THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Ця робота розповсюджується під ліцензією Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License. Для перегляду копії цієї ліцензії, відвідайте ***http://creativecommons.org/licenses/by-nc-sa/3.0/*** або відправте листа до Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Передмова від Скота Чакона

Ласкаво просимо до другого видання Pro Git. Перше видання було опубліковано близько чотирьох років тому. Відтоді багато чого змінилось, але важливі речі залишились незмінними. Тим часом, поки більшість команд та понять залишаються актуальними сьогодні, оскільки команда ядра Git фантастична в плані збереження зворотньої сумісності, стались значні доповнення та зміни в навколишньому оточенні Git. Друге видання цієї книги призначено описати ці зміни та оновити книжку, щоб вона могла бути більш корисною для нового користувача.

Коли я писав перше видання, Git ще був складним для використання і ледь прийнятним інструментом для хакерів. Він починав набирати обертів у деяких спільнотах, але ще навіть близько не досяг тієї всюдисущості яку має сьогодні. Відтоді, майже кожна спільнота відкритого коду прийняла його. Git зробив неймовірний прогрес на Windows, в різноманітті графічних користувацьких інтерфейсів до нього для всіх платформ, в підтримці IDE і у використанні для бізнесу. Pro Git чотири роки тому не знав ні про що з цього. Одна з головних цілей нового видання це додати в нього опис усіх цих нових меж в Git спільноті.

Спільнота відкритого коду використовуючи Git також отримала піднесення. Коли я вперше сів писати книгу близько п'яти років тому (видання першої версії потребувало певного часу), я лише розпочав працювати в дуже маловідомій компанії, де розробляв сайт для розгортання (hosting) Git під назвою GitHub. На час публікації було близько декількох тисяч людей, що використовували сайт, і лише четверо нас, хто підтримував його. Під час написання цього вступу, GitHub анонсував наш 10 мільйонний проект, та близько 5 мільйонів зареєстрованих облікових записів і коло 230 працівників. Подобається вам це чи ні, GitHub сильно змінив ділянки спільноти відкритого коду, що навряд чи було можливо, коли я сів писати перше видання.

Я написав невеличку секцію в оригінальній версії Pro Git про GitHub, як приклад сайту розгортання Git, з яким мені ніколи не було комфортно. Мені не дуже подобалось, що я пишу книжку, яка на мою

думку є по суті відкритим ресурсом, і водночас розповідаю в ній про свою компанію. Хоч я й досі не люблю цей конфлікт інтересів, важливість GitHub в Git спільноті незаперечна. Замість прикладу Git хостингу, я вирішив перетворити цю частину книги в більш глибокий опис того чим є GitHub і як ефективно використовувати його. Якщо ви збираєтесь вивчати як використовувати Git, то знання того як користуватись GitHub допоможе вам стати частиною величезної спільноти, котра дуже цінна незалежно від того який Git хост ви вирішите використовувати для власного коду.

Інша велика зміна з моменту останнього видання, це розробка та розширення протоколу HTTP для мережевих операцій Git. Більшість прикладів у книзі були змінені з SSH на HTTP, бо він набагато простіший.

Це вражаюче — спостерігати за тим, як протягом останніх років Git виріс з порівняно маловідомої системи контролю версій в домінуючу, комерційну систему з відкритим вихідним кодом. Я щасливий що Pro Git добре спрацювала, і також здатна бути однією з кількох технічних книжок в магазині, яка одночасно успішна та з повністю відкритим кодом.

Я сподіваюсь вам сподобається оновлена версія Pro Git;

Передмова від Бена Страуба

Перша версія цієї книги стала тим, що втягло мене в Git. Це був мій вступ в стиль створення програмного забезпечення, яке відчувається більш природнім ніж будь-що бачене дотоді. Я був розробником декілька років до того, але це був правильний поворот, який направив мене набагато цікавішим шляхом, ніж той, на якому я був.

Тепер, по кількох роках, я долучився до головної реалізації Git, я працював на найбільшу компанію розгортання Git, і я подорожував світом навчаючи людей Git. Коли Скот запитав мене чи був би я зацікавлений в роботі над другим виданням, я навіть не роздумував.

Працювати над цією книгою було великим задоволенням та привілеєм. Я надіюсь вона допоможе тобі так, як допомогла мені.

Присвячується

Моїй дружині, Беккі, без кого ця пригода ніколи б не розпочалась. — Бен

Ця версія присвячується моїм дівчатам. Моїй дружині Джесіці яка підтримувала мене всі ці роки і моїй доньці Жозефіні, котра буде підтримувати мене, коли я буду занадто старий, щоб усвідомлювати те, що відбувається. — Скотт

Автори

Оскільки це книга з відкритим кодом, накопичилась певна кількість помилок та змін, внесених за останні роки. Нижче наведено список всіх учасників, що внесли свої зміни в англомовну версію книги Pro Git, як в проект з відкритим кодом. Дякуємо кожному за допомогу в створенні кращої книги для кожного.

```
  2  Aaron Schumacher
  4  Aggelos Orfanakos
  4  Alec Clews
  1  Alex Moundalexis
  2  Alexander Harkness
  1  Alexander Kahn
  1  Andrew McCarthy
  1  AntonioK
  1  Benjamin Bergman
  1  Brennon Bortz
  2  Brian P O'Rourke
  1  Bryan Goines
  1  Cameron Wright
  1  Chris Down
  1  Christian Kluge
  1  Christoph Korn
  2  Ciro Santilli
  2  Cor
  1  Dan Croak
  1  Dan Johnson
  1  Daniel Kay
  2  Daniel Rosen
  1  DanielWeber
  1  Dave Dash
 10  Davide Fiorentino lo Regio
  2  Dilip M
  1  Dimitar Bonev
  1  Emmanuel Trillaud
  1  Eric-Paul Lecluse
  1  Eugene Serkin
  1  Fernando Dobladez
  2  Gordon McCreight
```

```
  1  Helmut K. C. Tessarek
 31  Igor Murzov
  1  Ilya Kuznetsov
  1  Jason St. John
  1  Jay Taggart
  1  Jean Jordaan
 51  Jean-Noël Avila
  1  Jean-Noël Rouvignac
  1  Jed Hartman
  1  Jeffrey Forman
  1  John DeStefano
  1  Junior
  1  Kieran Spear
  1  Larry Shatzer, Jr
  1  Linquize
  1  Markus
  7  Matt Deacalion Stevens
  1  Matthew McCullough
  1  Matthieu Moy
  1  Max F. Albrecht
  1  Michael Schneider
  8  Mike D. Smith
  1  Mike Limansky
  1  Olivier Trichet
  1  Ondrej Novy
  6  Ori Avtalion
  1  Paul Baumgart
  1  Peter Vojtek
  1  Philipp Kempgen
  2  Philippe Lhoste
  1  PowerKiKi
  1  Radek Simko
  1  Rasmus Abrahamsen
  1  Reinhard Holler
  1  Ross Light
  1  Ryuichi Okumura
  1  Sebastian Wiesinger
  1  Severyn Kozak
  1  Shane
  2  Shannen
  8  Sitaram Chamarty
  5  Soon Van
  4  Sven Axelsson
  2  Tim Court
  1  Tuomas Suutari
  1  Vlad Gorodetsky
  3  W. Trevor King
  1  Wyatt Carss
  1  Włodzimierz Gajda
  1  Xue Fuqiao
```

```
1  Yue Lin Ho
2  adelcambre
1  anaran
1  bdukes
1  burningTyger
1  cor
1  iosias
7  nicesw123
1  onovy
2  pcasaretto
1  sampablokuper
```

Вступ

Ви збираєтеся витратити кілька годин свого життя на читання про Git. Отже, викроїмо хвильку, щоб пояснити, що для вас тут приготовлено. Нижче подано поверхневий огляд всіх десяти розділів та трьох додатків цієї книги.

У **Розділі 1**, ми збираємося оглянути системи контролю версій (СКВ) та нетехнічні основи Git, просто що таке Git, чому він з'явився (на тлі, вже й так переповненому іншими СКВ), що вирізняє Git та чому так багато людей ним користуються. Потім розкажемо як завантажити Git та налаштувати вперше, якщо ви цього ще не зробили.

У **Розділі 2**, пройдемося по основах користування Git — це вам згодиться у 80% випадків з найпоширеніших задач. Після прочитання цього розділу, ви зможете клонувати репозиторій, бачити що відбувалося в історії проекту, змінювати файли, та долучати свої зміни. Якщо після цього книга спопелиться в цей момент, ви вже будете в змозі з неабиякою користю вживати Git, доки не знайдете нову копію книги.

Розділ 3 про модель гілкоутворення в Git, яку часто називають вбивчою особливістю Git. Тут ви дізнаєтеся що насправді вирізняє Git від йому подібних. По закінченню, у вас може виникнути бажання побути наодинці та поміркувати як ви жили до того, як ви познайомилися з гілками Git.

Розділ 4 описує Git на сервері. Це для тих з вас, хто хоче налаштувати Git всередині організації або свій персональний сервер. Також розглянемо варіанти розміщення (hosting), якщо ви надаєте перевагу, щоб хтось для вас займався розгортанням.

Розділ 5 детально пройдеться по різних розподілених процесах роботи та як їх досягнути за допомогою Git. Як розквитаєтеся з цим розділом, то зможете, з вправністю експерта, працювати з кількома сховищами, користуватися Git-ом через електронну пошту та хвацько жонглювати низкою віддалених гілок та латками зі змінами.

Розділ 6 покриває деталі розміщення в GitHub та його інструментарій. Ми розглянемо реєстрацію облікового запису, управління ним, створення та використання Git репозиторіїв, типові

схеми долучання до чиїхось проектів, та приймання наробків у свої, програмний інтерфейс GitHub та багато дрібних порад, які мали б спростити ваші життя.

Розділ 7 про складніші команди Git. Тут дізнаєтесь про такі теми як опановування страшної команди *reset*, залучення двійкового пошуку для виявлення вад, зміну історії, докладно про вибір ревізій, та багато іншого. Цей розділ доповнить ваші знання про Git, щоб ви були дійсно майстром Git.

Розділ 8 про підлаштування Git середовища. Включно з встановленням гаків (hook scripts), які змушують, чи спонукають, до використання своїх правил роботи з Git, а також використання конфігураційних налаштувань середовища для того, щоб робота з Git була такою, як вам це пасує. Також покриємо побудову власних скриптів, щоб запровадити власні нетипові вимоги-обмеження до процесу створення комітів.

Розділ 9 розповідає як мати справу з Git разом з іншими СКВ. Зокрема, про використання Git у світі Subversion (SVN) та конвертацію з інших СКВ до Git. Низка організацій досі використовує SVN та не збирається це змінювати, але ви вже вивчили про надзвичайну силу Git і цей розділ показує як з цим справлятися, якщо вам все-таки потрібно використовувати SVN сервер. Ми також розкажемо як імпортувати проекти з кількох різних систем, якщо раптом ви переконали всіх зробити такий крок.

Розділ 10 копається в непроглядних, але красивих нетрях внутрішнього світу Git. Тепер, ви вже знаєте все про Git та можете ним володіти з граціозною силою, тому, час перейти на дискусію про те, як Git зберігає свої об*єкти, його об*єктну модель, детальніше про пакфайли (packfiles), протоколи сервера, та інше. Ми посилатимемося до цього розділу впродовж цілої книги для того, щоб ви, при бажанні, могли заглибитися в ту чи іншу тему; але якщо ви, як і ми, полюбляєте занурюватися в технічні деталі, можете розпочати книгу з 10 розділу. Вам вирішувати.

У **Додатку A** ми розглянемо приклади використання Git в кількох специфічних середовищах. Ми охоплюємо ряд графічних інтерфейсів та розробницьких середовищ, які, можливо, є у вас в наявності та якими ви можете захотіти користуватися. Якщо вас цікавить огляд користування Git у вашій оболонці розробки, Visual Studio чи Eclipse, — загляньте сюди.

У **Додатку B** ми оглядаємо скриптування та розширення Git за допомогою таких бібліотек, як libgit2 та JGit. Якщо ви зацікавлені в написанні власних складних та швидких допоміжних інструментів та

вам потрібен низькорівневий доступ до Git, ось де ви зможете ознайомитися як ця кухня виглядає.

Накінець, у **Додатку C** ми проходимося окремо по кожній з основних команд Git та оглядаємо де та що ми з ними робили в книзі. Користуйтеся цим для того, щоб віднайти де в книзі зустрічалася та чи інша команда.

Розпочнімо.

Table of Contents

Передмова від Скота Чакона **iii**

Передмова від Бена Страуба **v**

Присвячується **vii**

Автори **ix**

Вступ **xiii**

РОЗДІЛ 1: Вступ **29**

Про систему контролю версій 29

Локальні системи контролю версій 29

Централізовані системи контролю версій 31

Децентралізовані системи контролю версій 32

Коротка історія Git 34

Основи Git 34

Знімки, а не відмінності 35

Майже кожна операція локальна 36

Git цілісний 37

Git, зазвичай, тільки додає дані 37

Три стани 38

Командний рядок 39

Інсталяція Git 40

Інсталяція на Linux 40

Інсталяція на Mac 40
Інсталяція на Windows 41
Встановлення з джерельного коду 42
Початкове налаштування Git 43
Ім’я користувача 43
Редактор 44
Перевірка налаштувань 44
Отримання допомоги 45
Підсумок 46

РОЗДІЛ 2: Основи **Git** **47**
Створення Git-репозіторія 47
Ініціалізація репозіторія в існуючому каталозі 47
Клонування існуючого репозиторію 48
Запис змін до репозиторію 49
Перевірка статусу ваших файлів 50
Контролювання нових файлів 51
Індексування змінених файлів 52
Короткий статус 53
Ігнорування файлів 54
Перегляд ваших доданих та недоданих змін 56
Збереження ваших змін у комітах 59
Обходимо індекс 60
Removing Files 61
Пересування файлів 63
Перегляд історії комітів 64
Обмеження виводу журналу 69
Скасування речей 71
Вилучання файла з індексу 72
Скасування змін у зміненому файлі 73
Взаємодія з віддаленими сховищами 74

Дивимось на ваші сховища 74
Додавання віддалених сховищ 76
Отримання (fetching) та затягування (pulling) з ваших віддалених сховищ 77
Надсилання змін до ваших віддалених сховищ 77
Оглядання віддаленого сховища 78
Видалення та перейменування віддалених сховищ 79
Теґування 80
Показати ваші теґи 80
Створення теґів 81
Анотовані теґи 81
Легкі теґи 82
Створення теґів пізніше 82
Розповсюдження теґів 83
Переключення до теґів 84
Псевдоніми Git 85
Підсумок 86

РОЗДІЛ 3: Галуження в **git** **87**
Гілки у кількох словах 87
Створення нової гілки 90
Переключення гілок 91
Основи галуження та зливання 95
Основи галуження 95
Основи зливання 100
Основи конфліктів зливання 102
Управління гілками 105
Процеси роботи з гілками 106
Довго-триваючі гілки 106
Тематичні гілки 108
Віддалені гілки 110

Надсилання 116
Відслідковувані гілки 118
Витягування змін 120
Видаляння віддалених гілок 120
Перебазовування 121
Просте перебазовування 121
Цікавіше перебазовування 124
Небезпеки перебазовування 126
Перебазовуйся коли перебазовуєшся 129
Перебазовування чи зливання 131
Підсумок 131

РОЗДІЛ 4: Git на сервері **133**
Протоколи 134
Локальний протокол 134
Протоколи HTTP 136
Протокол SSH 138
Протокол Git 139
Отримання Git на сервері 140
Розміщення очищеного сховища на сервер 141
Маленькі проекти 142
Генерація вашого публічного ключа SSH 143
Налаштування Серверу 144
Демон Git 147
Розумний HTTP 149
GitWeb 151
GitLab 154
Інсталяція 154
Адміністрування 154
Базове користування 157
Співпраця 158

Варіанти стороннього хостингу 159
Підсумок 159

РОЗДІЛ 5: Розподілений **Git** **161**
Розподілені процеси роботи 161
Централізований процес роботи 162
Процес роботи з менеджером інтеграції 163
Процес роботи з диктатором та лейтенантами 164
Підсумок щодо процесів роботи 165
Contributing to a Project 165
Commit Guidelines 166
Private Small Team 168
Private Managed Team 175
Forked Public Project 181
Public Project over E-Mail 185
Summary 189
Maintaining a Project 189
Working in Topic Branches 189
Applying Patches from E-mail 190
Checking Out Remote Branches 193
Determining What Is Introduced 194
Integrating Contributed Work 196
Tagging Your Releases 202
Generating a Build Number 204
Preparing a Release 204
The Shortlog 205
Підсумок 205

РОЗДІЛ 6: **GitHub** **207**
Створення та налаштування облікового запису 207
SSH доступ 208

Ваш аватар 210
Ваші поштові адреси 211
Двох-крокова авторизація 212
Як зробити внесок до проекту 213
Робимо форк проекту 213
Потік роботи GitHub 214
Глибше про Запити на Пул 222
Markdown 228
Супроводжування проекту 233
Створення нового сховища 233
Додавання співпрацівників 236
Керування Запитами на Пул (Pull Requests) 237
Згадки та повідомлення 244
Особливі файли 247
README (Прочитай мене) 248
CONTRIBUTING (Як зробити внесок) 248
Адміністрування Проекту 249
Керування організацією 250
Основи організацій 250
Команди 252
Журнал подій 253
Скриптування GitHub 254
Хуки 255
GitHub API 259
Базове використання 260
Коментування завдання 261
Зміна статусу Запиту на Пул 263
Octokit 265
Підсумок 265

РОЗДІЛ 7: Інструменти **Git** **267**

Вибір ревізій 267
Окремі ревізії 267
Короткий SHA-1 267
Гілкові посилання 269
Скорочення reflog (журнал посилань) 270
Батьківські Посилання 271
Інтервали комітів 273
Інтерактивне індексування 276
Індексування та деіндексування файлів 277
Індексування латок (patch) 279
Хования та чищення 281
Хования ваших змін 281
Винахідливе хования 284
Створення гілки з хования 285
Очищення робочої директорії 286
Підписання праці 288
Знайомство з GPG 288
Підписання теґів 288
Перевірка теґів 289
Підписання комітів 290
Усі мусять підписувати 292
Пошук 292
Git Grep 292
Пошук у журналі Git 294
Переписування історії 296
Зміна останнього коміту 296
Зміна декількох повідомлень комітів 297
Зміна послідовності комітів 300
Зварювання комітів 300
Розщеплення коміту 301
Ядерний варіант: filter-branch 302

