemple ci-dessus réécrit en utilisant le portage Ruby de Libgit2, appelé Rugged et qui peut être trouvé sur ***https://github.com/libgit2/rugged***.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Tout de suite, le code est moins verbeux. Déjà, Rugged utilise des exceptions ; il peut lever `ConfigError` ou `ObjectE` pour signaler des conditions d'erreur. Ensuite, il n'y a pas de libération explicite des ressources, puisque Ruby utilise un ramasse-miettes. Voyons un exemple légèrement plus compliqué : créer un *commit* à partir de rien.

```
blob_id = repo.write("Blob contents", :blob) ❶

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ❷
```

```
sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo), ❸
    :author => sig,
    :committer => sig, ❹
    :message => "Add newfile.txt", ❺
    :parents => repo.empty? ? [] : [ repo.head.target ].compact, ❻
    :update_ref => 'HEAD', ❼
)
commit = repo.lookup(commit_id) ❽
```

❶ Créer un nouveau blob qui contient le contenu d'un nouveau fichier.

❷ Peupler l'index avec l'arbre du *commit* HEAD et ajouter le nouveau fichier sous le chemin `newfile.txt`.

❸ Ceci crée un nouvel arbre dans la base de données des objets et l'utilise pour le nouveau *commit*.

❹ Nous utilisons la même signature pour l'auteur et le validateur.

❺ Le message de validation.

❻ À la création d'un *commit*, il faut spécifier les parents du nouveau *commit*. on utilise le sommet de HEAD comme parent unique.

❼ Rugged (et Libgit2) peuvent en option mettre à jour la référence lors de la création du *commit*.

❽ La valeur retournée est une empreinte SHA-1 du nouvel objet *commit* que vous pouvez alors utiliser pour obtenir un objet `Commit`.

Le code Ruby est joli et propre, mais comme Libgit2 réalise le gros du travail, il tourne aussi plutôt rapidement. Si vous n'êtes pas rubyiste, nous aborderons d'autres portages dans **"Autres liaisons"**.

Fonctionnalité avancée

Libgit2 a certaines capacités qui ne sont pas disponibles dans Git natif. Un exemple est la possibilité de greffons : Libgit2 vous permet de fournir des services

« d'arrière-plan » pour différents types d'opérations, pour vous permettre de stocker les choses d'une manière différente de Git. Libgit2 autorise des services d'arrière-plan pour la configuration, le stockage des références et la base de données d'objets, entre autres.

Voyons comment cela fonctionne. Le code ci-dessous est emprunté à un ensemble d'exemples de services fourni par l'équipe Libgit2 (qui peut être trouvé sur ***https://github.com/libgit2/libgit2-backends***). Voici comment un service d'arrière-plan pour une base de données d'objets peut être créée :

```
git_odb *odb;
int error = git_odb_new(&odb); ❶

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*…*/); ❷

error = git_odb_add_backend(odb, my_backend, 1); ❸

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb); ❹
```

(Notez que les erreurs sont capturées, mais ne sont pas gérées. Nous espérons que votre code est meilleur que le nôtre).

❶ Initialise une enveloppe d'interface d'une base de données d'objets vide (ODB) qui agit comme un conteneur pour les tâches de fond qui feront le vrai travail.

❷ Initialise une tâche de fond ODB.

❸ Ajoute la tâche de fond dans l'enveloppe.

❹ Ouvre le dépôt et le paramètre pour utiliser notre ODB pour rechercher les objets.

Mais qu'est-ce que ce `git_odb_backend_mine` ? Hé bien, c'est le constructeur de notre propre réalisation de l'ODB et nous pouvons la faire comme nous voulons tant que la structure `git_odb_backend` est correctement renseignée. Voici à quoi elle *pourrait* ressembler :

```
typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;
```

```
int git_odb_backend_mine(git_odb_backend **backend_out, /*…*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = …;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // …

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

La contrainte la plus subtile ici est que le premier membre de `my_backend_structure` doit être une structure `git_odb_backend` ; cela assure que la disposition en mémoire correspond à ce qu'attend le code de Libgit2. Le reste est totalement arbitraire ; cette structure peut être aussi grande ou petite que nécessaire.

La fonction d'initialisation alloue de la mémoire pour la structure, initialise le contexte spécifique, puis remplit les membres de la structure `parent` qu'elle supporte. Référez-vous au fichier `include/git2/sys/odb_backend.h` dans les sources de Libgit2 pour l'intégralité des signatures d'appels ; votre cas d'utilisation particulier vous permettra de déterminer lesquelles vous souhaitez supporter.

Autres liaisons

Libgit2 dispose de liaisons vers de nombreux langages. Nous allons montrer ici un petit exemple en utilisant quelques-unes des liaisons les plus abouties au moment de la rédaction de ce livre ; des bibliothèques existent pour de nombreux autres langages qui incluent C++, Go, Node.js, Erlang et la JVM à différents stades de maturité. La collection officielle de liaisons peut être trouvée en parcourant les dépôts sur ***https://github.com/libgit2***.

Le code que nous allons écrire retournera le message de validation du *commit* finalement pointé par HEAD (`git log -1` en quelque sorte).

LIBGIT2SHARP

Si vous écrivez une application .NET ou Mono, LigGit2Sharp (***https://github.com/libgit2/libgit2sharp***) est tout ce que vous cherchez. Les liaisons sont écrites en C# et une grande attention a été portée à envelopper les appels directs à Libgit2 avec une interface de programmation naturelle en C#. Voici à quoi notre programme d'exemple ressemble :

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Pour les applications graphiques Windows, il existe même un paquet NuGet qui vous permettra de démarrer vos développements rapidement.

OBJECTIVE-GIT

Si votre application tourne sur une plateforme Apple, vous avez de grandes chances d'utiliser Objective-C comme langage de programmation. Objective-Git (***https://github.com/libgit2/objective-git***) est le nom de la liaison de Libgit2 pour cet environnement.

Le programme d'exemple ressemble à ceci :

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git est totalement interopérable avec Swift, donc n'ayez crainte si vous avez abandonné Objective-C.

PYGIT2

La liaison avec Libgit2 en Python s'appelle Pygit2 et elle peut être trouvée sur ***http://www.pygit2.org/***. Notre programme d'exemple :

```
pygit2.Repository("/chemin/du/depot") # ouvre le depot
    .head                             # recupere la branche en cours
    .peel(pygit2.Commit)              # descend au commit
    .message                          # lit le message
```

Pour aller plus loin

Bien sûr, un traitement complet des capacités de Libgit2 est hors du cadre de ce livre. Si vous souhaitez plus d'information sur Libgit2 elle-même, la documentation de programmation se trouve sur ***https://libgit2.github.com/libgit2*** et un ensemble de guides sur ***https://libgit2.github.com/docs***. Pour les autres liai-

sons, cherchez dans le README et dans les tests ; il y a souvent des petits didacticiels et des pointeurs sur d'autres documents.

JGit

Si vous voulez utiliser Git depuis un programme Java, il existe une bibliothèque complète appelée JGit. JGit est une réalisation relativement complète de Git écrite nativement en Java etelle est largement utilisée dans la communauté Java. Le projet JGit est développé sous l'égide d'Eclipse et son site se trouve sur ***http://www.eclipse.org/jgit***.

Mise en place

Il y a différents moyens de connecter votre projet à JGit et de commencer à l'utiliser dans votre code. La manière probablement la plus facile consiste à utiliser Maven – on réalise l'intégration en ajoutant la section suivante sous la balise `<dependencies>` de votre fichier pom.xml :

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

La `version` aura très certainement évolué lorsque vous lirez ces lignes ; vérifiez ***http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit*** pour une information de version mise à jour. Une fois cette étape accomplie, Maven va automatiquement récupérer et utiliser les bibliothèques JGit dont vous aurez besoin.

Si vous préférez gérer les dépendances binaires par vous-même, des binaires JGit pré-construits sont disponibles sur ***http://www.eclipse.org/jgit/download***. Vous pouvez les inclure dans votre projet en lançant une commande telle que :

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Plomberie

JGit propose deux niveaux généraux d'interfaçage logiciel : plomberie et porcelaine. La terminologie de ces niveaux est directement calquée sur celle de Git

lui-même et JGit est partitionné globalement de la même manière : les APIs de porcelaine sont une interface de haut niveau pour des interactions de niveau utilisateur (le genre de choses qu'un utilisateur normal ferait en utilisant la ligne de commande), tandis que les APIs de plomberie permettent d'interagir directement avec les objets de bas niveau du dépôt.

Le point de départ pour la plupart des sessions JGit est la classe `Repository` et la première action consiste à créer une instance de celle-ci. Pour un dépôt basé sur un système de fichier (hé oui, JGit permet d'autre modèles de stockage), cela passe par l'utilisation d'un `FileRepositoryBuilder` :

```
// Creer un nouveau depot
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Ouvrir un depot existant
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

Le constructeur contient une API souple qui fournit tout ce dont il a besoin pour trouver un dépôt Git, que votre programme sache ou ne sache pas exactement où il est situé. Il peut utiliser des variables d'environnement (`.readEnvironment()`), démarrer depuis le répertoire de travail et chercher (`setWorkTree(…).findGitDir()`) ou juste ouvrir un répertoire `.git` connu, comme ci-dessus.

Une fois muni d'une instance de `Repository`, vous pouvez faire toutes sortes de choses avec. En voici un échantillon rapide :

```
// acceder a une reference
Ref master = repo.getRef("master");

// acceder a l'objet pointe par une reference
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Charger le contenu brut d'un objet
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Creer une branche
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();
```

```
// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

Il y a pas mal de choses mises en œuvre ici, donc nous allons détailler chaque section.

La première ligne récupère un pointeur sur la référence `master`. JGit récupère automatiquement la référence master *réelle* qui se situe dans `refs/heads/master` et retourne un objet qui vous permet de récupérer des informations sur cette référence. Vous pouvez récupérer le nom (`.getName()`) ou bien l'objet cible de la référence directe (`.getObjectId()`), ou encore la référence pointée par une référence symbolique (`.getTarget()`). Les objets de référence sont aussi utilisés pour représenter les références d'étiquette ou des objets, donc vous pouvez demander si l'étiquette est « pelée », ce qui signifie qu'elle pointe sur une cible finale d'une série (potentiellement longue) d'objets étiquettes.

La seconde ligne donne la cible de la référence `master` qui est retournée comme une instance d'ObjectId. ObjectId représente l'empreinte SHA-1 d'un objet qui peut exister ou non dans la base de données des objets de Git. La troisième ligne est similaire, mais elle montre comment JGit gère la syntaxe « rev-parse » (pour plus d'information, voir **"Références de branches"**). Vous pouvez passer n'importe quel spécificateur d'objet que Git comprend et JGit retournera un ObjectId valide ou bien `null`.

Les deux lignes suivantes montrent comment charger le contenu brut d'un objet. Dans cet exemple, nous appelons `ObjectLoader.copyTo()` pour rediriger le contenu de l'objet directement sur la sortie standard, mais `ObjectLoader` dispose aussi de méthodes pour lire le type et la taille d'un objet et le retourner dans un tableau d'octets. Pour les gros objets (pour lesquels `.isLarge()` renvoie `true`), vous pouvez appeler `.openStream()` pour récupérer un objet similaire à InputStream qui peut lire l'objet brut sans le tirer intégralement en mémoire vive.

Les quelques lignes suivantes montrent ce qui est nécessaire pour créer une nouvelle branche. Nous créons une instance de RefUpdate, configurons quelques paramètres et appelons `.update()` pour déclencher la modification. Le code pour effacer cette branche suit juste après. Notez que `.setForceUp-`

`date(true)` est nécessaire pour que cela fonctionne ; sinon l'appel à `.delete()` retourne `REJECTED` et il ne se passera rien.

Le dernier exemple montre comment récupérer la valeur `user.name` depuis les fichiers de configuration de Git. Cette instance de Config utilise le dépôt que nous avons ouvert plus tôt pour la configuration locale, mais détectera automatiquement les fichiers de configuration globale et système et y lira aussi les valeurs.

Ceci n'est qu'un petit échantillon de toute l'API de plomberie ; il existe beaucoup d'autre méthodes et classes. Ici, nous ne montrons pas non plus comment JGit gère les erreurs, au moyen d'exceptions. Les APIs JGit lancent quelques fois des exceptions Java standard (telles que `IOException`), mais il existe aussi une liste de types d'exception spécifiques à JGit (tels que `NoRemoteRepositoryException`, `CorruptObjectException` et `NoMergeBaseException`).

Porcelaine

Les APIs de plomberie sont plutôt complètes, mais il peut s'avérer lourd de les enchaîner pour des activités fréquentes, telles que l'ajout de fichier à l'index ou la validation. JGit fournit un ensemble de plus haut niveau d'APIs pour simplifier celles-ci et le point d'entrée pour ces APIs est la classe `Git` :

```
Repository repo;
// construit le depot...
Git git = new Git(repo);
```

La classe Git propose un joli ensemble de méthodes de haut niveau de style `builder` qui peuvent être utilisées pour construire des comportements assez complexes. Voyons un exemple, tel que recréer quelque chose comme `git ls-remote` :

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}
```

C'est la structure habituelle avec la classe Git ; les méthodes renvoient un objet commande qui permet d'enchaîner les appels de paramétrage qui sont fi-

nalement exécutés par l'appel à `.call()`. Dans notre cas, nous interrogeons le dépôt distant `origin` sur ses étiquettes, et non sur ses sommets de branches. Notez aussi l'utilisation d'un objet `CredentialsProvider` pour l'authentification.