Усвідомлення скидання (reset) 304
Три дерева 305
Робочий процес 307
Роль скидання (reset) 313
Скидання зі шляхом 318
Зварювання (squashing) 321
Отримання (checkout) 324
Підсумок 326
Складне злиття 327
Конфлікти злиття 328
Скасування зливань 339
Інші типи зливань 343
Rerere 347
Зневадження з Git 354
Анотація файла 354
Двійковий пошук 356
Підмодулі 358
Основи підмодулів 359
Клонування проекту з підмодулями 361
Робота над проектом з підмодулями 363
Поради щодо підмодулів 374
Проблеми з підмодулями 376
Пакування 379
Заміна 383
Збереження посвідчення (credential) 391
Під капотом 393
Спеціальний кеш посвідчень 395
Підсумок 397

РОЗДІЛ 8: Налаштування **Git** **399**
Конфігурація Git 399

Базові налаштування клієнта 400
Кольори у Git 404
External Merge and Diff Tools 405
Formatting and Whitespace 408
Server Configuration 411
Git Attributes 412
Binary Files 412
Keyword Expansion 415
Exporting Your Repository 418
Merge Strategies 420
Git Hooks 421
Installing a Hook 421
Client-Side Hooks 421
Server-Side Hooks 423
Приклад політики користування виконуваної Git-ом 424
Гак серверної частини 425
Гаки клієнтської частини 431
Підсумок 435

РОЗДІЛ 9: Git та інші системи контролю версій 437
Git як клієнт 437
Git and Subversion 437
Git і Mercurial 449
Git і Perforce 458
Git and TFS 476
Міграція на Git 486
Subversion 486
Mercurial 488
Perforce 491
TFS 493
A Custom Importer 494

Підсумок 502

РОЗДІЛ 10: Git Internals 503

Сантехніка та порцеляна 503

Git Objects 505

Tree Objects 507

Commit Objects 510

Object Storage 513

Git References 515

The HEAD 516

Tags 517

Remotes 519

Packfiles 519

The Refspec 523

Pushing Refspecs 525

Deleting References 525

Transfer Protocols 526

The Dumb Protocol 526

The Smart Protocol 528

Protocols Summary 531

Maintenance and Data Recovery 532

Maintenance 532

Data Recovery 533

Removing Objects 535

Змінні середовища 539

Глобальна поведінка 540

Місцезнаходження сховищ 540

Специфікації файлів 541

Фіксація комітів 542

Мережа 542

Визначення відмінностей та злиття 543

Зневадження 543
Інше 545
Summary 546

Git in Other Environments **547**

Embedding Git in your Applications **563**

Git Commands **575**

Index **593**

1 Вступ

Цей розділ про те, як почати працювати з Git. Спочатку ви опануєте основи систем контролю версій, потім - дізнаєтеся як запустити Git на вашій ОС і, зрештою, як його налаштувати. До кінця розділу ви зрозумієте, що таке Git, для чого він вам, а також підготуєте все необхідне для початку роботи.

Про систему контролю версій

Що таке “система контролю версій”, і чому це важливо? Система контролю версій - це система, що записує зміни у файл або набір файлів протягом деякого часу, так що ви зможете повернутися до певної версії пізнише. Як приклад, в цій книзі, для файлів, що знаходяться під контролем версій, буде використовуватися код програмного забезпечення, хоча насправді ви можете використовувати контроль версій практично для будь-яких типів файлів.

Якщо ви графічний або веб-дизайнер і хочете зберегти кожну версію зображення або макета (швидше за все, захочете), система контролю версій (далі СКВ) якраз те, що потрібно. Вона дозволяє повернути файли до стану, в якому вони були до змін, повернути весь проект до попереднього стану, побачити зміни, побачити, хто останній міняв щось і спровокував проблему, хто вказав на проблему і коли, та багато іншого. Використання СКВ також в цілому означає, що, якщо ви зламали щось або втратили файли, ви просто можете все виправити. Крім того, ви отримаєте все це за дуже невеликі накладні витрати.

Локальні системи контролю версій

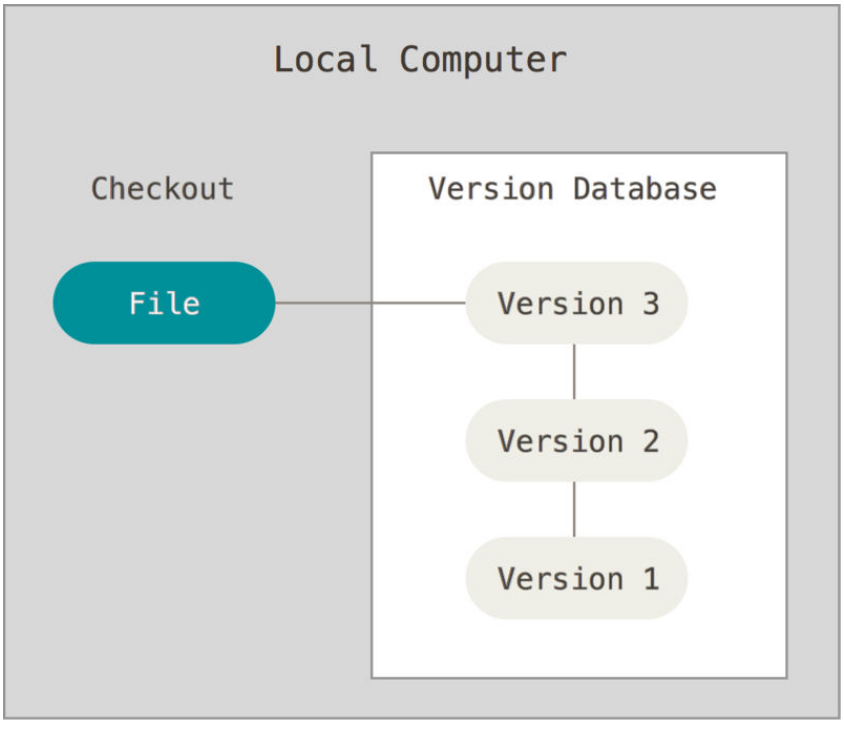
Багато людей в якості одного з методів контролю версій застосовують копіювання файлів в окрему директорію (можливо навіть директорію з

відміткою за часом, якщо вони достатньо розумні). Даний підхід є дуже поширеним завдяки його простоті, проте він, неймовірним чином, схильний до появи помилок. Можна легко забути в якій директорії ви знаходитеся і випадково змінити не той файл або скопіювати не ті файли, які ви хотіли.

Щоб справитися з цією проблемою, програмісти давно розробили локальні СКВ, що мають просту базу даних, яка зберігає всі зміни в файлах під контролем версій.

ЗОБРАЖЕННЯ 1-1

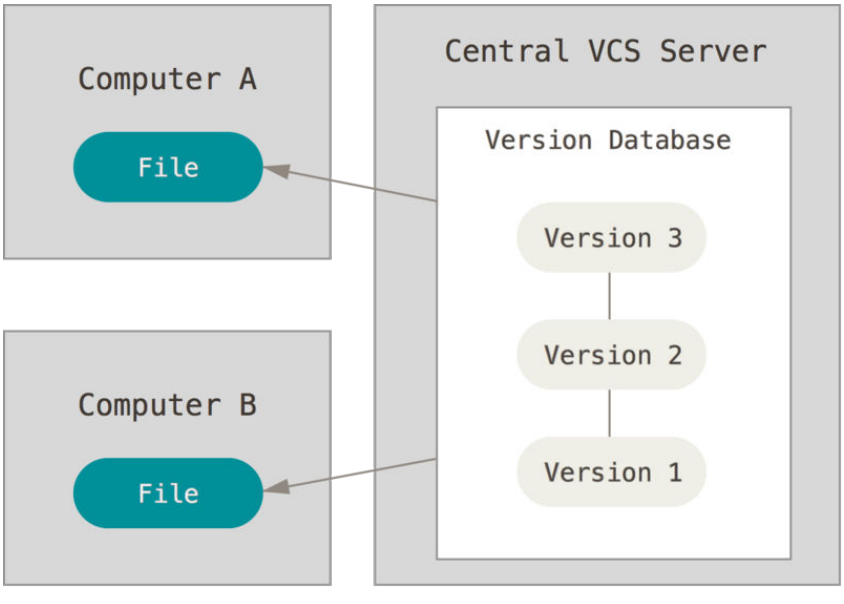
Локальні системи контролю версій.



Одним з найбільш поширених інструментів СКВ була система під назвою RCS, яка досі поширюється з багатьма комп'ютерами сьогодні. Навіть популярна операційна система Mac OS X включає в себе команду rcs при інсталяції Developer Tools. RCS зберігає набори латок (тобто, відмінності між файлами) в спеціальному форматі на диску; він може заново відтворити будь-який файл, як він виглядав, в будь-який момент часу, шляхом додавання всіх латок.

Централізовані системи контролю версій

Наступним важливим питанням, з яким стикаються люди, є необхідність співпрацювати з іншими розробниками. Щоб справитися з цією проблемою, були розроблені централізовані системи контролю версій (ЦСКВ). Такі системи як CVS, Subversion і Perforce, мають єдиний сервер, який містить всі версії файлів, та деяке число клієнтів, які отримують файли з центрального місця. Протягом багатьох років, це було стандартом для систем контролю версій.



ЗОБРАЖЕННЯ 1-2

Централізовані системи контролю версій.

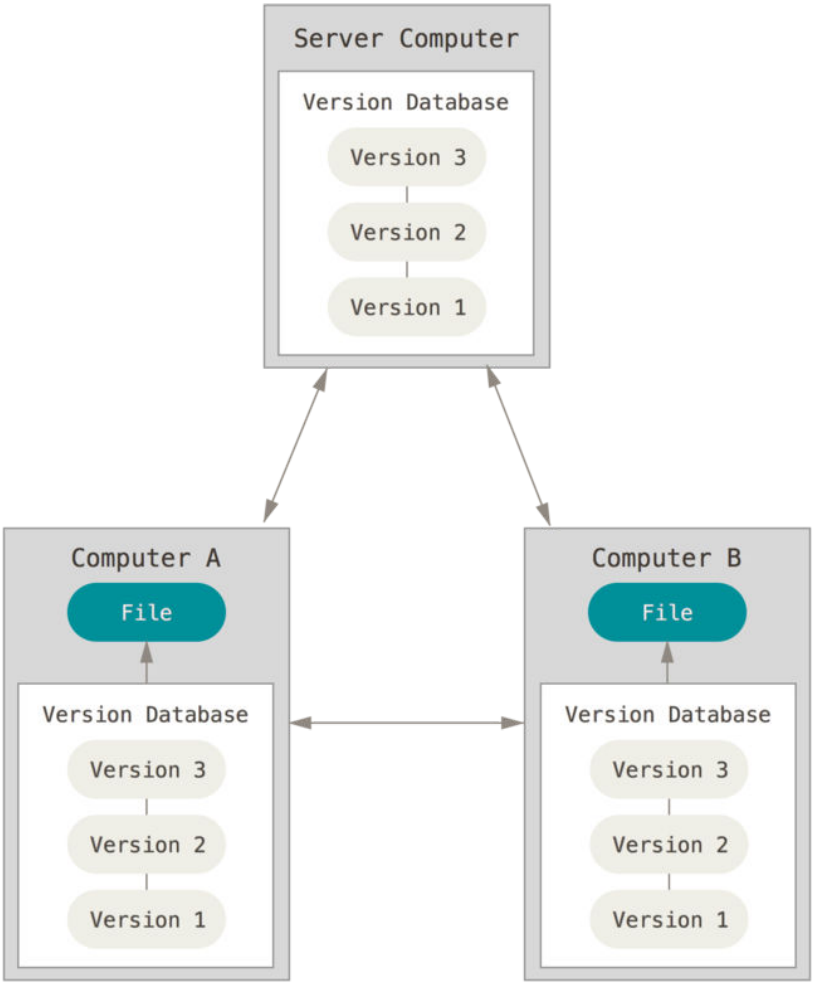
Такий підхід має безліч переваг, особливо над локальними СКВ. Наприклад, кожному учаснику проекту відомо, певною мірою, чим займаються інші. Адміністратори мають повний контроль над тим, хто і що може робити; і це набагато легше адмініструвати з ЦСКВ, ніж мати справу з локальними базами даних для кожного клієнта.

Але цей підхід також має деякі серйозні недоліки. Найбільш очевидним є єдина точка відмови, яким є централізований сервер. Якщо сервер виходить з ладу протягом години, то протягом цієї години ніхто не може співпрацювати або зберігати зміни над якими вони працюють під версійним контролем. Якщо жорсткий диск центральної бази даних на сервері пошкоджено, і своєчасні резервні копії не були зроблені, ви втрачаєте абсолютно все - всю історію

проекту, крім одиночних знімків проекту, що збереглися на локальних машинах людей. Локальні СКВ страждають тією ж проблемою - щоразу, коли вся історія проекту зберігається в одному місці, ви ризикуєте втратити все.

Децентралізовані системи контролю версій

Долучаються до гри децентралізовані системи контролю версій (ДСКВ). В ДСКВ (таких як, Git, Mercurial, Bazaar або Darcs), клієнти не просто отримують останній знімок файлів репозиторію: вони повністю відображають сховище. Таким чином, якщо вмирає який-небудь сервер, через який співпрацюють розробники, будь-який з клієнтських репозиторіїв може бути скопійований назад до серверу, щоб відновити його. Кожна копія дійсно є повною резервної копією всіх даних.



ЗОБРАЖЕННЯ 1-3

Децентралізовані системи контролю версій.

Більш того, багато з цих систем дуже добре взаємодіють з декількома віддаленими репозиторіями, так що ви можете співпрацювати з різними групами людей, застосовуючи різні підходи в межах одного проекту одночасно. Це дозволяє налаштувати декілька типів робочих процесів, таких як ієрархічні моделі, які неможливі в централізованих системах.

Коротка історія Git

Як і в багатьох великих речах у житті, Git почався з трохи творчого руйнування та палких суперечок.

Ядро Linux — це проект досить великого обсягу з відкритим програмним кодом. Більшу частину часу підтримання ядра Linux (1991-2002) виконувалося у вигляді патчів та архівів. У 2002 році проект ядра Linux почав використовувати закриту ДСКВ BitKeeper.

У 2005 році відносини між спільнотою розробників ядра Linux і комерційною компанією, що розробила BitKeeper почали псуватись, і безкоштовне використання продуктом було скасовано. Це підштовхнуло розробників Linux (і зокрема Лінуса Торвальдса, автора Linux) розробити власну систему, ґрунтуючись на деяких з уроків, які вони дізналися під час використання BitKeeper. Деякі з цілей нової системи були:

- швидкість
- проста архітектура
- сильна підтримка для нелінійного розвитку (тисячі паралельних гілок)
- децентралізація
- можливість ефективно управляти великими проектами, такими як ядро Linux (швидкість і розмір даних)

З моменту свого народження в 2005 році, Git розвинувся і дозрів, щоб бути простим у використанні і в той же час зберегти свої первинні властивості. Git неймовірно швидкий, та дуже ефективний для великих проектів, і має неймовірну систему галуження для нелінійного розвитку (див. **Chapter 3**).

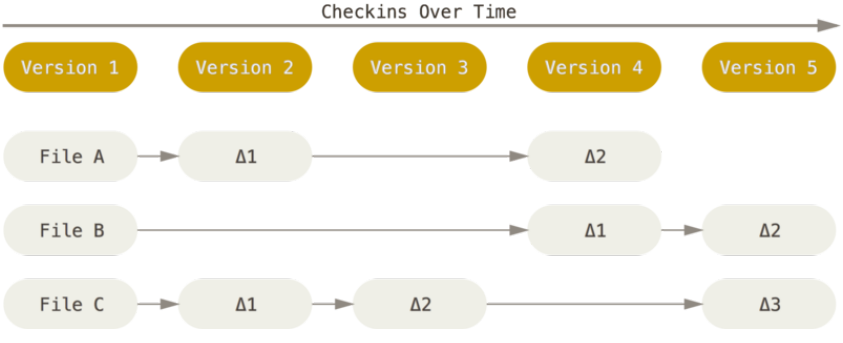
Основи Git

Отже, що таке Git в двох словах? Важливо розуміти цей розділ, тому що, якщо ви розумієте, що таке Git і основи того, як він працює, потім ефективне використання Git, ймовірно, буде набагато простішим. Доки ви вивчаєте Git, спробуйте очистити свій розум від речей, які ви, можливо, знаєте про інші СКВ, такі як Subversion і Perforce; це допоможе вам уникнути деяких проблем при його використанні. Git зберігає і думає про інформацію зовсім по-іншому, у порівнянні з іншими системами, хоча користувальницький інтерфейс дуже схожий, і

розуміння цих відмінностей допоможе вам уникнути плутанини при його використанні.

Знімки, а не відмінності

Головна відмінность від інших систем (таких як Subversion та подібних їй) є те, як Git сприймає дані. Концептуально, більшість СКВ зберігають інформацію як список файлових редагувань. Ці системи (CVS, Subversion, Perforce, Bazaar тощо) розглядають інформацію як список файлів та змін кожного з них протягом деякого часу.



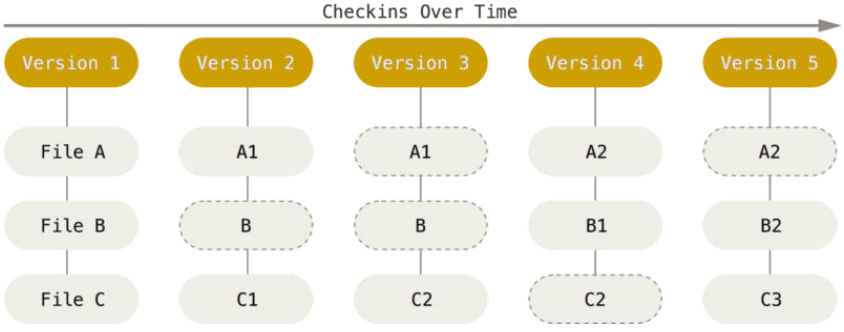
ЗОБРАЖЕННЯ 1-4

Збереження даних, як переліку змін від базової версії кожного файлу.