De nombreuses autres commandes sont disponibles au travers de la classe Git, dont entre autres `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert` et `reset`.

Pour aller plus loin

Tout ceci n'est qu'un mince échantillon de toutes les capacités de JGit. Si vous êtes intéressé et souhaitez en savoir plus, voici des liens vers plus d'information et d'inspiration :

- La documentation officielle de l'API JGit est disponible sur ***http://download.eclipse.org/jgit/docs/latest/apidocs***. Il existe des Javadoc standard, donc votre EDI JVM favori sera aussi capable de les installer localement.
- Le livre de recettes JGit ***https://github.com/centic9/jgit-cookbook*** contient de nombreux exemples de réalisation de différentes tâches avec JGit.
- Il existe quelques bonnes ressources énumérées sur ***http://stackoverflow.com/questions/6861881***.

Commandes Git C

Tout au long du livre, nous avons introduit des dizaines de commandes Git et avons essayé le plus possible de les introduire de manière narrative, en ajoutant progressivement des commandes dans l'histoire. Cependant, cela nous laisse avec des exemples d'utilisation des commandes quelque peu dispersés tout au long du livre.

Dans cette annexe, nous allons revoir toutes les commandes Git qui se trouvent dans le livre, regroupées à peu près par leur usage. Nous allons parler de ce que chaque commande fait très généralement et nous indiquerons où vous pouvez les retrouver dans le livre.

Installation et configuration

Il y a deux commandes qui sont les plus utilisées, depuis les premières invocations de Git jusqu'à la mise au point et à la gestion quotidienne des branches, ce sont les commandes `config` et `help`.

git config

Git a une façon par défaut de faire des centaines de choses. Pour beaucoup de ces choses, vous pouvez demander à Git de les faire par défaut d'une autre manière ou de définir vos préférences. Cela implique tout depuis dire à Git quel est votre nom jusqu'aux préférences spécifiques des couleurs du terminal ou de l'éditeur que vous utilisez. Il y a plusieurs fichiers que cette commande lira et dans lesquels elle écrira de façon à définir des valeurs globales ou spécifiques à des dépôts.

La commande `git config` a été utilisée dans presque tous les chapitres du livre.

Dans **"Paramétrage à la première utilisation de Git"**, nous l'avons utilisée pour indiquer notre nom, adresse de courrier électronique et la préférence d'éditeur avant même que nous commencions à utiliser Git.

Dans **“Les alias Git”**, nous avons montré comment vous pourriez l’utiliser pour créer des commandes raccourcies qui se développent en longues séquences d’options pour que vous n’ayez pas à les taper à chaque fois.

Dans **“Rebaser (*Rebasing*)”**, nous l’avons utilisée pour faire de `--rebase` le comportement par défaut quand vous lancez `git pull`.

Dans **“Stockage des identifiants”**, nous l’avons utilisée pour définir un stockage par défaut pour vos mots de passe HTTP.

Dans **“Expansion des mots-clés”**, nous avons montré comment définir des filtres « smudge » et « clean » sur le contenu entrant et sortant de Git.

Enfin, la totalité de **“Configuration de Git”** est dédiée à cette commande.

git help

La commande `git help` est utilisée pour vous montrer toute la documentation livrée avec Git à propos de n’importe quelle commande. Bien que nous donnions un panorama général des plus populaires d’entre elles dans cette annexe, pour une liste complète de toutes les options possibles pour chaque commande, vous pouvez toujours lancer `git help <commande>`.

Nous avons introduit la commande `git help` dans **“Obtenir de l’aide”** et vous avons montré comment l’utiliser pour trouver plus d’informations sur le `git shell` dans **“Mise en place du serveur”**.

Obtention et création des projets

Il y a deux façons de récupérer un dépôt Git. L’une consiste à le copier d’un dépôt existant sur le réseau ou ailleurs et l’autre est d’en créer un nouveau dans un dossier existant.

git init

Pour transformer un dossier en un nouveau dépôt Git afin que vous puissiez commencer sa gestion de version, vous pouvez simplement lancer `git init`.

Nous l’avons d’abord présentée dans **“Démarrer un dépôt Git”**, où nous avons montré la création d’un tout nouveau dépôt pour commencer à l’utiliser.

Nous parlons brièvement de la façon dont vous pourriez changer la branche par défaut depuis « master » dans **“Branches distantes”**.

Nous utilisons cette commande pour créer un dépôt vierge pour un serveur dans **“Copie du dépôt nu sur un serveur”**.

Enfin, nous rentrons dans quelques détails de ce qu’il fait effectivement en coulisses dans **“Plomberie et porcelaine”**.

git clone

La commande `git clone` sert en fait à englober plusieurs autres commandes. Elle crée un nouveau dossier, va à l'intérieur de celui-ci et lance `git init` pour en faire un dépôt Git vide, ajoute un serveur distant (`git remote add`) à l'URL que vous lui avez passée (appelé par défaut `origin`), lance `git fetch` à partir de ce dépôt distant et ensuite extrait le dernier *commit* dans votre répertoire de travail avec `git checkout`.

La commande `git clone` est utilisée dans des dizaines d'endroits du livre, mais nous allons seulement lister quelques endroits intéressants.

C'est présenté et expliqué en gros dans **"Cloner un dépôt existant"**, où vous trouverez quelques exemples.

Dans **"Installation de Git sur un serveur"**, nous montrons l'utilisation de l'option `--bare` pour créer une copie d'un dépôt Git sans répertoire de travail.

Dans **"Empaquetage (*bundling*)"**, nous l'utilisons pour dépaqueter un dépôt Git empaqueté.

Enfin, dans **"Cloner un projet avec des sous-modules"**, nous apprenons l'option `--recursive` pour rendre le clonage d'un dépôt avec sous-modules un peu plus simple.

Bien qu'elle soit utilisée dans beaucoup d'autres endroits du livre, ceux-là sont ceux qui sont en quelque sorte uniques ou qui sont utilisés de manière un peu différente.

Capture d'instantané basique

Pour le flux de travail basique d'indexation du contenu et de sa validation dans votre historique, il n'y a que quelques commandes basiques.

git add

La commande `git add` ajoute le contenu du répertoire de travail dans la zone d'index pour le prochain *commit*. Quand la commande `git commit` est lancée, par défaut elle ne regarde que cette zone d'index, donc `git add` est utilisée pour réaliser le prochain *commit* exactement comme vous le voulez.

Cette commande est une commande incroyablement importante dans Git et est mentionnée ou utilisée des dizaines de fois dans ce livre. Nous allons revoir rapidement quelques utilisations uniques qui peuvent être trouvées.

Nous avons d'abord présenté et expliqué `git add` en détails dans **"Placer de nouveaux fichiers sous suivi de version"**.