Git не оброблює та не зберігає свої дані таким чином. Замість цього, Git сприймає свої дані більш як набір знімків мініатюрної файлової системи. Кожен раз, коли ви робите комміт, тобто зберігаєте стан вашого проекту в Git, він запам'ятовує як виглядають всі ваші файли в той момент і зберігає посилання на цей знімок. Для ефективності, якщо файли не змінилися, Git не зберігає файли знову, просто робить посилання на попередній ідентичний файл, котрий вже зберігається. Git вважає свої дані більш як **потік знімків**.

ЗОБРАЖЕННЯ 1-5

Зберігання даних як знімків проекту за хронологією.



Це дуже важлива різниця між Git та майже всіма іншими СКВ. З цієї причини в Git було заново переосмислено майже кожен аспект контролю версій, які інші системи просто копіювали з попереднього покоління. Це зробило Git більш схожим на мініатюрну файлову систему з деякими неймовірно потужними вбудованими інструментами на додаток, а не просто СКВ. Ми познайомимось з деякими перевагами, які ви отримаєте при сприйнятті інформації подібним чином, у **Chapter 3**, де йдеться про гілки.

Майже кожна операція локальна

Більшість операцій у Git потребують лише локальних файлів та ресурсів для здійснення операцій – немає необхідності в інформації з інших комп'ютерів вашої мережі. Якщо ви звикли до ЦСКВ, де більшість операцій обтяжені такими мережевими запитами, то цей аспект може привести вас до думки, що боги швидкості наділили Git неземною силою. Через те, що повна історія проекту знаходиться на вашому локальному диску, більшість операцій здійснюються майже миттєво.

Наприклад, для перегляду історії проекту, Git не має потреби брати її з серверу, він просто зчитує її прямо з локальної бази даних. Це означає, що ви отримуєте історію проекту не встигнувши кліпнути оком. Якщо ви бажаєте переглянути відмінності між поточною версією файлу та його редакцією місячної давності, Git знайде копію збережену місяць тому і проведе локальне обчислення різниці замість того, щоб звертатись за цим до віддаленого серверу чи спочатку робити запит на отримання старішої версії файлу.

Також це означає, що за відсутності мережевого з'єднання ви не будете мати особливих обмежень. Перебуваючи в літаку чи потязі

можна цілком комфортно створювати коміти, доки не відновите з'єднання з мережею для їх відвантаження. Якщо ви прийшли додому та не можете змусити належним чином працювати свій VPN-клієнт, усе одно можна продовжувати роботу. У багатьох інших системах подібні дії або неможливі, або пов'язані з безліччю труднощів. Наприклад, у Perforce, без з'єднання з мережею вам не вдасться зробити багато; у Subversion та CVS ви можете редагувати файли, але не можете створювати коміти з внесених змін (оскільки немає зв'язку з базою даних). На перший погляд такі речі здаються незначними, але ви будете вражені наскільки велике значення вони можуть мати.

Git цілісний

Будь-що в Git, перед збереженням, отримує контрольну суму, за якою потім і можна на нього посилатися. Таким чином, неможливо змінити файл чи директорію так, щоб Git про це не дізнався. Цей функціонал вбудовано в систему на найнижчих рівнях і є невід'ємною частиною її філософії. Ви не можете втратити інформацію при передачі чи отримати пошкоджений файл без відома Git.

Механізм, який використовується для цього контролю, називається хеш SHA-1. Він являє собою 40-символьну послідовність цифр та перших літер латинського алфавіту (a-f) і вираховується на основі вмісту файлу чи структури директорії в Git. SHA-1 хеш виглядає це приблизно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

При роботі з Git ви повсюди зустрічатимете такі хеші, адже Git постійно їх використовує. Фактично, Git зберігає все не за назвою файлу, а саме за значенням хешу його змісту.

Git, зазвичай, тільки додає дані

Коли ви виконуєте певні дії в Git, при цьому, майже завжди відбувається виключно додавання інформації до бази даних Git. Складно змусити систему зробити щось невиправне чи повністю видалити дані будь-яким чином. Як і в будь-якій СКВ, ви можете втратити чи зіпсувати лише не збережені в коміті зміни. Але це майже неможливо, коли вже збережено знімок, особливо, якщо ви регулярно надсилаєте свою базу до іншого сховища.

Це робить використання Git приємним, оскільки ми точно можемо експериментувати без загрози щось зіпсувати. Про те, як Git зберігає інформацію та як можна відновити втрачені дані, що нібито загублені, детальніше розповідається у **“Скасування речей”**.

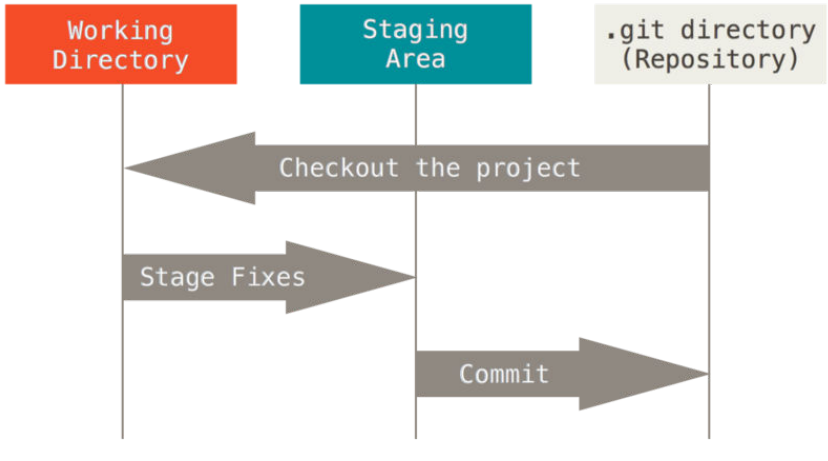
Три стани

А зараз, будьте уважні. Це найважливіша річ, яку потрібно запам’ятати, якщо ви хочете щоб подальше вивчення Git пройшло гладко. Git має три основних стани, в яких можуть перебувати ваші файли: збережений у коміті (commited), змінений (modified) та індексований (staged). Збережений у коміті означає, що дані безпечно збережено в локальній базі даних. Змінений означає, що у файл внесено редагування, які ще не збережено в базі даних. Індексований стан виникає тоді, коли ви позначаєте змінений файл у поточній версії, щоб ці зміни ввійшли до наступного знімку коміту.

Це приводить нас до трьох головних відділів проекту під управлінням Git: директорія Git, робоча директорія та індекс.

ЗОБРАЖЕННЯ 1-6

Робоча директорія, індекс та директорія Git.



У директорії Git система зберігає метадані та базу даних об’єктів вашого проекту. Це найважливіша частина Git, саме вона копіюється при клонуванні сховища з іншого комп’ютеру.

Робоча директорія являє собою одну окрему версію проекту, взяту зі сховища. Ці файли видобуваються з бази даних у теці Git та розміщуються на диску для подальшого використання та редагування.

Індекс — це файл, що зазвичай знаходиться в директорії Git і містить інформацію про те, що буде збережено у наступному коміті. Також цей файл називають “областю додавання” (staging area), проте ми переважно будемо користуватись терміном “індекс”.

Найпростіший процес взаємодії з Git виглядає приблизно так:

1. Ви редагуєте файли у своїй робочій директорії.
2. Надсилаєте файли до індексу, що зберігає їхній поточний стан.
3. Створюєте коміт: знімок з індексу остаточно зберігається в директорії Git.

У випадку, якщо окрема версія файлу вже є в директорії Git, цей файл вважається збереженим у коміті. Якщо він зазнав змін і перебуває в індексі, то він індексований. Якщо ж його стан відрізняється від того, який був у коміті, і файл не знаходиться в індексі, то він називається зміненим. У **Chapter 2** ви дізнаєтесь більше про ці стани, а також про те, як використовувати їхні переваги або взагалі пропускати етап індексу.

Командний рядок

Є багато різних варіантів використання Git. Крім оригінальних клієнтів командного рядка, є безліч клієнтів з графічним користувацьким інтерфейсом з різними можливостями. Для цієї книги ми будемо використовувати Git в командному рядку. З одного боку, командний рядок — єдине місце, де можна виконувати **всі** команди Git - більшість графічних інтерфейсів для простоти реалізують тільки деяку підмножину функціональності Git. Якщо ви знаєте, як виконати щось з командного рядка, ви, ймовірно, також можете з’ясувати, як виконати це і графічному інтерфейсі, у той час як зворотне не завжди вірно. Крім того, в той час, як вибір графічного клієнта справа особистого смаку, інструменти командного рядка мають *усі* користувачі одразу ж після інсталяції.

Таким чином, ми очікуємо, що ви знаєте, як відкрити термінал в Mac або командний рядок, або Powershell в Windows. Якщо ви не розумієте, про що ми тут говоримо, можливо, вам потрібно буде зупинитися та швидко дізнатися це, щоб ви були в змозі розуміти інші приклади і описи в цій книзі.

Інсталяція Git

Перед початком використання Git, ви повинні встановити його на вашому комп'ютері. Навіть якщо він вже встановлений, ймовірно, хороша ідея, щоб оновитися до останньої версії. Ви можете встановити Git з пакету або за допомогою іншого інсталятора або завантажити програмний код і скомпілювати його самостійно.

У цій книзі використовується Git версії **2.0.0**. Хоча більшість команд, які ми використовуємо, повинні працювати навіть в дуже старих версіях Git, деякі з них, можливо, мають діяти трохи по-іншому, якщо ви використовуєте стару версію. Git досить добре зберігає зворотну сумісність, будь-яка версія після 2.0 повинна працювати нормально.

Інсталяція на Linux

Якщо ви бажаєте встановити Git на Linux за допомогою бінарного пакету, ви можете зробити це через основний менеджер управління пакетами, що йде з вашим дистрибутивом. Якщо, наприклад, ви використовуєте Fedora, скористуйтеся yum:

```
$ sudo yum install git
```

Якщо ви використовуєте Debian-подібний дистрибутив, такий як Ubuntu, спробуйте apt-get:

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

Для отримання додаткових можливостей, чи інструкцій з інсталяції для декількох різновидів Unix, скористайтеся сайтом Git ***http://git-scm.com/download/linux***.

Інсталяція на Mac

Є декілька способів установки Git на Mac. Найпростіше, встановити Xcode Command Line Tools. На Mavericks (10.9) або вище, ви можете зробити це просто перший раз виконавши *git* з терміналу. Якщо він ще не встановлен, вам буде запропоновано встановити його.

Якщо ви бажаєте більш свіжу версію, зробіть це за допомогою бінарного інсталятору. OSX Git підтримується та доступний для завантаження на сайті Git ***http://git-scm.com/download/mac***.



ЗОБРАЖЕННЯ 1-7

Git OS X Installer.

Ви також можете встановити його як частину GitHub для Mac. Їхній графічний інструмент Git має можливість інсталяції і інструменти командної строки. Ви можете завантажити цей інструмент з GitHub для Mac сайті ***http://mac.github.com***.

Інсталяція на Windows

Є декілька шляхів встановити Git під Windows. Найофіційніша збірка доступна для завантаження з сайту Git. Просто перейдіть до ***http://git-scm.com/download/win*** і завантаження почнеться автоматично. Зауважте, що цей проект називається *Git for Windows* (також називається msysGit), що є окремим від самого Git. Щоб дізнатись більше, перейдіть до ***http://msysgit.github.io/***.

Ще один простий спосіб встановити Git це встановити GitHub для Windows. Встановка включає версію командного рядка Git та графічну теж. Це також добре працює з Powershell, та налаштовує безпечне кешування даних про користувача та розумні опції CRLF. Ми більше про це все дізнаємось пізніше, зараз просто скажемо, що вам це потрібно. Ви можете завантажити GitHub для Windows за адресою ***http://windows.github.com***.

Встановлення з джерельного коду

Дехто вважає корисним встановлювати Git з джерельного коду, адже так ви отримаєте найновішу версію. Бінарні інсталятори зазвичай трошки відстають, хоч Git і став більш розповсюдженим протягом останніх років, це нічого не змінило.

Якщо ви бажаєте встановити Git з коду, вам необхідно мати наступні бібліотеки, від яких залежить Git: curl, zlib, openssl, expat та libiconv. Якщо ви користуєтесь системою, в якій є утиліта yum (наприклад, Fedora) чи apt-get (будь-яка система на основі Debian), для встановлення цих залежностей у нагоді вам може стати одна з таких команд:

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

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

Щоб бути в змозі додати документацію у різних форматах (doc, html, info), необхідні наступні додаткові залежності:

```
$ sudo yum install asciidoc xmlto docbook2x

$ sudo apt-get install asciidoc xmlto docbook2x
```

Після отримання необхідних бібліотек, можна рухатись далі й стягнути останній затеґований релізний архів з декількох місць. Ви можете отримати його через сайт kernel.org, за адресою ***https://www.kernel.org/pub/software/scm/git***, або з дзеркала на сайті GitHub, за адресою ***https://github.com/git/git/releases***. Взагалі на сторінці GitHub легше зрозуміти, яка версія остання, проте на сторінці kernel.org є підписи релізів, якщо ви бажаєте перевірити завантаження.

Потім компіляція і встановлення:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Коли все вищевказане виконано, ви можете отримати Git за допомогою самого Git задля оновлення:

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

Початкове налаштування Git

Зараз, коли у ви вже маєте Git у системі, можливо, ви захочете зробити декілька речей, щоб налаштувати ваше Git середовище. Це потрібно виконати лише один раз - налаштування залишаються між оновленнями. Ви також можете змінити їх у будь-який час, знову виконавши декілька команд.

До Git входить утиліта що має назву `git config`, яка дозволяє отримати чи встановити параметри, що контролюють усіма аспектами того, як Git виглядає чи працює. Ці параметри можуть бути збережені в трьох різних місцях:

1. Файл `/etc/gitconfig` містить значення для кожного користувача в системі і всіх їх репозиторіїв. Якщо ви передаєте опцію `--system` при виконанні `git config`, параметри читаються та пишуться з цього файлу.
2. Файл `~/.gitconfig` або `~/.config/git/config` - характерні для вашого користувача. Ви можете налаштувати Git читати і писати в цей файл, вказуючи опцію `--global`.
3. Файл `config` у каталозі Git (тобто `.git/config`) у тому репозиторії, який ви використовуєте в даний момент, зберігає налаштування конкретного репозиторію.

Кожен рівень має пріоритет над налаштуваннями в попередньому рівні, тобто параметри в `.git/config` перевизначають параметри в `/etc/gitconfig`.

У системах Windows, Git шукає файл `.gitconfig` в каталозі `$HOME` (`C:\Users\$USER` для більшості користувачів). Він також шукає файл для `/etc/gitconfig`, хоча відносно кореня MSys, котрий знаходиться там, де ви вирішили встановити Git у вашій Windows системі, коли ви запускали інсталяцію.

Ім'я користувача

Перше, що ви повинні зробити, коли ви інсталюєте Git - це встановити ім'я користувача та адресу електронної пошти. Це важливо, тому що кожен коміт в Git використовує цю інформацію, і вона незмінно включена у комміти, які ви робите:

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

Знову ж таки, якщо ви передаєте опцію `--global`, ці налаштування потрібно зробити тільки один раз, тоді Git завжди буде використовувати цю інформацію для всього, що ви робите у цій системі. Якщо ви хочете, перевизначити ім'я або адресу електронної пошти для конкретних проектів, ви можете виконати цю ж команду без опції `--global` в каталозі необхідного проекту.

Багато з графічних інструментів допомагають зробити це при першому запуску.

Редактор

Зараз, коли ваше ім'я вже вказано, ви можете налаштувати текстовий редактор за замовчуванням, який буде використовуватися Git при необхідності ввести повідомлення. Якщо це не налаштовано, Git використовує редактор за замовчуванням вашої системи, як правило це Vim. Якщо ви бажаєте використовувати інший текстовий редактор, наприклад Emacs, необхідно зробити наступне:

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

Vim і Emacs популярні текстові редактори, що часто використовуються розробниками на Unix-похідних системах, таких як Linux і Mac. Якщо ви не знайомі з жодним з цих редакторів або ви користуєтеся Windows, можливо, вам потрібно буде знайти інструкції по налаштуванню вашого улюбленого редактора з Git. Якщо ви не встановите свій редактор, і ви не знайомі з Vim або Emacs, швидше за все, ви потрапите в дійсно скрутне становище, коли вони запустяться.

Перевірка налаштувань

Якщо ви хочете, перевірити налаштування, ви можете використовувати команду `git config --list`, щоб переглянути всі налаштування, котрі Git може знайти:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
```

```
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Ви можете побачити ключі більш ніж один раз, тому що Git читає однакові ключі з різних файлів (наприклад `/etc/gitconfig` або `~/.gitconfig`). У цьому випадку, Git використовує останнє значення для кожного ключа.

Ви також можете перевірити значення конкретного ключа виконавши `git config <key>`:

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

Отримання допомоги

Якщо вам коли-небудь знадобиться допомога при використанні Git, є три способи, щоб отримати допомогу на сторінках довідника (manpage) для будь-якої команди Git:

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

Наприклад, ви можете отримати допомогу у довіднику для команди config, виконавши:

```
$ git help config
```

Ці команди гарні тим, що ви можете отримати доступ до них в будь-якому місці, навіть без доступу до мережі. Якщо сторінок довідника і цієї книги вам недостатньо, ви можете спробувати пошукати допомоги на `#git` або `#github` каналі на сервері IRC Freenode (irc.freenode.net). Ці канали постійно заповнені сотнями людей, які дуже добре інформовані про Git і готові допомогти.

Підсумок

Ви отримали базові знання про те, що таке Git і чим він відрізняється від централізованих систем контролю версій, якими ви, можливо, користувалися раніше. Ви також тепер отримали робочу версію Git у вашій системі, встановлену з вашими персональними налаштуваннями. Настав час, щоб дізнатися деякі основи Git.