Nous mentionnons comment s'en servir pour résoudre les conflits de fusion dans **"Conflits de fusions (*Merge conflicts*)"**.

Nous la passons en revue pour indexer interactivement seulement des parties spécifiques d'un fichier modifié dans **"Indexation interactive"**.

Enfin, nous l'émulons à bas niveau dans **???** pour que vous ayez une idée de ce qu'elle fait en coulisses.

git status

La commande `git status` vous montrera les différents états des fichiers de votre répertoire de travail et de l'index. Quels sont les fichiers modifiés et non indexés et lesquels sont indexés mais pas encore validés. Dans sa forme normale, elle vous donnera aussi des conseils basiques sur comment passer les fichiers d'un état à l'autre.

Nous couvrons pour la première fois `status` dans **"Vérifier l'état des fichiers"**, à la fois dans sa forme basique et dans sa forme simplifiée. Bien que nos l'utilisions tout au long du livre, pratiquement tout ce que vous pouvez faire avec la commande `git status` y est couvert.

git diff

La commande `git diff` s'utilise lorsque vous voulez voir la différence entre deux arbres. Cela peut être la différence entre votre répertoire de travail et votre index (`git diff` en elle-même), entre votre index et votre dernier *commit* (`git diff --staged`) ou entre deux *commits* (`git diff master brancheB`).

Nous regardons d'abord les utilisations basiques de `git diff` dans **"Inspecter les modifications indexées et non indexées"** où nous montrons comment voir quels changements sont indexés et lesquels ne le sont pas.

Nous l'utilisons pour chercher de possibles problèmes d'espaces blancs avant de valider avec l'option `--check` dans **"Guides pour une validation"**.

Nous voyons comment vérifier les différences entre branches plus efficacement avec la syntaxe `git diff A...B` dans **"Déterminer les modifications introduites"**.

Nous l'utilisons pour filtrer les différences d'espaces blancs avec `-w` et comment comparer différentes étapes de fichiers conflictuels avec `--theirs`, `--ours` et `--base` dans **"Fusion avancée"**.

Enfin, nous l'utilisons pour comparer efficacement les modifications de sous-modules avec `--submodule` dans **"Démarrer un sous-module"**.

git difftool

La commande `git difftool` lance simplement un outil externe pour vous montrer la différence entre deux arbres dans le cas où vous voudriez utiliser quelque chose de différent de la commande `git diff` intégrée.

Nous ne mentionnons ceci que brièvement dans **Git Diff dans un outil externe**.

git commit

La commande `git commit` prend tout le contenu des fichiers qui ont été indexés avec `git add` et enregistre un nouvel instantané permanent dans la base de données puis fait pointer la branche courante dessus.

Nous couvrons d'abord les bases de la validation dans **"Valider vos modifications"**. Là nous montrons aussi comment utiliser l'option `-a` pour sauter l'étape `git add` dans le travail quotidien et comment utiliser l'option `-m` pour passer un message de validation en ligne de commande plutôt que d'ouvrir un éditeur.

Dans **"Annuler des actions"**, nous couvrons l'utilisation de l'option `--amend` pour refaire le *commit* le plus récent.

Dans **"Les branches en bref"**, nous allons plus dans le détail sur ce que `git commit` fait et pourquoi elle le fait ainsi.

Nous avons vu comment signer cryptographiquement les *commits* avec l'option `-S` dans **"Signer des *commits*"**.

Enfin, nous regardons ce que la commande `git commit` fait en coulisses et comment elle est réellement implémentée dans **"Les objets *commit*"**.

git reset

La commande `git reset` est d'abord utilisée pour défaire des choses, comme son nom l'indique. Elle modifie le pointeur `HEAD` et change optionnellement l'index et peut aussi modifier le répertoire de travail si vous l'utilisez avec l'option `--hard`. Cette dernière option rend possible la perte de votre travail par cette commande si elle est mal employée, alors soyez certain de bien la comprendre avant de l'utiliser.

Nous couvrons d'abord effectivement le cas le plus simple de `git reset` dans **"Désindexer un fichier déjà indexé"** où nous l'utilisons pour désindexer un fichier sur lequel nous avons lancé `git add`.

Nous la couvrons ensuite de façon assez détaillée dans **"Reset démystifié"**, qui est entièrement dédié à l'explication de cette commande.

Nous utilisons `git reset --hard` pour annuler une fusion dans **"Abandonner une fusion"**, où nous utilisons aussi `git merge --abort`, qui est un peu un enrobage pour la commande `git reset`.

git rm

La commande `git rm` est utilisée pour supprimer des fichiers de l'index et du répertoire de travail pour Git. Elle est similaire à `git add` dans le sens où elle indexe la suppression d'un fichier pour le prochain *commit*.

Nous couvrons la commande `git rm` dans le détail dans **"Effacer des fichiers"**, y compris en supprimant récursivement les fichiers et en ne supprimant les fichiers que de l'index mais en les laissant dans le répertoire de travail avec `--cached`.

Le seul autre usage différent de `git rm` dans le livre est dans **"Suppression d'objets"** où nous utilisons et expliquons brièvement l'option `--ignore-unmatch` quand nous lançons `git filter-branch`, qui ne sort tout simplement pas d'erreur lorsque le fichier que nous essayons de supprimer n'existe pas. Cela peut être utile dans le but d'utiliser des scripts.

git mv

La commande `git mv` est une commande de faible utilité pour renommer un fichier et ensuite lancer `git add` sur le nouveau fichier et `git rm` sur l'ancien.

Nous ne mentionnons cette commande que brièvement dans **"Déplacer des fichiers"**.

git clean

La commande `git clean` est utilisée pour supprimer les fichiers indésirables de votre répertoire de travail. Cela peut aller de la suppression des fichiers temporaires de compilation jusqu'aux fichiers de conflit de fusion.

Nous couvrons une grande part des options et des scénarios dans lesquels vous pourriez utiliser la commande `clean` dans **"Nettoyer son répertoire de travail"**.

Création de branches et fusion

Il y a une poignée seulement de commandes qui implémentent la plupart des fonctionnalités de branche et de fusion dans Git.

git branch

La commande `git branch` est en fait une sorte d'outil de gestion de branche. Elle peut lister les branches que vous avez, créer une nouvelle branche, supprimer des branches et renommer des branches.

La plus grande partie de **Chapter 3** est dédiée à la commande `branch` et elle est utilisée tout au long du chapitre. Nous la présentons d'abord dans **"Créer une nouvelle branche"** et nous explorons la plupart de ses autres fonctionnalités (listage et suppression) dans **"Gestion des branches"**.

Dans **"Suivre les branches"**, nous utilisons l'option `git branch -u` pour définir une branche de suivi.