Основи Git 2

Якщо ви бажаєте прочитати тільки один розділ, щоб почати працювати з Git, саме цей вам і потрібен. У цьому розділі розглядаються всі основні команди, які потрібні для переважної більшості завдань, що виникають під час роботи з Git. До кінця розділу, ви будете в змозі налаштувати й ініціалізувати репозиторій, починати і зупиняти відстеження файлів, а також готувати і вносити зміни. Ми також покажемо вам, як налаштувати Git ігнорувати певні файли чи шаблони файлів, як швидко і легко скасувати помилки, як переглядати історію своїх проектів або зміни між комітами, а також як відправляти та отримувати зміни з віддалених репозиторіїв.

Створення Git-репозіторія

Для створення Git репозиторію використовують два основних підходи. Перший приймає існуючий проект або каталог і імпортує його в Git. Другий клонує існуючий репозіторій Git з іншого сервера.

Ініціалізація репозіторія в існуючому каталозі

Якщо ви починаєте використовувати Git для існуючого проекту, вам треба зайти в каталог проекту та виконати:

```
$ git init
```

Це створить новий підкаталог `.git`, який містить всі необхідні файли вашого репозіторія - скелет Git-репозиторія. На даний момент, у вашому проекті ще нічого не відстежується. (Див **Chapter 10** для отримання додаткової інформації про файли, що містяться в каталозі `.git`, котрий ви щойно створили.)

Якщо ви бажаєте додати існуючі файли під версійний контроль (на відміну від порожнього каталогу), ймовірно, вам слід проіндексувати ці файли і зробити перший коміт. Ви можете це зробити за допомогою декількох команд `git add`, що визначають файли, за якими ви бажаєте слідкувати, після яких треба виконати `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Перша версія проекту'
```

Ми розповімо що саме роблять ці команди за хвильку. Наразі, у вас є Git репозиторій з декількома файлами та першим комітом.

Клонування існуючого репозиторію

Якщо ви бажаєте отримати копію існуючого Git репозиторію – наприклад, проекту, в якому ви хочете прийняти участь – вам потрібна команда `git clone`. Якщо ви знайомі з іншими СКВ, наприклад Subversion, ви помітите, що команда називається “clone” (клонувати), а не “checkout” (перевірити). Це важлива відмінність – замість отримання просто робочої копії, Git отримує повну копію майже всіх даних, що є у сервера. Кожна версія кожного файлу в історії проекту витягується автоматично, коли ви виконуєте `git clone`. Насправді, якщо щось станеться з диском вашого серверу, ви зазвичай можете використати майже будь-який з клонів на будь-якому клієнті щоб повернути сервер до стану на момент клонування (ви можете втратити деякі серверні хуки (hook), проте усі дані під контролем версій повернуться – дивіться **“Отримання Git на сервері”** задля детальнішої інформації).

Щоб клонувати репозиторій треба використати команду `git clone [url]`. Наприклад, якщо ви бажаєте зробити клон бібліотеки Git libgit2, ви можете це зробити так:

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

Це створить директорію під назвою “libgit2”, проведе ініціалізацію директорії `.git`, забере всі дані для репозиторію, та приведе директорію до стану останньої версії. Якщо ви зайдете до нової директорії `libgit2`, ви побачите, що всі файли проекту на місці, готові для використання. Якщо ви бажаєте зробити клон репозиторія в

директорію з іншою назвою, ви можете задати її у наступному параметрі команди:

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

Ця команда робить те саме, що й попередня, тільки цільова директорія називається `mylibgit`.

Git має декілька різних протоколів передачі даних, які ви можете використовувати. Попередній приклад використовує протокол `https://`, проте ви також можете побачити `git://` або `user@server:шлях/до/репозиторію.git`, що використовує SSH протокол. **“Отримання Git на сервері”** познайомить вас з усіма можливими варіантами доступу до Git репозиторію, які може мати серверу, та “за” та “проти” кожного.

Запис змін до репозиторію

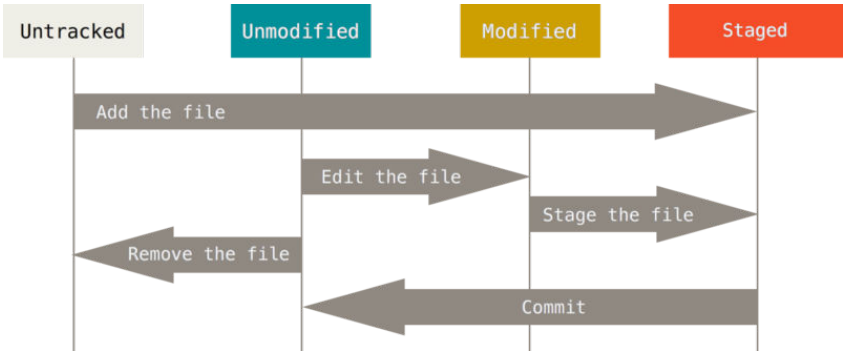
У вас є справжній Git репозиторій та робоча директорія з файлами цього проекту Вам треба зробити деякі зміни та записати їх у вашому репозиторії кожного разу, коли ваш проект набуває стану, що ви бажаєте зберегти.

Пам’ятайте, що кожен файл вашої робочої директорії може бути в одному з двох станів: контрольований (tracked) чи неконтрольований (untracked). Контрольовані файли – це файли, що були в останньому знімку. Вони можуть бути не зміненими, зміненими або індексованими. Неконтрольовані файли це все інше – будь-які файли у вашій робочій директорії, що не були у вашому останньому знімку та не існують у вашому індексі. Якщо ви щойно зробили клон репозиторію, усі ваші файли контрольовані та не змінені, адже ви щойно їх отримали та не редагували нічого.

По мірі редагування файлів, Git бачить, що вони змінені, адже ви їх змінили після останнього коміту. Ви індексуєте ці змінені фали та потім зберігаєте всі індексовані зміни, та цей цикл повторюється.

ЗОБРАЖЕННЯ 2-1

Цикл життя статусу ваших файлів.



Перевірка статусу ваших файлів

Щоб дізнатись, в якому стані ваші файли, варто скористатись командою `git status`. Якщо ви виконаєте цю команду відразу після клонування, ви маєте побачити таке:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Це означає, що ваша робоча директорія чиста – іншими словами, у вас нема контрольованих та змінених файлів. Git також не бачить неконтрольованих файлів, інакше він би їх тут вказав. Нарешті, ця команда показує вам, в якій ви зараз гілці та інформує вас про те, що вона не розбіглася з такою ж гілкою на сервері. Поки що, ця гілка завжди буде “master”, така гілка створюється автоматично. Це нас не обходить у цьому розділі. **Chapter 3** розповідає про гілки та посилання докладно.

Припустімо, ви додали новий файл до вашого проекту, простий файл README. Якщо файл раніше не існував, і ви виконаєте `git status`, ви побачите ваш неконтрольований файл так:

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Ви можете бачити, що ваш новий README файл неконтрольований, адже він під заголовком “Untracked files” у статусі. Неконтрольований (untracked) означає, що Git бачить файл, якого нема у попередньому знімку (коміті). Git не почне включати його до ваших комітів доки ви явно не скажете йому це зробити. Так зроблено щоб ви випадково не почали включати генеровані бінарні файли чи інші файли, які ви не збирались включати. Ви все ж таки хочете почати включати README, отже давайте почнемо контролювати файл.

Контролювання нових файлів

Щоб почати контролювати новий файл, вам треба використати команду `git add`. Почати контролювати файл README можна так:

```
$ git add README
```

Якщо ви знову виконаєте команду status, ви побачите, що ваш файл README тепер контролюється та готовий до включення до коміту:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

Ви можете зрозуміти, що цей файл доданий, бо він під заголовком “Changes to be committed”. Якщо ви створите коміт зміни зараз, версія файлу на момент коли ви виконали `git add` буде збережена в знімку в історії. Ви можете пригадати, що коли ви виконали `git init` раніше, ви потім виконали `git add (файли)` – це було зроблено щоб розпочати контролювати файли у вашій директорії. Команда `git add` приймає шлях файлу або директорії. Якщо це директорія, команда додає усі вайли в цій директорії рекурсивно.

Індексування змінених файлів

Змінімо файл, що вже контролюється. Якщо ви зміните файл “CONTRIBUTING.md”, що вже контролюється, та потім виконаєте команду `git status` знову, ви отримаєте щось на кшталт:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Файл “CONTRIBUTING.md” з’явився під секцією названою “Changed but not staged for commit” – це означає, що контрольований файл був редагований у робочій директорії проте його не індексували. Щоб проіндексувати його, виконайте команду `git add`. `git add` багатоцільова команда – її слід використовувати щоб почати контролювати нові файли, щоб додавати файли, та для інших речей, наприклад позначання конфліктних файлів як розв’язаних. Про неї краще думати “Додай цей зміст до наступного коміту” а не “додай цей файл до проекту”. Виконаймо `git add` зараз для індексації файлу “CONTRIBUTING.md”, а потім знову виконаємо `git status`:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Обидва файли індексовані та будуть включені до наступного коміту. Припустімо, що саме зараз ви пригадали маленьку зміну, яку ви хочете зробити в `CONTRIBUTING.md` до того, як зробити коміт з ним. Ви

знову його відкриваєте та редагуєте, і ви готові зробити коміт. Втім, виконаймо `git status` ще раз:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Якого біса? Тепер `CONTRIBUTING.md` є як в індексованих, *так і в* неіндексованих. Як таке можливо? Виявляється, що Git індексує файл саме таким, яким він був, коли ви виконали команду `git add`. Якщо ви зараз створите коміт, в історії збережеться версія `CONTRIBUTING.md`, яка була коли ви востаннє викликали `git add`, а не поточна версія файлу з вашої робочої директорії, коли ви виконаєте `git commit`. Якщо ви зміните файл після того, як виконаєте `git add`, вам треба знову виконати `git add` щоб проіндексувати останню версію файлу:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Короткий статус

Хоча вивід `git status` доволі вичерпний, він також дещо довгий. Git також пропонує опцію короткого перегляду статусу, щоб ви могли побачити свої зміни в більш компактному вигляді. Якщо ви виконаєте

`git status -s` або `git status --short`, ви отримаєте набагато простіший вивід.

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

Нові неконтрольовані файли позначаються `??`, нові індексовані файли позначаються `A`, змінені файли позначаються `M` тощо. Результат має дві колонки – ліва означає, що файл був індексований, а права що він був змінений. Наприклад у цьому виводі, файл `README` змінений у робочій директорії, проте не індексований, а файл `lib/simplegit.rb` змінений та індексований. `Rakefile` був змінений, індексований та знову змінений, тому є зміни в обох колонках.

Ігнорування файлів

Буває, що у вас є клас файлів, що ви не хочете щоб Git їх автоматично індексував чи навіть відображав як неконтрольовані. Зазвичай це автоматично згенеровані файли, наприклад файли логів або файли вироблені вашою системою збірки. У таких випадках, ви можете створити файл `.gitignore`, що містить взірці, яким відповідають ці файли. Ось приклад файлу `.gitignore`:

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

Перший рядок каже Git ігнорувати файли, що закінчуються на “.o” або “.a” – об’єктні та архівні файли, що можуть бути продуктами компіляції вашого коду. Другий рядок каже Git ігнорувати всі файли, що закінчуються на тильду (~), яка використовується багатьма текстовими редакторами (такими як Emacs) щоб позначати тимчасові файли. Ви також можете додати директорії log, tmp та pid, автоматично згенеровану документацію, тощо. Заповнення файлу `.gitignore` до початку праці зазвичай гарна думка, адже це допоможе вам

випадково не додати файли, які ви не хочете додавати до репозиторію Git.

Правила для взірців, які ви можете додати до файлу `.gitignore`:

- Пусті рядки та рядки, що починаються з #, ігноруються.
- Стандартні ґлоб взірці працюють.
- Ви можете завершити взірець похилою рискою (/) щоб позначити директорію.
- Ви можете відкинути взірець, якщо почнете його зі знаку оклику (!).

Glob patterns are like simplified regular expressions that shells use. Ґлоб (glob) взірці – це ніби спрощені регулярні вирази, що їх використовують оболонки. Зірочка (*) відповідає нулю або більше символам. [абв] відповідає будь-якому з символів всередині квадратних дужок (у цьому випадку а, б або в). Знак питання (?) відповідає одному символу. Квадратні дужки з символами, що розділені дефісом ([0-9]) відповідають будь-якому символу між ними (у даному випадку від 0 до 9). Ви можете використовувати дві зірочки щоб позначити вкладені директорії: a/**/z відповідає a/z, a/b/z, a/b/c/z тощо.

Ось ще один приклад файлу .gitignore:
///TODO

```
# Ніяких файлів .a
*.a

# Проте відстежувати lib.a, хоч ми й ігноруємо .a файли вище
!lib.a

# Ігнорувати тільки файл TODO у корені, не в інших теках subdir/TODO
/TODO

# Ігнорувати усі файли в теці build/
build/

# Ігнорувати doc/notes.txt, проте не doc/server/arch.txt
doc/*.txt

# Ігнорувати усі .txt файли в теці doc/
doc/**/*.txt
```

> GitHub підтримує доволі вичерпний список гарних прикладів файлів `.gitignore` для десятків проектів та мов за адресою *https://github.com/github/gitignore*, якщо ви бажаєте мати зразок для свого проекту.

Перегляд ваших доданих та недоданих змін

Якщо команда `git status` занадто зверхня для вас – ви хочете знати що саме ви змінили, а не просто які файли ви змінили – ви можете використати команду `git diff`. Ми докладніше розглянемо `git diff` пізніше, проте напевно найчастіше ви її будете використовувати для того щоб дізнатись дві речі: Що ви змінили, проте ще не індексували? Та що ви індексували та збираєтесь зберегти? Хоч `git status` відповідає на ці питання дуже загально – тільки показує список файлів, `git diff` показує вам усі індексовані та видалені рядки – латку, як вона є.

Припустімо, що ви внесли та проіндексували зміни до файлу `README` знову, а потім змінили файл `CONTRIBUTING.md` але не індексували його. Якщо ви виконаєте команду `git status`, ви знову побачите щось на кшталт:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Щоб побачити зміни, які ви ще не індексували, наберіть `git diff` без параметрів:

```
$ 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
```

Ця команда порівнює вашу робочу директорію з індексом. Результат показує вам зміни, котрі ви зробили проте не індексували.

Якщо ви хочете побачити, що ви індексували та ввійде до вашого наступного коміту, ви можете скористатись `git diff --staged`. Ця команда порівнює індексовані зміни з вашим останнім знімком:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Важливо пам'ятати, що команда `git diff` без опцій не відображає всіх змін з останнього коміту – тільки неіндексовані зміни. Це може збивати з пантелику, адже якщо ви проіндексували всі свої зміни, вивід `git diff` буде порожнім.

Наведемо інший приклад, припустимо, що ви проіндексували файл `CONTRIBUTING.md` та знову його відредагували, ви можете скористатись `git diff` щоб побачити індексовані та неіндексовані зміни. Якщо наше середовище виглядає наступним чином:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Тепер ви можете використати `git diff` щоб побачити неіндексовані зміни:

```
$ 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/PROJE
+test line
```

та `git diff --cached` щоб побачити наразі індексовані зміни (`--staged` та `--cached` мають однакове значення):

```
$ 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 У ЗОВНІШНІЙ УТИЛІТІ.

Ми продовжимо використати команду `git diff` різноманітними шляхами в решті книги. Є інший шлях подивитись на різницю між файлами, якщо вам більш до смаку графічна чи зовнішня програма для відображення різниці. Якщо ви виконаєте `git difftool` замість `git diff`, ви зможете використати будь-яку з програм на кшталт Araxis, emerge, vimdiff тощо. Виконайте `git difftool --tool-help` щоб дізнатись, що доступно на вашій системі.

Збереження ваших змін у комітах

Припустімо, що ваш індекс саме в стані, який ви бажаєте, та тепер ви можете створити коміт з ваших зміни. Пам'ятайте, що будь-які неіндексовані зміни – будь-які файли, що ви їх створили чи змінили, але ви не виконали `git add` після їх редагування – не потраплять до цього коміту. Вони так і залишаться зміненими файлами на вашому диску. У цьому випадку, припустімо, що останнього разу, коли ви виконали `git status`, ви побачили, що всі зміни індексовані, отже ви готові зберегти ваші зміни. Найпростіший спосіб створити коміт – набрати `git commit`:

```
$ git commit
```

Це запустить ваш редактор. (Це редактор, який встановлено в змінній середовища `$EDITOR` вашої оболонки – зазвичай vim або emacs, хоча ви можете налаштувати його як завгодно за допомогою команди `git config --global core.editor`, яку ви бачили в **Chapter 1**).

Редактор покаже вам наступний текст (це приклад екрану Vim):

```
# 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:
#       new file:   README
#       modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Ви бачите, згенероване повідомлення коміту містить закоментований останній вивід команди `git status` та один пустий

рядок нагорі. Ви можете видалити ці коментарі на написати своє повідомлення коміту, або можете залишити їх, щоб вони допомогли вам пригадати що ви зберігаєте. (Для навіть більш докладного нагадування про ваші зміни, ви можете передати опцію `-v` до `git commit`. Це призведе то того, що у вашому редакторі також будуть відображені всі зміни, що ввійдуть до коміту.) Коли ви виходите з редактору, Git створює коміт з цим повідомленням коміту (після видалення коментарів та змін до файлів).

Іншим чином, ви можете набрати повідомлення коміту прямо в команді `commit`, якщо напишете її після опції `-m`, ось так:

```
$ 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 README
```

Тепер ви створили свій перший коміт! Ви можете бачити, що команда `commit` розповіла вам дещо про коміт: до якої гілки ви зберегли зміни (`master`), який SHA-1 хеш отримав коміт (`463dc4f`), скільки файлів було змінено, та статистику щодо індексованих та видалених рядків у коміті.

Пам'ятайте, що коміт записує знімок, який ви створили в індексі. Усе, що ви не проіндексували, залишиться зміненим. Ви можете зробити інший коміт, щоб додати ці зміни до історії. Щоразу ви створюєте коміт, ви записуєте знімок вашого проекту, до якого ви можете повернутися або порівняти щось пізніше.

Обходимо індекс

Хоч індекс може бути неперевершено корисним для підготовки комітів саме до потрібного вам вигляду, іноді він може буди надто складним для ваших потреб. Якщо ви бажаєте обійти індекс, Git надає вам простий короткий шлях. Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part: Додавання опції `-a` до команди `git commit`, змушує Git автоматично додати кожен файл, що вже контролюється, до коміту, що дозволяє вам пропустити команди `git add`:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Зауважте, що вам не довелося виконувати `git add` до файлу "CONTRIBUTING.md" у цьому випадку до того, як ви створили коміт.

Removing Files

Щоб видалити файл з Git, вам треба прибрати його з контрольованих файлів (вірніше, видалити його з вашого індексу) та створити коміт. Команда `git rm` це робить, а також видаляє файл з вашої робочої директорії, щоб наступного разу він не відображався неконтрольованим.

Якщо ви просто видалите файл з вашої робочої директорії, він з'явиться під заголовком "Changed but not updated" (тобто, *неіндексованим*) виводу команди `git status`:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

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

Потім, якщо ви виконаєте `git rm`, файл буде індексованим на видалення:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

Наступного разу, коли ви створите коміт, файл зникне та більше не буде контрольованим. Якщо ви редагували файл та вже додали його до індексу, ви маєте примусово видалити його за допомогою опції `-f`. Це мера безпеки, щоб завадити випадковому видаленню даних, які ви не записали до знімку, і тому вони не можуть буди відновлені Git.

Інша корисна річ, яку ви можливо захочете зробити, це зберегти файл у робочій директорії, проте видалити його з індексу. Іншими словами, ви можете забажати зберегти файл на диску, проте більше не контролювати його Git. Це може бути корисним, якщо ви забули щось додати до свого файлу `.gitignore` та випадково проіндексували, наприклад великий файл журнал чи купу скомпільованих файлів `.a`. Щоб це зробити, скористайтеся опцією `--cached`:

```
$ git rm --cached README
```

Ви можете передавати команді `git rm` файли, директорії або файлові ґлоб шаблони. Це означає, що ви можете зробити щось таке:

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

Зверніть увагу на зворотню похилу (\) попереду *. Вона необхідна адже Git має власне розкриття шаблону на додаток до розкриття шаблону вашої оболонки. Ця команда видаляє всі файли, що мають `.log` розширення та знаходяться в директорії `log/`. Або, ви можете зробити щось таке:

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

Ця команда видаляє всі файли, що закінчуються на ~.

Пересування файлів

На відміну від багатьох інших СКВ, Git явно не стежить за пересуванням файлів. Якщо ви перейменуєте файл у Git, ніяких метаданих про це не буде збережено. Втім, Git доволі кмітливий, та може сам зрозуміти, що перейменування відбулося вже після нього – ми повернемося до виявлення пересування файлів трохи пізніше.

Тому присутність команди mv у Git трохи спантеличує. Якщо ви бажаєте перейменувати файл у Git, ви можете виконати щось таке:

```
$ git mv стара_назва нова_назва
```

і це зробить що вам треба. Насправді, якщо ви виконаєте щось таке і подивитесь на статус, ви побачите, що Git вважає, що перейменував файл:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Втім, це рівнозначно виконанню таких команд:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git без підказок розуміє, що це перейменування файлу, тому неважливо, чи ви перейменували файл так, або за допомогою команди mv. Єдина дійсна різниця в тому, що mv це одна команда замість трьох – ця команда існує тільки для зручності. Більш важливо, що ви можете використати будь-яку утиліту для перейменування файлу та зробити add/rm пізніше, до збереження коміту.

Перегляд історії комітів

Після того як ви створили декілька комітів, або якщо ви зробили клон репозиторію з існуючою історією комітів, ви напевно забажаєте дізнатись, що було відбувалося. Найбільш могутньою утилітою для цього є команда `git log`.

Ці приклади використовують дуже простий проект під назвою “simplegit”. Щоб отримати цей проект, виконайте

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

Якщо ви виконаєте `git log` у цьому проекті, ви маєте побачити щось на кшталт:

```
$ 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
```

Якщо не передати жодної опції до команди `git log`, вона виведе зроблені коміти репозиторію у зворотньому хронологічному порядку – тобто, найновіші коміти будуть показані першими. Як бачите, ця команда показує для кожного коміту його SHA-1 хеш, ім’я та пошту автора, дату запису, та повідомлення коміту.

Існує величезне різноманіття опцій до команди `git log` щоб відобразити саме те, що ви хочете. Тут ми продемонструємо вам найпоширеніші.

Одна з найкорисніших опцій це `-p`, котра показує різницю привнесену при кожному коміті. Ви також можете використати `-2`, що обмежує кількість виводу тільки до двох останніх елементів:

```
$ 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
```

Ця опція показує ту саму інформацію та ще зміни відразу після кожного елементу. Це дуже корисно для перегляду коду або швидкого перегляду що сталося протягом декількох комітів, що їх додав співробітник. Ви можете також використати ряд підсумкових опцій з `git log`. Наприклад, якщо ви бажаєте побачити дещо скорочену статистику для кожного коміту, ви можете скористатись опцією `--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(+)
```

Як бачите, опція `--stat` друкує під кожним комітом список змінених файлів, скільки файлів було змінено, скільки рядків у кожному файлі було додано та видалено. Також видає підсумок інформації наприкінці.

Інша дійсно корисна опція це `--pretty`. Ця опція змінює вивід — відображає його в іншому форматі. Вам доступні декілька вбудованих опцій формату. Опція `oneline` друкує кожен коміт в один рядок, що корисно, якщо ви дивитесь на багато комітів. На додаток, опції `short`,

`full` та `fuller` показують вивід приблизно в такому ж форматі, але зменшують чи збільшують кількість інформації, відповідно:

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

Найцікавіша опція це `format`, що дозволяє вам визначити свій власний формат виводу. Це особливо корисно, якщо ви генеруєте вивід для розбору програмою, адже ви можете явно вказати формат, та ви будете знати, що він не зміниться у наступних версіях 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 описує деякі найкорисніші опції формату.

ТАБЛИЦЯ 2-1. *Корисні опції для `git log --pretty=format`*

Опція	Опис Виводу
`%H`	Хеш коміту
`%h`	Скорочений хеш коміту
`%T`	Хеш дерева
`%t`	Скорочений хеш дерева
`%P`	Хеши батьків
`%p`	Скорочені хеши батьків
`%an`	Ім’я автора
`%ae`	Поштова адреса автора
`%ad`	Дата автора (формат враховує опцію --date=option)
`%ar`	Відносна дата автора
`%cn`	Ім’я користувача, що зробив коміт (далі фіксатор)
`%ce`	Поштова адреса фіксатора

`%cd`	Дата фіксатора
`%cr`	Відносна дата фіксатора
`%s`	Тема

Вам може стати цікаво в чому різниця між *автором* та *фіксатором*. Автор це людина, що спочатку зробила роботу, тоді як фіксатор – це людина, яка востаннє застосувала роботу. Отже, якщо ви відправили патч до проекту та один з програмістів ядра застосує вашу латку, ви обидва будете згадані – ви як автор, а програміст ядра як фіксатор. Ми більше про це поговоримо у **Chapter 5**.

Опції oneline та format особливо корисні з іншою опцією `log`, що називається `--graph`. Ця опція додає маленький гарний ASCII граф, що показує історію ваших гілок та зливань:

```
$ 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
```

Цей тип виводу стане цікавішим, коли ми будемо розповідати про гілки та злиття у наступному розділі.

Це тільки декілька простих опцій формату виводу `git log` – їх набагато більше. **Table 2-2** наводить опції, про які ми вже розповідали, та деякі інші розповсюджені опції формату, що можуть бути корисними, з поясненням того, як вони змінюють вивід команди log.

ТАБЛИЦЯ 2-2. *Розповсюджені опції `git log`*

Опція	Опис
`-p`	Показує зміни файлів кожного коміту
`--stat`	Показує статистику змінених файлів для кожного коміту.

Опція	Опис
`--shortstat`	Відображає тільки рядок зміни/вставки/видалення з опції --stat.
`--name-only`	Показує список змінених файлів після інформації про коміт.
`--name-status`	Показує список змінених файлів з інформацією додано/змінено/видалено.
`--abbrev-commit`	Показує тільки перші декілька символів SHA-1 суми замість усіх 40.
`--relative-date`	Відображає дату у відносному форматі (наприклад, “2 тижня тому”) замість використання повного формату дати.
`--graph`	Відображає ASCII граф історії гілок та зливань поряд зі звичайним виводом.
`--pretty`	Показує коміти в альтернативному форматі. Можливі значення: online, short, full, fuller та format (якому задаєте свій власний формат).

Обмеження виводу журналу

На додаток до опцій, що контролюють формат виводу, `git log` також приймає декілька корисний обмежувальних опцій – тобто, опцій, що дозволяють вам показувати тільки підмножину комітів. Ви вже бачили одну таку опцію: `-2`, що показує тільки останні два коміти. Насправді, ви можете використати `-<n>`, де `n` це будь-яке ціле число, щоб показати останні `n` комітів. Однак навряд чи ви будете використовувати це часто, адже Git зазвичай передає весь свій вивід переглядачу, отже ви бачите тільки одну сторінку журналу за раз.

Втім, опції обмеження по часу, такі як `--since` (від) та `--until` (до) дуже корисні. Наприклад, ця команда отримає список комітів за останні два тижні:

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

Ця команда працює з різноманітними форматами – ви можете задати точну дату, наприклад `"1991-08-24"`, чи відносну дату, наприклад `"2 years 1 day 3 minutes ago"`.

Ви також можете відсіювати список комітів, що відповідають якомусь критерію пошуку. Опція `--author` дозволяє вам відбирати по

заданому автору, а опція `--grep` дозволяє вам шукати ключові слова в повідомленнях комітів. (Зауважте, що якщо ви бажаєте відсіяти і по автору, і по змісту, вам необхідно додати опції `--all-match`, інакше команда видасть коміти, що задовольняють хоча б один з критеріїв)

Інша дійсно корисна опція `-S` приймає рядок та відображає тільки коміти, що в зміні змісту додали або видалили цей рядок. Наприклад, якщо ви бажаєте знайти останній коміт, що додав чи видалив посилання на певну функцію, вам варто викликати:

```
$ git log -Sfunction_name
```

Остання дійсно корисна опція, яку можна передати до `git log` — це шлях. Якщо ви зазначите директорію або ім'я файлу, ви можете обмежити вивід до комітів, що змінювали ці файли. Це завжди остання опція та зазвичай перед нею ставлять подвійний дефіс (`--`) щоб відділити шляхи від опцій.

У **Table 2-3** ми перелічили ці та ще декілька інших розповсюджених опцій для довідки.

ТАБЛИЦЯ 2-3. *Опції для обмеження виводу `git log`*

Опція	Опис
`-(n)`	Показати тільки останні n комітів
`--since`, `--after`	Обмежитись комітами, що були створені після переданої дати.
`--until`, `--before`	Обмежитись комітами, що були створені до переданої дати.
`--author`	Показати тільки ті коміти, автор яких збігається із переданим.
`--committer`	Показати тільки ті коміти, фіксатор яких збігається із переданим
`--grep`	Показати тільки ті коміти, повідомлення яких містить рядок.
`-S`	Показати тільки ті коміти, в яких додали або видалили рядок, що містить переданий рядок.

Наприклад, якщо ви бажаєте побачити, в яких комітах були змінені тестові файли в коді Git, що були збережені Junio Hamano та не були злиттям у жовтні 2008 року, ви можете виконати таку команду:

```
$ 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 branch
```

З приблизно 40000 комітів в історії коду Git, ця команда відображає тільки 6, що задовольняють цим критеріям.

Скасування речей

У будь-який момент, ви можете забажати щось скасувати. Тут, ми розглянемо декілька базових утиліт для скасування змін, що ви зробили. Будьте обережними, адже ви не завжди в змозі скасувати деякі з цих скасувань. Це одна з не багатьох ділянок, де Git може втратити вашу працю, якщо ви помилитесь.

Одне з розповсюджених скасувань відбувається, коли ви зробили коміт зарано, можливо забули додати деякі файли, або ви зіпсували повідомлення коміту. Якщо ви хочете знову спробувати зробити коміт, ви можете виконати команду `commit` з опцією `--amend`:

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

Ця команда бере ваш індекс та використовує його для коміту. Якщо ви нічого не змінили з останнього коміту (наприклад, ви виконуєте цю команду відразу після попереднього коміту), то ваш знімок буди виглядати так само, та все що ви можете зробити — це змінити повідомлення коміту.

З'явиться вже знайомий редактор повідомлення коміту, проте в ньому вже міститься повідомлення вашого попереднього коміту. Ви можете відредагувати повідомлення як завжди, тільки воно перепише ваш попередній коміт.

Наприклад, якщо ви зробили коміт, а потім збагнули, що забули додати якісь зміни у файлі, які мають потрапити до цього коміту, ви можете зробити так:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

У підсумку ви отримаєте єдиний коміт – другий коміт замінить результати першого.

Вилучання файла з індексу

Наступні дві секції покажуть, як скіпати ваші зміни в індексі та робочій директорії. Гарно те, що команда, яку ви використовуєте для визначення статусу цих двох областей, також нагадує вам, як скасувати зміни в них. Наприклад, припустімо, що ви змінили два файли та хочете зберегти їх у двох окремих змінах, проте випадково набрали `git add *` та проіндексували їх обох. Як ви можете вилучити один з них? Команда `git status` нагадує вам:

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Прямо під текстом “Changes to be committed” (зміни, що буде збережено), написано “use `git reset HEAD <file>...` to unstage” (скористайтесь `git reset HEAD <file>...` щоб вилучити) Отже, скористаймося цією порадою, щоб вилучити файл `CONTRIBUTING.md`:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M       CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Команда трохи дивна, проте працює. Файл `CONTRIBUTING.md` змінений але неіндексований.

Хоча `git reset` *може* бути небезпечним, якщо ви викличете його з `--hard`, у цьому випадку файл у вашій робочій директорії не чіпали. Виклик `git reset` без опцій безпечний – він змінює тільки індекс.

Поки що цей магічний виклик це все, що вам треба знати про команду `git reset`. Ми розповімо набагато докладніше про `reset` та як його використовувати щоб робити дійсно цікаві речі у **“Усвідомлення скидання (reset)”**.

Скасування змін у зміненому файлі

Що як ви зрозумієте, що ви не хочете залишити ваші зміни до файлу `CONTRIBUTING.md`? Як ви можете легко скасувати зміни – повернути файл до стану, в якому він був під час вашого останнього коміту (або не вашого, байдуже як ви отримали його)? На щастя, `git status` розповідає вам і про це. У виводі останнього прикладу, неіндексована область виглядає так:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Тут чітко розповідають, як скасувати ваші зміни. (третій рядок перекладається (використайте "git checkout -- <file>..." щоб скасувати зміни у вашій робочій директорії)) Так і зробимо:

```
$ git checkout -- CONTRIBUTING.md
$ git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Як ви бачите, ваших змін більше нема.

Важливо усвідомлювати, що команда `git checkout -- [file]` небезпечна. Будь-які зроблені зміни зникли – ви просто скопіювали інший файл поверх них. Ніколи не використовуйте цю команду, якщо у вас нема абсолютної впевненості, що цей файл вам не потрібен.

Якщо ви бажаєте зберегти зроблені зміни до файлу, проте вам необхідно їх тимчасово прибрати, ми розповімо про хования та гілки в **Chapter 3**; це зазвичай кращі засоби.

Пам'ятайте, все *збережене в комітах* Git майже завжди може бути відновлено. Навіть коміти у видалених гілках чи коміти, переписані за допомогою `--amend`, можуть бути відновлені (дивіться **"Data Recovery"** задля відновлення даних). Однак, будь-що втрачене до коміту ви навряд чи колись ще раз побачите.

Взаємодія з віддаленими сховищами

Задля співпраці з будь-яким проектом Git, вам необхідно знати, як керувати віддаленими сховищами. Віддалені сховища — це версії вашого проекту, що розташовані в Інтернеті, або десь у мережі. Їх може бути декілька, кожне зазвичай або тільки для читання, або для читання та змін. Співпраця з іншими вимагає керування цими віддаленими сховищами, надсилання (`pushing`) та отримання (`pulling`) даних до та з них, коли ви хочете зробити внесок. Керування віддаленими сховищами потребує знань про додавання віддалених сховищ, видалення сховищ, що більше не потрібні, керування різноманітними віддаленими гілками та визначення слідкування за ними, і багато іншого. У цій секції, ми пройдемо ці вміння керування віддаленими сховищами.

Дивимось на ваші сховища

Щоб побачити, які віддалені сервера ви налаштували, ви можете виконати команду `git remote`. Вона виводить список коротких імен

кожного віддаленого сховища, яке ви задали. Якщо ви отримали своє сховище клонуванням, ви маєте побачити хоча б `origin` – таке ім'я Git дає серверу, з якого ви зробили клон:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Ви також можете дати опцію `-v`, яка покаже вам посилання, які Git зберігає та використовує при читанні та записі до цього сховища:

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

Якщо у вас більш ніж одне віддалене сховище, ця команда описує їх усі. Наприклад, сховище з декількома віддаленими сховищами для роботи з багатьма співробітниками може виглядати так.