Enfin, nous explorons une partie de ce qu'elle fait en arrière-plan dans **"Références Git"**.

git checkout

La commande `git checkout` est utilisée pour passer d'une branche à l'autre et en extraire le contenu dans votre répertoire de travail.

Nous rencontrons cette commande pour la première fois dans **"Basculer entre les branches"** avec la commande `git branch`.

Nous voyons comment l'utiliser pour commencer à suivre des branches avec l'option `--track` dans **"Suivre les branches"**.

Nous nous en servons pour réintroduire des conflits de fichiers avec `--conflict=diff3` dans **"Examiner les conflits"**.

Nous allons plus en détail sur sa relation avec `git reset` dans **"Reset démystifié"**.

Enfin, nous voyons quelques détails d'implémentation dans **"La branche HEAD"**.

git merge

L'outil `git merge` est utilisé pour fusionner une ou plusieurs branches dans la branche que vous avez extraite. Il avancera donc la branche courante au résultat de la fusion.

La commande `git merge` est d'abord présentée dans **"Branches"**. Bien qu'elle soit utilisée à plusieurs endroits du livre, il n'y a que peu de variations de la commande `merge` — généralement juste `git merge <branche>` avec le nom de la seule branche que vous voulez fusionner.

Nous avons couvert comment faire une fusion écrasée (dans laquelle Git fusionne le travail mais fait comme si c'était juste un nouveau *commit* sans enre-

gistrer l'historique de la branche dans laquelle vous fusionnez) à la toute fin de **"Projet public dupliqué"**.

Nous avons exploré une grande partie du processus de fusion et de la commande, y compris la commande `-Xignore-all-whitespace` et l'option `--abort` pour abandonner un problème du fusion dans **"Fusion avancée"**.

Nous avons appris à vérifier les signatures avant de fusionner si votre projet utilise la signature GPG dans **"Signer des *commits*"**.

Enfin, nous avons appris la fusion de sous-arbre dans **"Subtree Merging"**.

git mergetool

La commande `git mergetool` se contente de lancer un assistant de fusion externe dans le cas où vous rencontrez des problèmes de fusion dans Git.

Nous la mentionnons rapidement dans **"Conflits de fusions (*Merge conflicts*)"** et détaillons comment implémenter votre propre outil externe dans **"Outils externes de fusion et de différence"**.

git log

La commande `git log` est utilisée pour montrer l'historique enregistré atteignable d'un projet en partant du *commit* le plus récent. Par défaut, elle vous montrera seulement l'historique de la branche sur laquelle vous vous trouvez, mais elle accepte des branches ou sommets différents ou même multiples comme points de départ de parcours. Elle est aussi assez souvent utilisée pour montrer les différences entre deux ou plusieurs branches au niveau *commit*.

Cette commande est utilisée dans presque tous les chapitres du livre pour exposer l'historique d'un projet.

Nous présentons la commande et la parcourons plus en détail dans **"Visualiser l'historique des validations"**. Là nous regardons les options `-p` et `--stat` pour avoir une idée de ce qui a été introduit dans chaque *commit* et les options `--pretty` et `--oneline` pour voir l'historique de manière plus concise, avec quelques options simples de filtre de date et d'auteur.

Dans **"Créer une nouvelle branche"**, nous l'utilisons avec l'option `--decorate` pour visualiser facilement où se trouvent nos pointeurs de branche et nous utilisons aussi l'option `--graph` pour voir à quoi ressemblent les historiques divergents.

Dans **"Cas d'une petite équipe privée"** et **"Plages de *commits*"**, nous couvrons la syntaxe `brancheA..brancheB` que nous utilisons avec la commande `git log` pour voir quels *commits* sont propres à une branche relativement à une autre branche. Dans **"Plages de *commits*"**, nous explorons cela de manière assez détaillée.

Dans **“Journal de fusion”** et **“Triple point”**, nous couvrons l’utilisation du format `brancheA...brancheB` et de la syntaxe `--left-right` pour voir ce qui est dans une branche ou l’autre mais pas dans les deux à la fois. Dans **“Journal de fusion”**, nous voyons aussi comment utiliser l’option `--merge` comme aide au débogage de conflit de fusion tout comme l’utilisation de l’option `--cc` pour regarder les conflits de *commits* de fusion dans votre historique.

Dans **“Raccourcis RefLog”**, nous utilisons l’option `-g` pour voir le reflog Git à travers cet outil au lieu de faire le parcours de la branche.

Dans **“Recherche”**, nous voyons l’utilisation des options `-S` et `-L` pour faire des recherches assez sophistiquées sur quelque chose qui s’est passé historiquement dans le code comme voir l’historique d’une fonction.

Dans **“Signer des *commits*”**, nous voyons comment utiliser `--show-signature` pour ajouter un message de validation pour chaque *commit* dans la sortie de `git log` basé sur le fait qu’il ait ou qu’il n’ait pas une signature valide.

git stash

La commande `git stash` est utilisée pour remiser temporairement du travail non validé afin d’obtenir un répertoire de travail propre sans avoir à valider du travail non terminé dans une branche.

Elle est entièrement décrite simplement dans **“Remisage et nettoyage”**.

git tag

La commande `git tag` est utilisée pour placer un signet permanent à un point spécifique de l’historique du code. C’est généralement utilisé pour marquer des choses comme des publications.

Cette commande est présentée et couverte en détail dans **“Étiquetage”** et nous la mettons en pratique dans **“Étiquetage de vos publications”**.

Nous couvrons aussi comment créer une étiquette signée avec l’option `-s` et en vérifier une avec l’option `-v` dans **“Signer votre travail”**.

Partage et mise à jour de projets

Il n’y a pas vraiment beaucoup de commandes dans Git qui accèdent au réseau ; presque toutes les commandes agissent sur la base de données locale. Quand vous êtes prêt à partager votre travail ou à tirer les changements depuis ailleurs, il y a une poignée de commandes qui échangent avec les dépôts distants.

git fetch

La commande `git fetch` communique avec un dépôt distant et rapporte toutes les informations qui sont dans ce dépôt qui ne sont pas dans le vôtre et les stocke dans votre base de données locale.

Nous voyons cette commande pour la première fois dans **"Récupérer et tirer depuis des dépôts distants"** et nous continuons à voir des exemples d'utilisation dans **"Branches distantes"**.

Nous l'utilisons aussi dans plusieurs exemples dans **"Contribution à un projet"**.

Nous l'utilisons pour aller chercher une seule référence spécifique qui est hors de l'espace par défaut dans **"Références aux requêtes de tirage"** et nous voyons comment aller chercher depuis un paquet dans **"Empaquetage (*bundling*)"**.

Nous définissons des refspecs hautement personnalisées dans le but de faire faire à `git fetch` quelque chose d'un peu différent que le comportement par défaut dans **"La *refspec*"**.

git pull

La commande `git pull` est essentiellement une combinaison des commandes `git fetch` et `git merge`, où Git ira chercher les modifications depuis le dépôt distant que vous spécifiez et essaie immédiatement de les fusionner dans la branche dans laquelle vous vous trouvez.

Nous la présentons rapidement dans **"Récupérer et tirer depuis des dépôts distants"** et montrons comment voir ce qui sera fusionné si vous la lancez dans **"Inspecter un dépôt distant"**.

Nous voyons aussi comment s'en servir pour nous aider dans les difficultés du rebasage dans **"Rebaser quand vous rebasez"**.

Nous montrons comment s'en servir avec une URL pour tirer ponctuellement les modifications dans **"Vérification des branches distantes"**.

Enfin, nous mentionnons très rapidement que vous pouvez utiliser l'option `--verify-signatures` dans le but de vérifier que les *commits* que vous tirez ont une signature GPG dans **"Signer des *commits*"**.

git push

La commande `git push` est utilisée pour communiquer avec un autre dépôt, calculer ce que votre base de données locale a et que le dépôt distant n'a pas, et ensuite pousser la différence dans l'autre dépôt. Cela nécessite un droit

d'écriture sur l'autre dépôt et donc normalement de s'authentifier d'une manière ou d'une autre.

Nous voyons la commande `git push` pour la première fois dans **"Pousser son travail sur un dépôt distant"**. Ici nous couvrons les bases de la poussée de branche vers un dépôt distant. Dans **"Pousser les branches"**, nous allons un peu plus loin dans la poussée de branches spécifiques et dans **"Suivre les branches"** nous voyons comment définir des branches de suivi pour y pousser automatiquement. Dans **"Suppression de branches distantes"**, nous utilisons l'option `--delete` pour supprimer une branche sur le serveur avec `git push`.

Tout au long de **"Contribution à un projet"**, nous voyons plusieurs exemples d'utilisation de `git push` pour partager du travail sur des branches à travers de multiples dépôts distants.

Nous voyons dans **"Partager les étiquettes"** comment s'en servir avec l'option `--tags` pour partager des étiquettes que vous avez faites.

Dans **"Publier les modifications dans un sous-module"**, nous utilisons l'option `--recurse-submodules` pour vérifier que tout le travail de nos sous-modules a été publié avant de pousser le super-projet, ce qui peut être vraiment utile quand on utilise des sous-modules.

Dans **"Autres crochets côté client"**, nous discutons brièvement du crochet `pre-push`, qui est un script que nous pouvons installer pour se lancer avant qu'une poussée ne s'achève pour vérifier qu'elle devrait être autorisée à pousser.

Enfin, dans **"Pousser des *refspecs*"**, nous considérons une poussée avec une refspec complète au lieu des raccourcis généraux qui sont normalement utilisés. Ceci peut vous aider à être très spécifique sur le travail que vous désirez partager.

git remote

La commande `git remote` est un outil de gestion pour votre base de dépôts distants. Elle vous permet de sauvegarder de longues URLs en tant que raccourcis, comme « origin », pour que vous n'ayez pas à les taper dans leur intégralité tout le temps. Vous pouvez en avoir plusieurs et la commande `git remote` est utilisée pour les ajouter, les modifier et les supprimer.

Cette commande est couverte en détail dans **"Travailler avec des dépôts distants"**, y compris leur listage, ajout, suppression et renommage.

Elle est aussi utilisée dans presque tous les chapitres suivants du livre, mais toujours dans le format standard `git remote add <nom> <URL>`.

git archive

La commande `git archive` est utilisée pour créer un fichier d'archive d'un instantané spécifique du projet.

Nous utilisons `git archive` pour créer une archive d'un projet pour partage dans **"Préparation d'une publication"**.

git submodule

La commande `git submodule` est utilisée pour gérer des dépôts externes à l'intérieur de dépôts normaux. Cela peut être pour des bibliothèques ou d'autres types de ressources partagées. La commande `submodule` a plusieurs sous-commandes (`add`, `update`, `sync`, etc) pour la gestion de ces ressources.

Cette commande est mentionnée et entièrement traitée uniquement dans **"Sous-modules"**.

Inspection et Comparaison

git show

La commande `git show` peut montrer un objet Git d'une façon simple et lisible pour un être humain. Vous l'utiliseriez normalement pour montrer les informations d'une étiquette ou d'un *commit*.

Nous l'utilisons d'abord pour afficher l'information d'une étiquette annotée dans **"Les étiquettes annotées"**.

Plus tard nous l'utilisons un petit peu dans **"Sélection des versions"** pour montrer les *commits* que nos diverses sélections de versions résolvent.

Une des choses les plus intéressantes que nous faisons avec `git show` est dans **"Re-fusion manuelle d'un fichier"** pour extraire le contenu de fichiers spécifiques d'étapes différentes durant un conflit de fusion.

git shortlog

La commande `git shortlog` est utilisée pour résumer la sortie de `git log`. Elle prendra beaucoup des mêmes options que la commande `git log` mais au lieu de lister tous les *commits*, elle présentera un résumé des *commits* groupés par auteur.

Nous avons montré comment s'en servir pour créer un joli journal des modifications dans **"Shortlog"**.

git describe

La commande `git describe` est utilisée pour prendre n'importe quelle chose qui se résoud en un *commit* et produit une chaîne de caractères qui est somme toute lisible pour un être humain et qui ne changera pas. C'est une façon d'obtenir une description d'un *commit* qui est aussi claire qu'un SHA de *commit* mais en plus compréhensible.

Nous utilisons `git describe` dans **"Génération d'un nom de révision"** et **"Préparation d'une publication"** pour obtenir une chaîne de caractères pour nommer notre fichier de publication après.

Débogage

Git possède quelques commandes qui sont utilisées pour aider à déboguer un problème dans votre code. Cela va de comprendre où quelque chose a été introduit à comprendre qui l'a introduite.

git bisect

L'outil `git bisect` est un outil de débogage incroyablement utile utilisé pour trouver quel *commit* spécifique a le premier introduit un bug ou problème en faisant une recherche automatique par dichotomie.

Il est complètement couvert dans **"Recherche dichotomique"** et n'est mentionné que dans cette section.

git blame

La commande `git blame` annote les lignes de n'importe quel fichier avec quel *commit* a été le dernier à introduire un changement pour chaque ligne du fichier et quelle personne est l'auteur de ce *commit*. C'est utile pour trouver la personne pour lui demander plus d'informations sur une section spécifique de votre code.

Elle est couverte dans **"Fichier annoté"** et n'est mentionnée que dans cette section.

git grep

La commande `git grep` peut aider à trouver n'importe quelle chaîne de caractères ou expression régulière dans n'importe quel fichier de votre code source, même dans des anciennes versions de votre projet.

Elle est couverte dans **“Git grep”** et n’est mentionnée que dans cette section.