```
$ 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)
```

Це означає, що ми можемо отримувати (`pull`) внески з будь-якого з цих користувачів доволі легко. Ми також можемо мати дозвіл на надсилання змін до якихось з них, хоч ми й не можемо цього тут визначити.

Завважте, що ці сховища використовують різноманітні протоколи. Ми більше про це поговоримо в **“Отримання Git на сервері”**.

Додавання віддалених сховищ

Ми згадували і навіть продемонстрували додавання віддаленого сховища у попередній секції, проте ось так ми можемо зробити це явно. Щоб додати нове віддалене Гіт сховище під заданим ім’ям, на яке ви можете легко посилатись, виконайте `git remote add [ім’я] [посилання]`:

```
$ 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)
```

Тепер ви можете використати рядок `pb` в командному рядку замість повного посилання. Наприклад, якщо ви хочете отримати (`fetch`) усю інформацію, яке є в Пола, проте її нема у вашому сховищі, ви можете виконати `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)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Гілка `master` Пола тепер доступна локально як `pb/master` – ви можете злити її з однією з ваших гілок, або зробити з неї локальну гілку, якщо хочете оглянути її. (Ми розповімо що таке гілки та як ними користуватися набагато докладніше в **Chapter 3**.)

Отримання (fetching) та затягування (pulling) з ваших віддалених сховищ

Як ви щойно побачили, щоб отримати дані з ваших віддалених проектів, ви можете виконати:

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

Ця команда заходить на віддалений проект та забирає звідти усі дані, котрих у вас досі нема. Після цього, у вас будуть посилання на всі гілки з того сховища, які ви можете зливати або оглядати в будь-який час.

Якщо ви зробили клон сховища, команда автоматично додає це віддалене сховище під ім'ям "origin". Отже, `git fetch origin` отримує будь-яку нову працю, що її виклали на той сервер після того, як ви зробили його клон (або востаннє отримували зміни з нього). Важливо зауважити, що команда `git fetch` отримує дані до вашого локального сховища – вона автоматично не зливає їх з вашою роботою, та не змінює вашу поточну працю. Вам буде потрібно вручну її злити, коли ви будете готові.

Якщо ваша гілка налаштована слідкувати за віддаленою гілкою (докладніше в наступній секції та **Chapter 3**), ви можете виконати команду `git pull` щоб автоматично отримати зміни та злити віддалену гілку до вашої поточної гілки. Це може бути легшим та зручнішим методом для вас. Та команда `git clone` автоматично налаштовує вашу локальну гілку master слідкувати за віддаленою гілкою master (хоча вона може називатись і по іншому) на віддаленому сервері, з якого ви зробили клон. Виконання `git pull` зазвичай дістає дані з серверу, з якого ви зробили клон, та намагається злити її з кодом, над яким ви зараз працюєте.

Надсилання змін до ваших віддалених сховищ

Коли ви довели свій проект до стану, коли хочете їм поділитись, вам треба надіслати (push) ваші зміни нагору (upstream). Це робиться простою командою: `git push [ім'я сховища] [ім'я гілки]`. Якщо ви бажаєте викласти свою гілку master до вашого серверу `origin` (клонування зазвичай налаштовує обидва імені для вас автоматично), ви можете виконати наступне для надсилання всіх зроблених комітів до сервера:

```
$ git push origin master
```

Ця команда спрацює тільки в разі, якщо ви зробили клон з серверу, до якого у вас є доступ на запис, та ніхто не оновлював його після цього. Якщо хтось інший зробив клон та надіслав щось назад перед вами, вашій спробі буде слушно відмовлено. Вам доведеться спершу забрати їх працю та вбудувати її до вашої до того, як вам дозволять надіслати свої зміни. Докладніше про надсилання змін до віддалених серверів у **Chapter 3**.

Оглядання віддаленого сховища

Якщо ви бажаєте більше дізнатись про окреме віддалене сховище, ви можете використати команду `git remote show [ім'я сховища]`. Якщо ви виконаєте цю команду з окремим ім'ям, наприклад `origin`, ви отримаєте щось на кшталт:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Вона виводить посилання для віддаленого сховища, а також інформацію про слідкування за гілками. Команда ґречно розповідає вам, що якщо ви на гілці master та виконаєте команду `git pull`, вона автоматично зіллє гілку master з віддаленою після того, як отримає всі дані з віддаленого сховища. Також видано список усіх віддалених посилань, які були забрані.

Ви напевно зустрінете такий простий приклад. Втім, коли ви почнете працювати з Git інтенсивніше, ви можете побачити набагато більше інформації від `git remote show`:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to date)
    markdown-strip                 pushes to markdown-strip                 (up to date)
    master                         pushes to master                         (up to date)
```

Ця команда показує, до яких гілок автоматично надсилаються ваші зміни, коли ви виконуєте `git push`, доки перебуваєте на певної гілці. Вона також показує, яких віддалених гілок з серверу у вас нема, які віддалені гілки, що у вас є, були видалені з серверу, та декілька гілок, що автоматично зливаються, коли ви виконуєте `git pull`.

Видалення та перейменування віддалених сховищ

Якщо ви бажаєте перейменувати віддалене сховище, ви можете виконати `git remote rename`. Наприклад, щоб перейменувати `pb` на `paul`, ви можете зробити це за допомогою `git remote rename`:

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

Варто зазначити, що це змінює і назви ваших віддалених гілок. Що раніше мало назву `pb/master`, тепер називається `paul/master`.

Якщо ви з якоїсь причини бажаєте видалити віддалене сховище – ви перемістили сервер або більше не використовуєте якесь дзеркало,

або можливо хтось припинив співпрацю – ви можете використати `git remote rm`:

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

Теґування

Як і більшість СКВ, Git дозволяє поставити теґ на окремому моменті історії, що чимось видатний. Зазвичай ця функціональність використовується щоб позначити релізи (v1.0 тощо). У цій секції, ви дізнаєтесь, як отримати список доступних теґів, як створювати нові теґи, та які типи теґів існують.

Показати ваші теґи

Отримати список доступних теґів у Git елементарно. Просто наберіть `git tag`:

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

Ця команда виводить список теґів в алфавітному порядку. Цей порядок насправді неважливий.

Ви також можете шукати теґи по шаблону. Наприклад, сховище Git містить більш ніж 500 теґів. Якщо вас цікавлять виключно версії 1.8.5, ви можете виконати:

```
$ 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
```

Створення теґів

Git має два головних типу теґів: легкі та анотовані.

Легкий теґ дуже схожий на гілку, що не змінюється – це просто вказівник на певний коміт.

Анотовані теґи, з іншого боку, зберігаються як повноправні об'єкти у базі даних Git. Для них обчислюється сума, вони зберігають ім'я та поштову адресу автора, дату створення, та повідомлення теґу. Також вони можуть бути підписані та перевірені за допомогою GNU Privacy Guard (GPG). Взагалі рекомендується використовувати анотовані теґи, щоб ви могли мати всю цю інформацію. Проте якщо вам цього не потрібно та ви бажаєте створити тимчасовий теґ, вам доступні легкі теґи.

Анотовані теґи

Створити анотований теґ у Git просто. Найлегший спосіб — додати `-a` до команди `tag`:

```
$ git tag -a v1.4 -m 'моя версія 1.4'
$ git tag
v0.1
v1.3
v1.4
```

`-m` визначає повідомлення теґу, що в ній буде збережено. Якщо ви не вкажете повідомлення анотованого теґу, Git запустить ваш редактор щоб ви могли його набрати.

Ви можете побачити дані теґу та коміт, на який він вказує, за допомогою команди `git show`:

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

my 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
```

Це показує інформацію про автора теґу, дату створення теґу, та повідомлення перед інформацією про коміт.

Легкі теґи

Другий спосіб позначати коміти — за допомогою легких позначок. Це просто хеш коміту збережений у файлі – ніякої іншої інформації не зберігається. Щоб створити легкий теґ, не додавайте опції `-a`, `-s` та `-m`:

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

Цього разу, якщо ви виконаєте `git show` з теґом, ви не побачите додаткової інформації про теґ. Команда покаже тільки коміт:

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

    changed the version number
```

Створення теґів пізніше

Ви також можете зробити теґ до комітів, від котрих ви вже пішли. Припустімо, що ваша історія комітів виглядає так:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
```

```
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Тепер, припустимо ви забули створити теґ до версії проекту v1.2, що має бути на коміті “updated rakefile”. Ви можете додати теґ і зараз. Щоб створити теґ до коміту, вам треба дописати суму коміту (чи її частину) наприкінці команди:

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

Ви можете побачити, що ви створити теґ до коміту:

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

Розповсюдження теґів

Без додаткових опцій команда `git push` не передає теґи на віддалені сервери. Вам доведеться явно надсилати теґи на спільний сервер

після створення. Цей процес не відрізняється від розповсюдження віддалених гілок – вам треба виконати `git push origin [ім'я теґу]`.

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

Якщо у вас багато теґів, та ви хочете надіслати їх разом, ви також можете використати опцію `--tags` команди `git push`. Це передасть усі ваші теґи до віддаленого серверу, яких там досі нема.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
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-lw -> v1.4-lw
```

Тепер, коли хтось інший зробить клон або отримає зміни з вашого сховища, він отримає також усі ваші теґи.

Переключення до теґів

Ви не можете дійсно переключитися (`check out`) на теґ у Git, адже він не може бути переміщеним. Якщо ви бажаєте привести вашу робочу директорію сховища до стану, що виглядає як певний теґ, ви можете створити нову гілку з цього теґу за допомогою `git checkout -b [ім'я гілки] [ім'я теґу]`:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Авжеж, ви можете це зробити та створити коміт, ваша гілка `version2` буде трохи відрізнятися від вашого теґу `v2.0.0`, адже вона переміститься вперед до ваших нових змін, отже будьте обережні.

Псевдоніми Git

До того, як завершити розділ про базовий Git, є ще одна маленька підказка, щоб зробити ваше користування Git простішим, легшим та більш знайомим: псевдоніми. Ми більше не будемо про них згадувати та будемо вважати, що ви використовуєте їх самостійно в решті книги, проте напевно вам слід знати, як їх використовувати.

Git сам не намагається здогадатись, яку команду ви набрали, якщо ви набрали її частково. Якщо ви не хочете набирати команди Git повністю, ви легко можете налаштувати псевдоніми для кожної команди за допомогою `git config`. Ось декілька прикладів як це можна зробити:

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

Це означає, що, наприклад, замість `git commit` ви можете набрати просто `git ci`. З часом, ви напевно почнете використовувати й інші команди часто. Не вагайтесь створювати нові псевдоніми.

Ця техніка також може бути дуже корисною для створення нових команд, які ви гадаєте мають існувати. Наприклад, щоб виправити незручність, яку ми бачили при скасуванні змін до файлу, ви можете додати свою власну команду unstage (видалити з індексу) до Git:

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

Після цього наступні команди еквівалентні:

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

Це здається трохи яснішим. Також нерідко додають команду `last` (останній), ось так:

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

Тепер ви легко можете побачити останній коміт:

```
$ 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>
```

Як ви напевно зрозуміли, Git просто підставляє замість псевдоніму його значення. Проте, можливо ви бажаєте виконати зовнішню команду, а не команду Git. У цьому разі, треба почати команду зі знаку оклику `!`. Це корисно, якщо ви бажаєте створити власні інструменти для роботи зі сховищем Git. Ми продемонструємо це, створивши псевдонім `git visual` для виконання `gitk`:

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

Підсумок

На цей час, ви можете робити всі основні операції Git - створювати або клонувати репозиторій, вносити зміни, індексувати зміни та створювати коміти, переглядати історію всіх змін у репозиторії. Далі, ми розглянемо вбивчу особливість Git: його моделі галуження.

Галуження в git 3

Майже кожна система контролю версій підтримує гілки (branches) в певній мірі. Галуження - це відмежування від основної лінії розробки для продовження своєї частини роботи та уникнення конфліктів з основною лінією. В багатьох системах контролю версій цей процес “дорогий”, часом вимагає створювати копію коду, що може зайняти багато часу для великих проектів.

Дехто вважає гілки Git вбивчою особливістю, що вирізняє Git від інших систем. Що ж в них такого особливого? Гілки Git надзвичайно легкі, операції галуження майже миттєві, перехід між гілками зазвичай теж. На відміну від інших систем, Git заохочує схеми, де гілки часто створюються та зливаються, навіть кілька разів на день. Розуміння та вміння працювати з цією “фішкою” дає вам потужний та унікальний інструмент, що може кардинально змінити ваш процес розробки.

Гілки у кількох словах

Щоб дійсно зрозуміти як Git працює з гілками, нам треба повернутись назад та розібратись, як Git зберігає дані.

Як ви можете пам’ятати з **Chapter 1**, Git зберігає дані не як послідовність змін, а як послідовність знімків.

Коли ви фіксуєте зміни, Git зберігає об’єкт фіксації, що містить вказівник на знімок змісту, який ви додали. Цей об’єкт також містить ім’я та поштову адресу автора, набране вами повідомлення та вказівники на фіксацію або фіксації, що були прямо до цієї фіксації (батько чи батьки): нуль для першої фіксації, одна фіксація для нормальної фіксації, та декілька фіксацій для фіксацій, що вони є результатом злиття двох чи більше гілок.

Щоб це уявити, припустимо, що у вас є тека з трьома файлами, які ви додали та зафіксували. Додання файлів обчислює контрольну суму для кожного (SHA-1 хеш про котрий ми згадували в **Chapter 1**), зберігає

версію файлу в сховищі Git (Git називає їх блобами), та додає їх контрольні суми до області додавання:

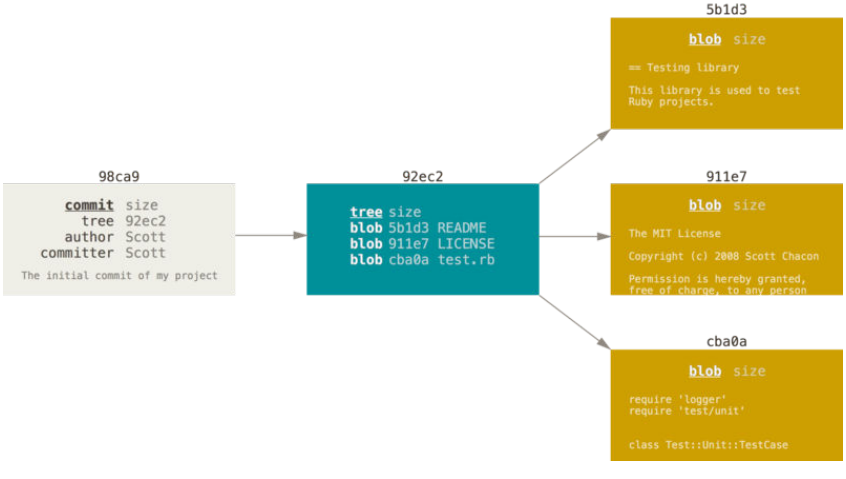
```
$ git add README test.rb LICENSE
$ git commit -m 'Перша фіксація проекту'
```

Коли ви створили фіксацію за допомогою `git commit`, Git обчислив контрольну суму кожної теки (у цьому випадку, тільки кореневої теки) та зберігає ці об'єкти дерева в сховищі Git. Потім Git створює об'єкт фіксації, що зберігає метадані та вказівник на корінь дерева проекту, щоб він міг відтворити цей знімок, коли потрібно.

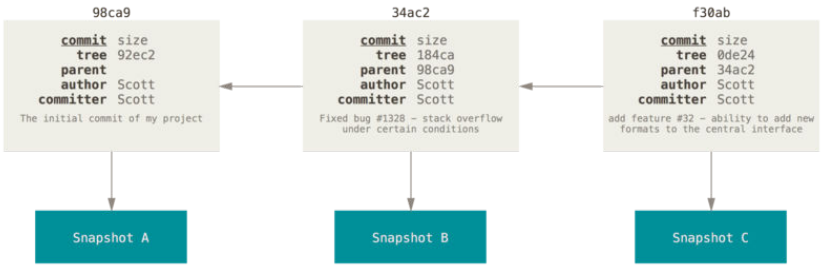
Ваше Git сховище тепер зберігає п'ять об'єктів: по одному блобу зі змістом на кожен з трьох файлів, одне дерево, що перелічує зміст теки та вказує, які файли зберігаються у яких блобах, та одну фіксацію, що вказує на корінь дерева, та зберігає метадані фіксації.

ЗОБРАЖЕННЯ 3-1

Фіксація як дерево



Якщо ви зробите якісь зміна та зафіксуєте знову, наступна фіксація буде зберігати вказівник на попередню.

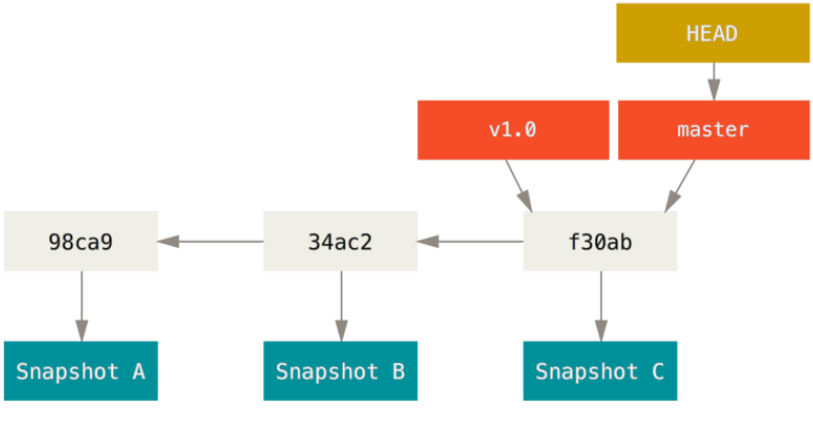


ЗОБРАЖЕННЯ 3-2

Фіксації та їх батьки

Гілка в Git це просто легкий вказівник, що може пересуватись, на одну з цих фіксацій. Загальноприйнятим ім'ям першої гілки в Git є `master`. Коли ви почнете робити фіксації, вам надається гілка `master`, що вказує на останню зроблену фіксацію. Щоразу ви фіксуєте, вона переміщується вперед автоматично.

Гілка "master" у Git не має нічого особливого. Вона нічим не відрізняється від інших. Єдина причина, чому майже кожне сховище має таку гілку – команда `git init` автоматично її створює, і більшість людей не мають клопоту змінити її.



ЗОБРАЖЕННЯ 3-3

Гілка та її історія фіксацій

Створення нової гілки

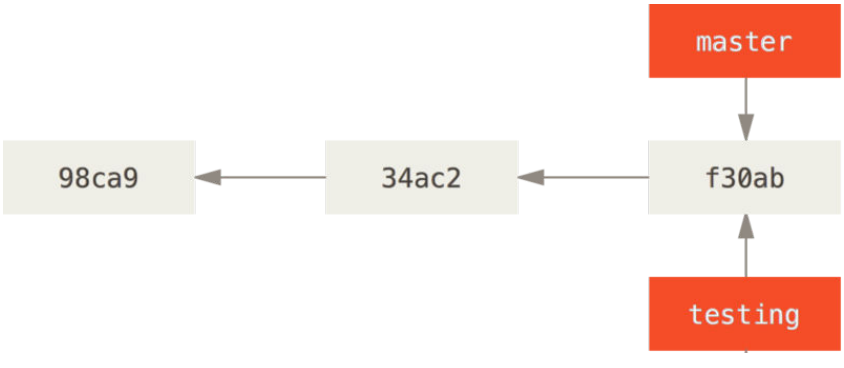
Що відбувається, якщо ви створюєте нову гілку? Ну, це створює новий вказівник, щоб ви могли пересуватися. Припустімо, ви створюєте нову гілку під назвою testing. Ви це робите за допомогою команди `git branch`:

```
$ git branch testing
```

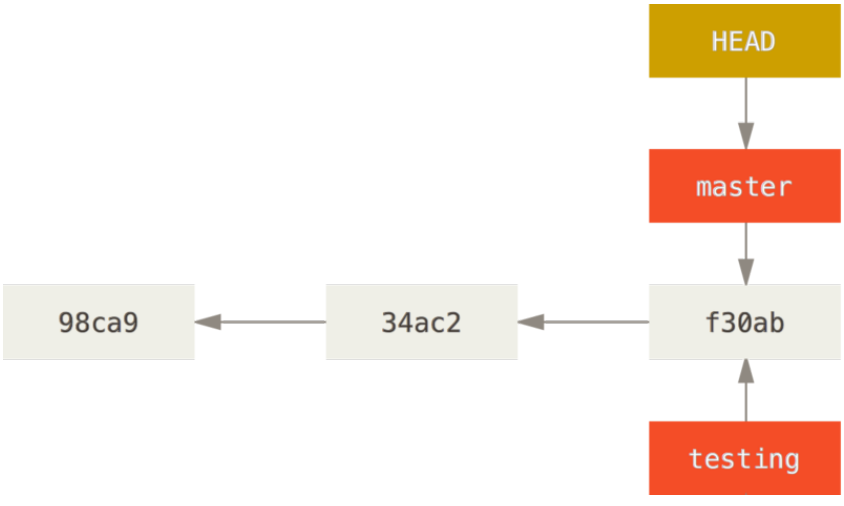
Це створює новий вказівник на фіксацію, в якій ви зараз знаходитесь.

ЗОБРАЖЕННЯ 3-4

Дві гілки вказують на одну послідовність фіксацій



Звідки Git знає, на якої гілці ви зараз знаходитесь? Він зберігає особливий вказівник під назвою HEAD. Завважте, що це геть інша концепція HEAD, ніж в інших СКВ, з якими ви могли працювати, таких як Subversion чи CVS. У Git це просто вказівник на локальну гілку, на якій ви знаходитесь. В даному випадку, ви досі на гілці master. Команда `git branch` тільки *створює* нову гілку – вона не переключає на цю гілку.



ЗОБРАЖЕННЯ 3-5

HEAD вказує на гілку

Ви легко можете це побачити за допомогою простої опції команди `git log`, що може показати куди вказують вказівники гілок. Ця опція називається `--decorate`.

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

Як бачите, гілки “master” та “testing” прямо поруч з фіксацією `f30ab`.

Переключення гілок

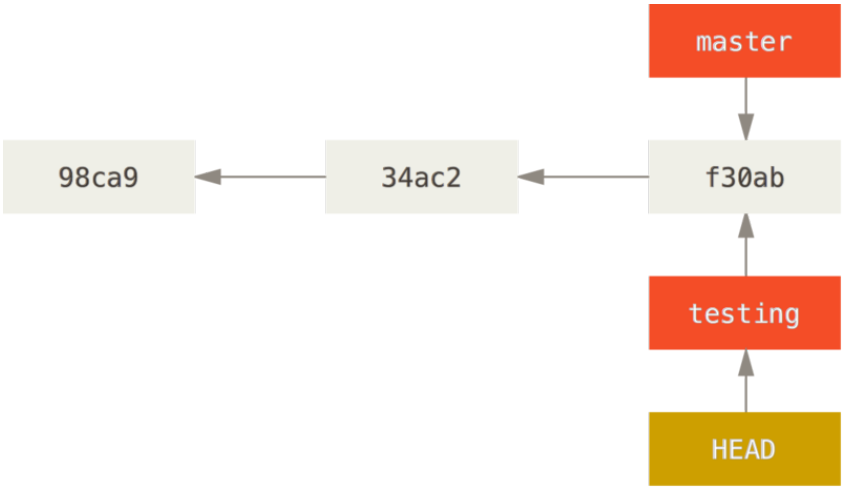
Щоб переключитися на існуючу гілку, треба виконати команду `git checkout`. Переключимося на нову гілку testing:

```
$ git checkout testing
```

Це пересуває `HEAD`, щоб він вказував на гілку `testing`.

ЗОБРАЖЕННЯ 3-6

HEAD вказує на поточну гілку

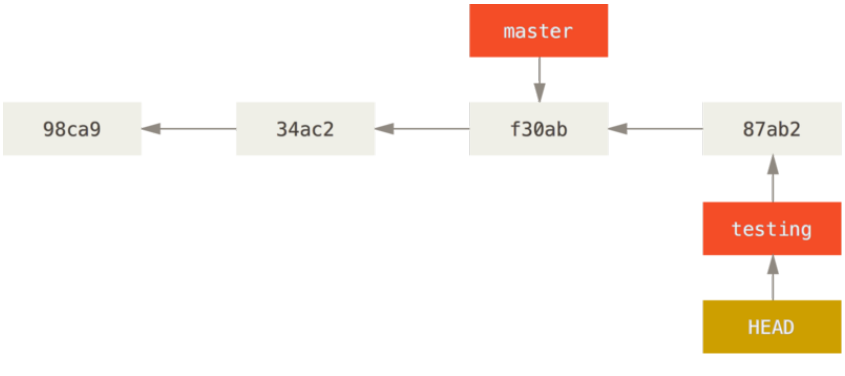


Чому це важливо? Ну, давайте зробимо ще одну фіксацію:

```
$ vim test.rb
$ git commit -a -m 'Зробив зміни'
```

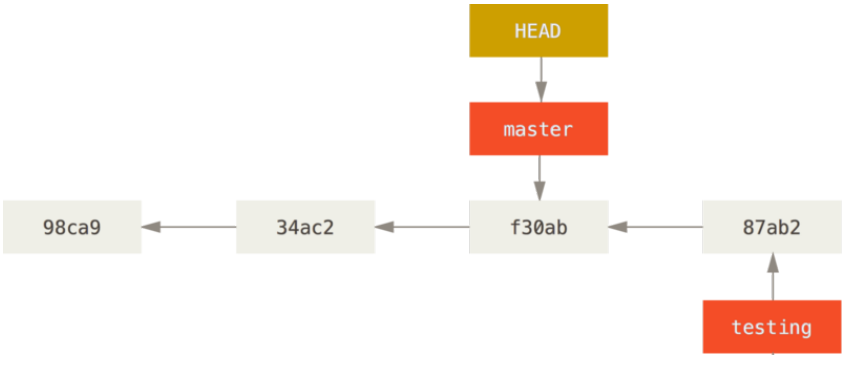
ЗОБРАЖЕННЯ 3-7

Гілка HEAD пересувається уперед при фіксації



Це цікаво, бо тепер ваша гілка testing пересунулась уперед, а ваша гілка master досі вказує на фіксацію, що й у момент виконання `git checkout` для переключення гілок. Переключимося назад до гілки master:

```
$ git checkout master
```



ЗОБРАЖЕННЯ 3-8

HEAD пересувається, коли ви отримуєте (checkout)

Ця команда зробила дві речі. Вона пересунула вказівник HEAD назад на гілку master, та повернула файли у вашій робочій теці до стану знімку, на який вказує master. Це також означає, що якщо ви зараз зробите нові зміни, вони будуть походити від ранішої версії проекту. Вона, суттєво, перемотує працю, що ви зробили у гілці testing, щоб ви могли працювати в іншому напрямку.

> ПЕРЕКЛЮЧЕННЯ МІЖ ГІЛКАМИ ЗМІНЮЄ ФАЙЛИ У ВАШІЙ РОБОЧІЙ ТЕЦІ
>
> Важливо зауважити, що коли ви переключаєте гілки в Git, файли у вашій робочій теці змінюються. Якщо ви переключаєтесь до старшої гілки, ваша робоча тека буде повернута до того стаку, який був на момент останнього фіксування у тій гілці. Якщо Git не може зробити це без проблем, він не дасть вам переключитися взагалі.

Зробимо декілька змін та знову зафіксуємо:

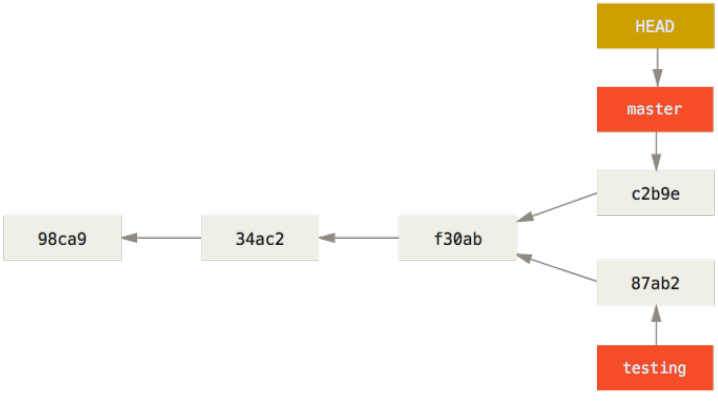
```
$ vim test.rb
$ git commit -a -m 'Зробив інші зміни'
```

Тепер історія вашого проекту розбіглася (diverged) (дивіться **Figure 3-9**). Ви створили гілку, дещо в ній зробили, переключились на головну гілку та зробили там щось інше. Обидві зміни ізольовані в окремих гілках. Ви можете переключатись між цими гілками та злити

їх, коли вони будуть готові. І все це ви зробили за допомогою простих команд `branch`, `checkout` та `commit`.

ЗОБРАЖЕННЯ 3-9

Історія, що розбіглася



Ви також можете легко це побачити за допомогою команди `git log`. Якщо ви виконаєте `git log --oneline --decorate --graph --all`, вона надрукує історію ваших фіксацій, покаже куди вказують ваші гілки та як розбіглася ваша історія.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) Зробив інші зміни
| * 87ab2 (testing) Зробив зміни
|/
* 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
```

Оскільки гілки в Git це насправді простий файл, що містить 50 символів контрольної суми SHA-1 фіксації, на яку вказує, гілки дешево створювати та знищувати. Створити гілку так же швидко, як записати 41 байт до файлу (40 символів та символ нового рядку).

Це вражаюча відмінність від того, як більшість інших СВК працюють з гілками – зазвичай це потребує копіювання усіх файлів проекту в другу теку. Це може зайняти декілька секунда, або навіть хвилин, в

залежності від розміру проекту, у той час як у Git процес завжди миттєвий. Також, оскільки ми записуємо батьків кожної фіксації, пошук відповідної бази для злиття може бути зроблено автоматично та зазвичай дуже просто. Ці можливості допомагають заохотити розробників створювати та використовувати гілки часто.

Подивимось, чому і вам варто так робити.

Основи галуження та зливання

Розглянемо простий приклад галуження та зливання на схемі, котра трапляється в реальності:

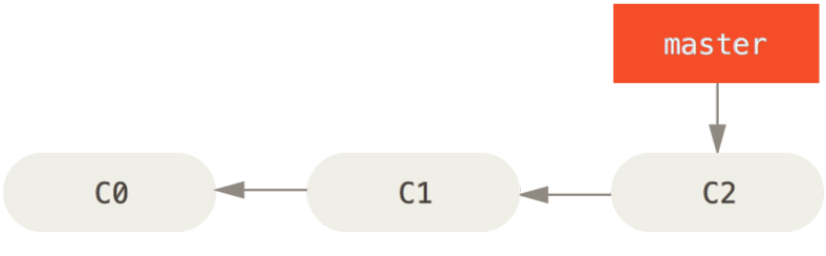
1. Вам потрібно внести зміни до веб сайту.
2. Створюєте гілку для своєї задачі.
3. Працюєте в цій гілці.

В якийсь момент вам дзвонять і кажуть, що є більш важлива задача та потрібно термінове виправлення. Ви зробите таке:

1. Переключитеся на головну гілку.
2. Створите гілку-виправлення.
3. Після тестування зливаєте гілку-виправлення та надсилаєте в основну гілку.
4. Переключаєтеся на першопочаткову гілку та продовжуєте роботу над задачею.

Основи галуження

Скажімо, ви працюєте над проектом і вже маєте кілька комітів.



ЗОБРАЖЕННЯ 3-10

A simple commit history

Тепер вирішили працювати над задачею, котра в системі вашої компанії зареєстрована як №53. Щоб створити гілку для цієї задачі та одразу перейти на неї, виконайте команду `git checkout` з параметром `-b`:

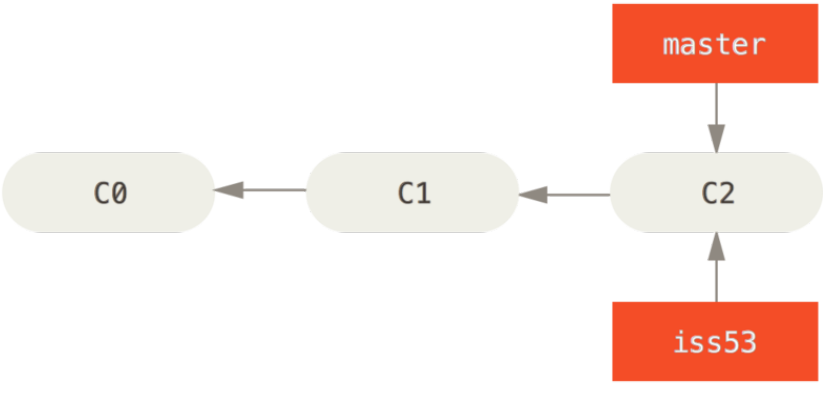
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Це скорочений запис наступного:

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

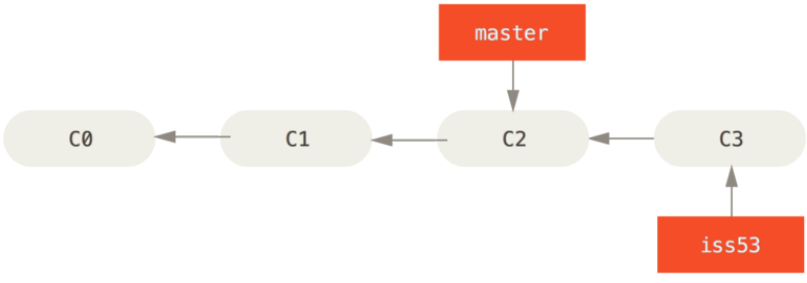
ЗОБРАЖЕННЯ 3-11

Створення нового вказівника гілки



Ви працюєте над змінами до сайту та комітите зміни. Таким чином ваша гілка `iss53` починає рухається вперед, оскільки ви на неї раніше переключилися (тобто вказівник `HEAD` вказує на цю гілку):

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```



ЗОБРАЖЕННЯ 3-12

Гілка iss53 рухається

Вам подзвонили та доповіли про проблему на сайті, якої потрібно якнайшвидше позбутися. Завдяки Git, вам не потрібно відсилати це виправлення разом із змінами в `iss53`, також, ви не докладете багато зусиль для того щоб скасувати поточні зміни та працювати над виправленням в основній гілці. Все що потрібно, це знову переключитися на основну гілку `master`.

Проте, зверніть увагу на те, що якщо у вашій робочій директорії чи області з підготовленими файлами є незакомічені зміни, це спричинить конфлікт з гілкою `master` та Git не дозволить зробити це переключення. Найкраще спочатку очистити вашу робочу область перед переключеннями.

Способи як це зробити (сховати (stash) та виправити (commit amend)) ми розглянемо пізніше в **“Ховання та чищення”**. Зараз вважаємо, що ми закомітили всі зміни в `iss53`, отже, можна перейти на гілку `master`:

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

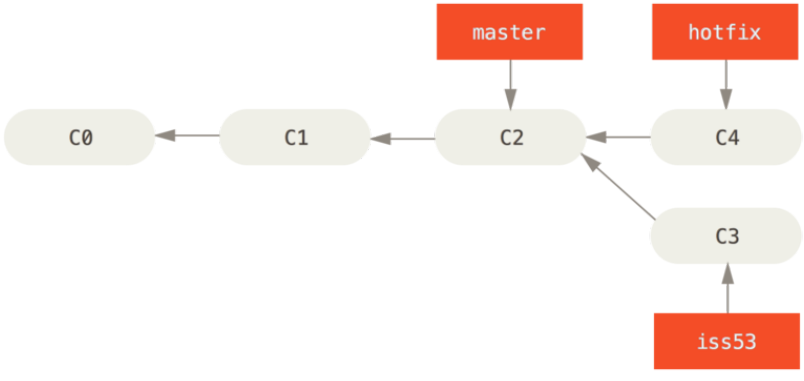
Тепер ваша робоча директорія точно в такому стані, як була до того, як ви почали працювати над №53 і ви можете сфокусуватися на терміновому виправленні. Важливо запам’ятати: коли перемикаєтеся між гілками, Git відновлює вашу робочу директорію, щоб вона виглядала так як після вашого останнього коміту. Git додає, видаляє та змінює файли автоматично, щоб впевнитися що ваша робоча копія точно відповідає тому якою була гілка на час вашого останнього коміту.