Patchs

Quelques commandes dans Git sont centrées sur la considération des *commits* en termes de changements qu’ils introduisent, comme si les séries de *commits* étaient des séries de patchs. Ces commandes vous aident à gérer vos branches de cette manière.

git cherry-pick

La commande `git cherry-pick` est utilisée pour prendre les modifications introduites dans un seul *commit* Git et essaye de les réintroduire en tant que nouveau *commit* sur la branche sur laquelle vous vous trouvez. Cela peut être utile pour prendre un ou deux *commits* sur une branche individuellement plutôt que fusionner dans la branche, ce qui prend toutes les modifications.

La sélection de *commits* est décrite et démontrée dans **“Gestion par rebasage et sélection de *commit*”**.

git rebase

La commande `git rebase` est simplement un `cherry-pick` automatisé. Elle détermine une série de *commits* puis les sélectionne et les ré-applique un par un dans le même ordre ailleurs.

Le rebasage est couvert en détail dans **“Rebaser (*Rebasing*)”**, y compris l’étude des problèmes de collaboration induits par le rebasage de branches qui sont déjà publiques.

Nous la mettons en pratique tout au long d’un exemple de scission de votre historique en deux dépôts séparés dans **“Replace”**, en utilisant aussi l’option `--onto`.

Nous explorons un conflit de fusion de rebasage dans **“Rerere”**.

Nous l’utilisons aussi dans un mode de script interactif avec l’option `-i` dans **“Modifier plusieurs messages de validation”**.

git revert

La commande `git revert` est fondamentalement le contraire de `git cherry-pick`. Elle crée un *commit* qui applique l’exact opposé des modifica-

tions introduites par le *commit* que vous ciblez, essentiellement en le défaisant ou en revenant dessus.

Nous l'utilisons dans **"Inverser le *commit*"** pour défaire un *commit* de fusion.

Courriel

Beaucoup de projets Git,y compris Git lui-même,sont entièrement maintenus à travers des listes de diffusion de courrier électronique. Git possède un certain nombre d'outils intégrés qui aident à rendre ce processus plus facile, depuis la génération de patchs que vous pouvez facilement envoyer par courriel jusqu'à l'application de ces patchs depuis une boîte de courrier électronique.

git apply

La commande `git apply` applique un patch créé avec la commande `git diff` ou même la commande GNU diff. C'est similaire à ce que la commande `patch` ferait avec quelques petites différences.

Nous démontrons son utilisation et les circonstances dans lesquelles vous pourriez ainsi faire dans **"Application des patchs à partir de courriel"**.

git am

La commande `git am` est utilisée pour appliquer des patchs depuis une boîte de réception de courrier électronique, en particulier ceux qui sont dans le format mbox. C'est utile pour recevoir des patchs par courriel et les appliquer à votre projet facilement.

Nous avons couvert l'utilisation et le flux de travail autour de `git am` dans **"Application d'un patch avec `am`"**, y compris en utilisant les options `--resolved`, `-i` et `-3`.

Il y a aussi un certain nombre de crochets dont vous pouvez vous servir pour vous aider dans le flux de travail autour de `git am` et ils sont tous couverts dans **"Crochets de gestion courriel"**.

Nous l'utilisons aussi pour appliquer les modifications d'un patch au format « GitHub Pull Request » dans **"Notifications par courriel"**.

git format-patch

La commande `git format-patch` est utilisée pour générer une série de patchs au format mbox que vous pouvez envoyer à une liste de diffusion proprement formattée.

Nous explorons un exemple de contribution à un projet en utilisant l'outil `git format-patch` dans **"Projet public via courriel"**.

git imap-send

La commande `git imap-send` téléverse une boîte mail générée avec `git format-patch` dans un dossier « brouillons » IMAP.

Nous explorons un exemple de contribution à un projet par envoi de patchs avec l'outil `git imap-send` dans **"Projet public via courriel"**.

git send-email

La commande `git send-email` est utilisée pour envoyer des patchs générés avec `git format-patch` par courriel.

Nous explorons un exemple de contribution à un projet en envoyant des patchs avec l'outil `git send-email` dans **"Projet public via courriel"**.

git request-pull

La commande `git request-pull` est simplement utilisée pour générer un exemple de corps de message à envoyer par courriel à quelqu'un. Si vous avez une branche sur un serveur public et que vous voulez faire savoir à quelqu'un comment intégrer ces modifications sans envoyer les patchs par courrier électronique, vous pouvez lancer cette commande et envoyer la sortie à la personne dont vous voulez qu'elle tire les modifications.

Nous démontrons comment utiliser `git request-pull` pour générer un message de tirage dans **"Projet public dupliqué"**.

Systèmes externes

Git est fourni avec quelques commandes pour s'intégrer avec d'autres systèmes de contrôle de version.

git svn

La commande `git svn` est utilisée pour communiquer avec le système de contrôle de version Subversion en tant que client. Cela signifie que vous pouvez vous servir de Git pour extraire depuis et envoyer des *commits* à un serveur Subversion.

Cette commande est couverte en profondeur dans **“Git et Subversion”**.

git fast-import

Pour les autres systèmes de contrôle de version ou pour importer depuis presque n’importe quel format, vous pouvez utiliser `git fast-import` pour associer rapidement l’autre format à quelque chose que Git peut facilement enregistrer.

Cette commande est couverte en profondeur dans **“Un importateur personnalisé”**.

Administration

Si vous administrez un dépôt Git ou si vous avez besoin de corriger quelque chose de façon globale, Git fournit un certain nombre de commandes administratives pour vous y aider.

git gc

La commande `git gc` lance le « ramasse-miette » sur votre dépôt, en supprimant les fichiers superflus de votre base de données et en empaquetant les fichiers restants dans un format plus efficace.

Cette commande tourne normalement en arrière-plan pour vous, même si vous pouvez la lancer manuellement si vous le souhaitez. Nous parcourons quelques exemples dans **“Maintenance”**.

git fsck

La commande `git fsck` est utilisée pour vérifier les problèmes ou les incohérences de la base de données interne.

Nous l’utilisons rapidement une seule fois dans **“Récupération de données”** pour chercher des objets ballants.

git reflog

La commande `git reflog` explore un journal de là où toutes vos branches sont passées pendant que vous travailliez pour trouver des *commits* que vous pourriez avoir perdus en ré-écrivant des historiques.

Nous couvrons cette commande principalement dans **"Raccourcis RefLog"**, où nous montrons un usage normal et comment utiliser `git log -g` pour visualiser la même information avec la sortie de `git log`.

Nous explorons aussi un exemple pratique de récupération d'une telle branche perdue dans **"Récupération de données"**.

git filter-branch

La commande `git filter-branch` est utilisée pour réécrire un tas de *commits* selon des motifs particuliers, comme supprimer un fichier partout ou filtrer le dépôt entier sur un seul sous-dossier pour l'extraction d'un projet.

Dans **"Supprimer un fichier de chaque *commit*"**, nous expliquons la commande et explorons différentes options telles que `--commit-filter`, `--subdirectory-filter` et `--tree-filter`.

Dans **"Git-p4"** et **"TFS"**, nous l'utilisons pour arranger des dépôts externes importés.

Commandes de plomberie

Il y a un certain nombre de commandes de plomberie de bas niveau que nous avons rencontrées dans le livre.

La première que nous avons rencontrée est `ls-remote` dans **"Références aux requêtes de tirage"** que nous utilisons pour regarder les références brutes sur le serveur.

Nous utilisons `ls-files` dans **"Re-fusion manuelle d'un fichier"**, **"Rerere"** et **"L'index"** pour jeter un coup d'œil plus cru sur ce à quoi ressemble votre index.

Nous mentionnons aussi `rev-parse` dans **"Références de branches"** pour prendre n'importe quelle chaîne de caractères et la transformer en un objet SHA.

Cependant, la plupart des commandes de plomberie de bas niveau que nous couvrons se trouvent dans **Chapter 10**, qui est plus ou moins ce sur quoi le chapitre se focalise. Nous avons évité de les utiliser tout au long de la majeure partie du reste du livre.

Index

Symbols

$EDITOR, **392**
$VISUAL
    voir $EDITOR, **392**
(contribuer, équipe privée gérée, **165**
.gitignore, **394**
.NET, **567**
@{upstream}, **107**
@{u}, **107**

A

achèvement tab
    bash, **555**
    powershell, **558**
    zsh, **556**
aliases, **72**
Apache, **139**
Apple, **567**
archivage, **411**
attributs, **404**
autocorrect, **395**

B

bash, **555**
BitKeeper, **22**
bitnami, **143**
branches, **75**
    basculer, **79**
    création, **78**
    différences, **185**
    distant, **99**
    distantes, **184**
    gestion, **93**
    long cours, **95**
    processus standard, **83**
    suivi, **106**
    suppression distante, **109**
    thématique, **96**
    thématiques, **180**
    upstream, **106**

C

C, **562**
C#, **567**
clés SSH, **133**
    avec GitHub, **201**
Cocoa, **567**
commandes git
    add, **38**, **39**
    am, **181**
    apply, **180**
    archive, **196**
    branche, **78**, **93**
    checkout, **79**
    cherry-pick, **192**
    clone
        bare, **130**
    commit, **46**, **76**
    config, **47**, **72**, **177**, **391**
    credential, **384**
    daemon, **137**
    describe, **195**
    diff, **43**
        check, **156**
    fast-import, **487**
    filter-branch, **485**
    format-patch, **176**
    gitk, **546**
    gui, **546**
    help, **137**
    http-backend, **138**
    init, **39**
        bare, **131**, **135**
    instaweb, **141**
    merge
        squash, **175**

mergetool, **92**
p4, **459**, **484**
pull, **65**
push, **65**, **71**, **104**
rebase, **110**
remote, **62**, **63**, **65**, **67**
request-pull, **172**
rerere, **193**
shortlog, **196**
show, **69**
show-ref, **433**
status, **37**
svn, **429**
tag, **67**, **68**, **70**
contribuer, **155**
grand projet public, **175**
petit projet public, **171**
petite équipe privée, **158**
couleur, **395**
courriel, **177**
courriel, appliquer des patchs depuis), **180**
credential caching, **29**
CRLF, **29**
crlf, **401**
crochets, **414**
post-update, **127**

D

difftool, **397**
Duplication, **205**

E

Eclipse, **554**
éditeur
changer par défaut, **47**
espaces blancs, **400**
étiquettes, **194**
annotées, **68**
légères, **69**
signer, **194**
excludes, **394**
exclusions, **498**
exemple de politique, **418**
expansion des mots-clés, **408**

F

fichiers
déplacer, **50**
effacer, **48**
fichiers binaires, **404**
fins de ligne, **401**
flux de travail, **151**
centralisés, **151**
dictateur et lieutenants, **154**
fusion, **187**
fusion (grande), **190**
gestionnaire d'intégration, **153**
rebaser et picorer, **191**
fork, **153**
fusion
stratégies, **413**
fusionner
conflits, **90**
fussionner
vs. rebaser, **120**

G

git commands
clone, **36**
config, **32**
fetch-pack, **526**
help, **33**
init, **35**
receive-pack, **524**
send-pack, **524**
upload-pack, **526**
Git comme client, **429**
git distribué, **151**
git-svn, **429**
git-tf, **468**
git-tfs, **468**
GitHub, **199**
API, **251**
comptes utilisateur, **199**
processus, **206**
regroupement, **242**
requêtes de tirage, **209**
GitHub pour Mac, **548**
GitHub pour Windows, **548**
gitk, **546**
GitLab, **143**
GitWeb, **140**
GPG, **394**
GUIs, **545**

H

historique

filtrage, **58**
format, **54**

I

identifiants, **384**
ignorer des fichiers, **42**
Importation
depuis les autres, **487**
depuis Mercurial, **481**
depuis Perforce, **483**
depuis Subversion, **478**
depuis TFS, **486**
intégrer les contributions, **187**
Interopération avec d'autres VCS
Mercurial, **442**
Perforce, **451**
Subversion, **429**
TFS, **467**
invites
bash, **555**
powershell, **558**
zsh, **556**
IRC, **33**

J

java, **568**
jgit, **568**

L

libgit2, **562**
Linux, **22**
installing, **28**

M

Mac
installing, **28**
maintenance d'un projet, **179**
master, **77**
Mercurial, **442**, **481**
mergetool, **397**
Migration vers Git, **478**
modèles de message de validation, **393**
Mono, **567**

N

nom de révision, **195**

O

Objective-C, **567**
origin, **99**
Outils graphiques, **545**

P

pager, **394**
Perforce, **22**, **451**, **483**
Git Fusion, **451**
posh-git, **558**
pousser, **104**
Powershell, **29**
powershell, **558**
protocoles
git, **129**
HTTP idiot, **126**
HTTP intelligent, **126**
local, **124**
SSH, **128**
publier, **196**
Python, **567**

R

rebaser, **109**
dangers, **115**
vs. fusionner, **120**
références
distant, **99**
rerere, **193**
Ruby, **563**

S

servir des dépôts, **123**
GitLab, **143**
GitWeb, **140**
HTTP, **138**
protocole git, **137**
SSH, **132**
SHA-1, **25**
Subversion, **22**, **429**, **478**

T

tags, **67**
TFS, **467**, **486**
TFVC (see TFS)
tirer, **108**

V

version control, **17**
  centralized, **19**
  distributed, **20**
  local, **18**
Visual Studio, **552**

W

Windows
  installing, **29**

X

Xcode, **28**

Z

zone d'index
  passer, **48**
zsh, **556**