Далі вам знову потрібно зробити ще одне швидке виправлення. Створимо гілку `hotfix` і будемо там працювати поки не закінчимо:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

ЗОБРАЖЕННЯ 3-13

Гілка `hotfix`, базована на `master`

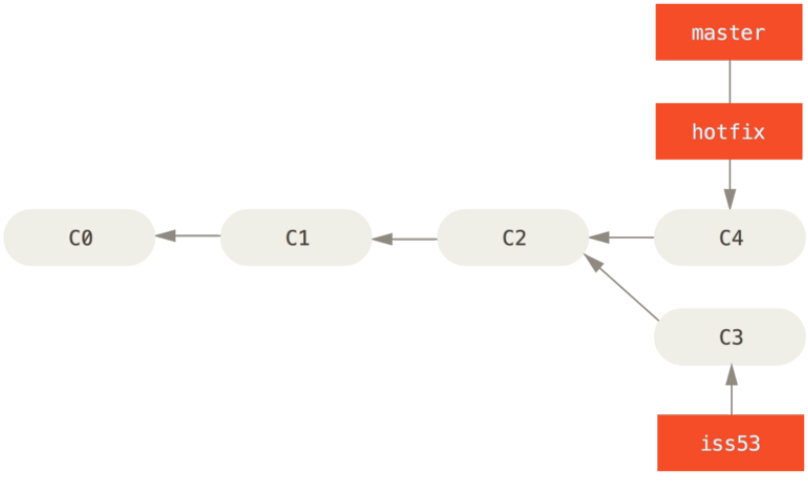


Тепер можете запускати тести, щоб впевнитися що зміна годиться і злити (merge) гілку назад у master щоб викласти зміни на виробництво. Робиться це за допомогою команди `git merge` command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Зверніть увагу на фразу “fast-forward” у цьому злитті. Через те, що коміт, який зливався, належав гілці, що була безпосереднім нащадком гілки куди зливалося (гілка `master` вверх по течії відносно `hotfix`), Git просто переміщує вказівник вперед. Іншими словами, коли ви зливаєте один коміт з іншим, і це можна досягнути слідуючи історії першого коміту, Git просто переставляє вказівник, оскільки немає змін-відмінностей, які потрібно зливати разом - це називається “перемоткою” (“fast-forward”).

Тепер ваша зміна міститься в знімку коміту, на який вказує `master` і ви можете викладати зміни.



ЗОБРАЖЕННЯ 3-14

`master` перемотаний на `hotfix`

Після того, як ваше супер важливе виправлення викладено, можна повернутися до роботи, яку було відкладено через швидке виправлення. Але спочатку видалимо гілку `hotfix` - нам її більше не потрібно, а `master` вказує на той самий знімок коду. Для цього виконайте `git branch` з опцією `-d`:

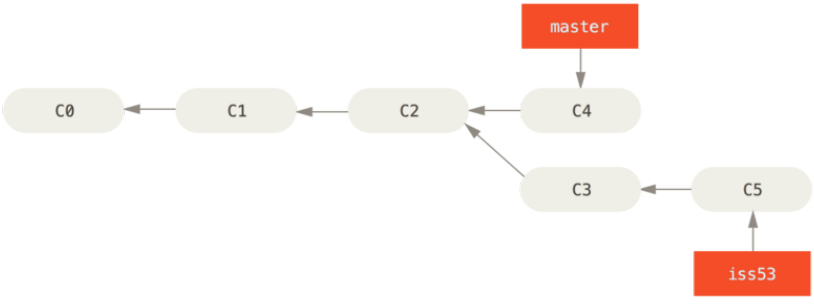
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Тепер перемикайтеся на вашу незакінчену гілку для №53 і продовжуйте роботу.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

ЗОБРАЖЕННЯ 3-15

Продовження роботи на `iss53`



Зауважте, що зміни з гілки `hotfix` відсутні в гілці `iss53`. Якщо вам потрібні ці зміни підчас роботи над №53, ви можете злити `master` з `iss53` командою `git merge master`, або просто почекати до того моменту коли ви будете інтегровувати `iss53` в `master`.

Основи зливання

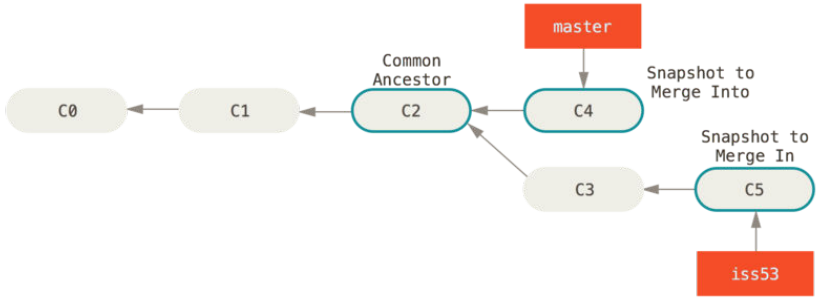
Допустимо, ви вирішили, що робота над №53 завершена і готова до злиття з гілкою `master`. Для цього ви виконаєте злиття гілки `iss53` точно так, як раніше робили це з гілкою `master`. Все що потрібно це перемкнутися на вашу робочу гілку і виконати команду `git merge`:

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

Виглядає трошки інакше, ніж те, що ми робили з гілкою `hotfix`. У цьому випадку історія змін двох гілок почала відрізнятися в якийсь момент. Оскільки коміт поточної гілки не є прямим нащадком гілки, в яку ви зливаєте зміни, Git мусить трохи попрацювати. В цьому випадку Git робить просте трьох-точкове злиття, користуючись двома знімками, що вказують на гілки та третім знімком - їх спільним нащадком.

ЗОБРАЖЕННЯ 3-16

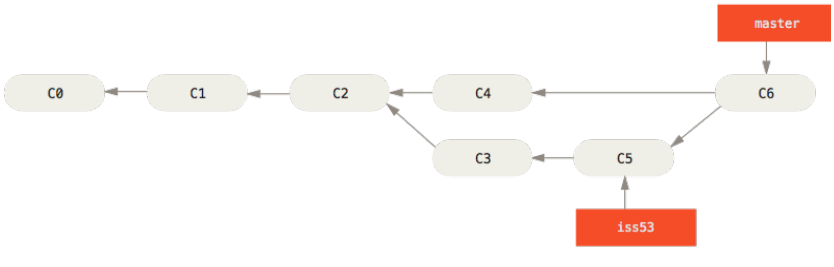
Три відбитки типового злиття



Замість того, щоб просто пересунути вказівник гілки вперед, Git створює новий знімок, що є результатом 3-точкового злиття, і автоматично створює новий коміт, що вказує на нього. Його називають комітом злиття (merge commit) та його особливістю є те, що він має більше одного батьківського коміту.

ЗОБРАЖЕННЯ 3-17

Коміт злиття



Варто зауважити, що Git сам визначає найбільш підходящого спільного нащадка, якого брати за основу зливання; це відрізняє Git від старіших систем таких як CVS чи Subversion (до версії 1.5), де розробник, що виконує зливання, сам повинен вказувати що брати за основу зливання. Це надзвичайно полегшує зливання, порівняно з іншими системами.

Тепер, коли ваші зміни злиті, гілка `iss53` вам більше не потрібна. Можете закривати задачу №53 та видаляти гілку:

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

Основи конфліктів зливання

Трапляється, що цей процес не проходить гладко. Якщо ви маєте зміни в одному й тому самому місці в двох різних гілках, Git не зможе їх просто злити. Якщо підчас роботи над №53 ви поміняли ту саму частину файлу, що й в `hotfix`, ви отримаєте конфлікт, що виглядає приблизно так:

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

У цьому випадку Git не створив автоматичний коміт зливання. Він призупинив процес допоки ви не вирішите конфлікт. Для того, щоб переглянути знову які саме файли спричинили конфлікт, виконайте `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")
```

Все, що має конфлікти, які не були вирішені є в списку незлитих (unmerged) файлів. У кожен такий файл Git додає стандартні позначки-вирішенння для конфліктів, отже ви можете відкрити ці файли і вирішити конфлікти самостійно. У вашому файлі з конфліктом появиться блок, схожий на таке:

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

```
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Розглянемо, як це розуміти. Версія файлу в HEAD (з вашої master гілки, оскільки ви запустили зливання, будучи на ній) у верхній частині блоку (все вище =======), а версія з iss53 - все, що нижче. Щоб розв'язати цю несумісність, вам потрібно вибрати одну із версій, або самостійно (вручну) поредагувати вміст файлу. Наприклад, ви можете вирішити цей конфілікт, замінивши блок повністю:

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

В цьому випадку ми взяли потрохи з кожної секції, а стрічки <<<<<<<, ======= та >>>>>>> видалили повністю. Після того, як ви розв*язали подібні несумісності в кожному блоці конфліктних файлів, виконайте для них git add, щоб індексувати та позначити, як розв*язані. Індексуючи файл, ви позначаєте його для Git таким, що більше не має конфлікту. Якщо ви хочете використовувати графічний інструмент для розв'язання конфліктів, виконайте команду git mergetool, яка запустить графічний редактор та проведе вас по всьому процесу:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge
Merging:
index.html

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

Для того, щоб використовувати інструмент-програму іншу, ніж по-замовчуванню (Git обрав opendiff, оскільки команду було запущено з Mac), подивіться на список сумісних зверху одразу після "one of the following tools." Просто введіть ім'я потрібного інструменту.

Про інструменти для розв'язання більш складних конфліктів ми повернемося в **"Складне злиття"**.

Після того, як ви вийшли з програми для зливання, Git спитає вас чи було зливання успішним. Якщо ви відповісте, що так, Git проіндексує файл для того, щоб позначити файл як безконфліктний Можете виконати `git status` знову, щоб перевірити чи всі конфлікти розв'язані:

```
$ 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
```

Якщо ви задоволені результатом та перевірили, що всі файли, котрі містили несумісності, проіндексовані, можете виконувати `git commit` і, таким чином, завершувати злиття. Повідомлення після коміту виглядає приблизно так:

```
Merge branch 'iss53'

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


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

Тепер можете змінити повідомлення, додаючи деталі про те, як ви вирішували конфлікт, якщо вважаєте, що це може допомогти іншим зрозуміти ваше зливання в майбутньому - чому ви це зробили, якщо це не очевидно.

Управління гілками

Тепер, коли ви вже вмієте створювати гілки, зливати їх та видаляти, розгляньмо те, як ними управляти, та на інструменти, які можуть в цьому допомогти.

Команда `git branch` насправді вміє більше ніж просто створювати та знищувати гілки. Запустіть її без параметрів і ви побачите просто список ваших гілок:

```
$ git branch
  iss53
* master
  testing
```

Зверніть увагу на символ `*` перед `master`: це вказівник на вашу поточно вибрану гілку (тобто ту, на котру вказує `HEAD`). Це означає, що якщо ви зараз захочете зробити коміт, `master` оновиться вашими новими змінами. Щоб побачити ваші останні коміти - запустіть `git branch -v`:

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

Опції `--merged` та `--no-merged` корисні для фільтрування списку гілок залежно від того чи вони були злиті з поточною гілкою. Для списку гілок, що були злиті з поточною гілкою виконайте `git branch --merged`:

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

Ви бачите `iss53` в цьому списку тому, що раніше її злили з `master`. Взагалі, гілки без `*` із цього списку можна вже видаляти (за допомогою `git branch -d`), адже ми вже інтегрували ті зміни, тому не втратимо їх.

Команда `git branch --no-merged` покаже гілки, які ви не зливали з поточною гілкою:

```
$ git branch --no-merged
  testing
```

Тут ви бачите свою іншу гілку. Оскільки дана гілка містить роботу, що не зливалася, спроба видалити її за допомогою `git branch -d` не буде успішною:

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

Якщо ж ви дійсно впевнені в тому, що гілка вам не потрібна і всі зміни з неї можна втрачати, можна змусити Git це зробити за допомогою параметра `-D`. Про що Git вам і повідомляв з попередньої спроби.

Процеси роботи з гілками

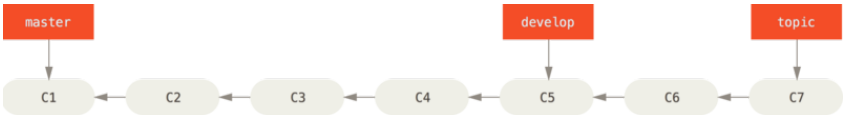
Тепер, коли ви навчилися основам роботи з гілками, що ж з ними можна чи потрібно робити далі? В цьому розділі ми розкажемо про найбільш вживані підходи до роботи з гілками, які дозволяє така легка система галуження Git, а ви можете вирішити чи втілювати вам їх у свій робочий цикл.

Довго-триваючі гілки

Оскільки Git використовує просте трьох-точкове злиття, то зливання одної гілки в іншу протягом тривалого часу зазвичай не є складною задачею. Це означає, що ви можете мати кілька активних гілок та використовувати їх для різних стадій вашого робочого циклу; можете періодично зливати з одної гілки в іншу.

Багато розробників підтримують з Git такий процес, коли тільки в `master` є стабільна версія коду - найімовірніше того, що був чи буде запроваджений у виробництво. Також вони мають паралельні гілки `develop` чи `next`, які використовуються для тестування стабільності - це не обов'язково постійно стабільні гілки, але, як тільки вони стабілізовуються, їх можна зливати з `master`. Також практикується мати тематичні гілки (коротко-термінові, як `iss53`, що ви створили раніше) та зливати їх, коли вони готові, проходять всі тести, та не привнесуть нових помилок.

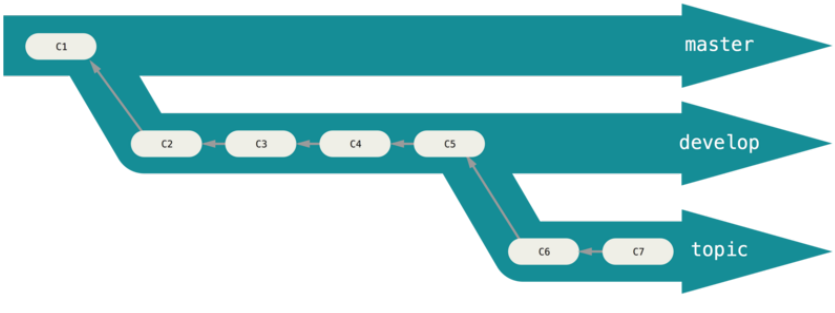
Насправді, ми маємо справу з вказівниками, що рухаються вздовж лінії комітів, які ви додаєте. Стабільні гілки нижче по лінії ваших комітів, а "кровоточащі" гілки, ті що містять нові зміни, випереджують їх.



ЗОБРАЖЕННЯ 3-18

Лінійне представлення прогресування стабільності в гілках

Взагалі простіше собі уявити цей процес як елеватор, коли набори комітів прямують до більш стабільних гілок, де вони повністю тестуються.



ЗОБРАЖЕННЯ 3-19

"Елеваторне" представлення прогресування стабільності в гілках

Ви можете мати декілька рівнів стабільності. На більших проектах також практикують мати `proposed` чи `pu` (proposed updates) гілки, де зберігаються гілки, що вже інтегровані, але не готові бути перенесені в

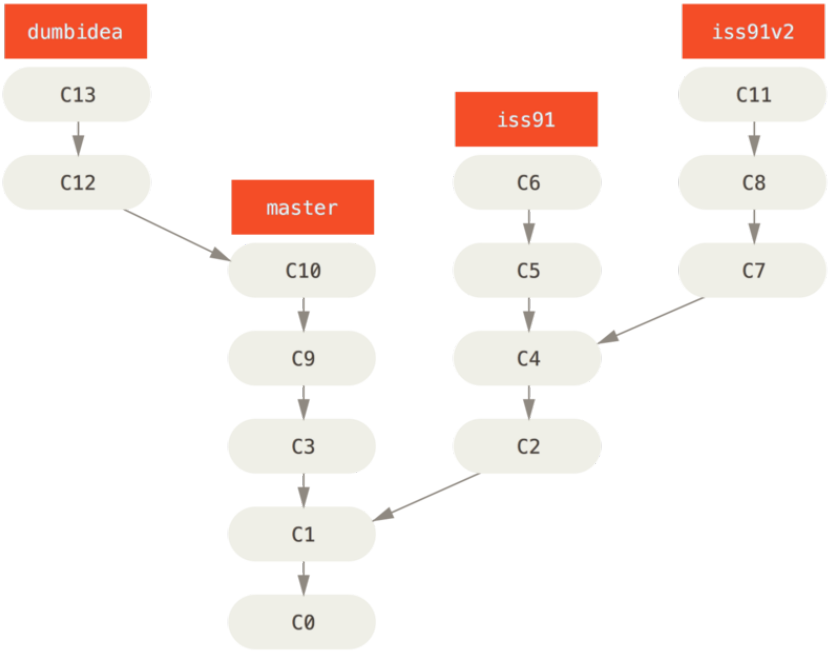
основні гілки next чи master. Ідея таких гілок в тому, що ваші гілки мають кілька рівнів стабільності і коли вони досягають більш стабільного рівня, їх зливають до гілок, котрі вище над ними. Зверніть увагу, що мати кілька довготривалих гілок не обов'язково, проте буває корисним, особливо коли маєте справу з великими чи складними проектами.

Тематичні гілки

Тематичні гілки мають своє застосування в проектах будь-якого розміру. Тематична гілка - це коротко-термінова гілка, що створюється для окремо взятого завдання чи функціональності. Напевно, ви таке раніше не робили часто з іншими СКВ, оскільки загалом це досить "дорого" створювати та зливати гілки. Проте в Git це досить звична практика створювати, додавати коміти, зливати та видаляти гілки по кілька разів на день.

Приклад цього ви бачили раніше в попередньому розділі з гілками iss53 та hotfix Ви додавали кілька комітів в ці гілки та видаляли їх одразу після зливання в основну гілку. Така техніка дозволяє швидко та повністю змінювати контекст роботи, оскільки ваша робота в окремому елеваторі, де всі зміни пов'язані тільки з однією темою чи завданням, це полегшує та ізолює перегляд коду тощо. Ви можете тримати свої зміни там кілька хвилин, днів, чи місяців, і зливати аж тоді, коли вони будуть готові.

Розглянемо приклад, коли нам потрібно зробити завдання №91 (в master), ми робимо гілку (iss91), трохи там працюємо, потім робимо з неї ще одну гілку (iss91v2) щоб попробувати інший підхід, повертаємося в master на деякий час, і тоді робимо ще одну гілку щоб перевірити одну непевну ідею (гілка dumbidea). Історія комітів виглядатиме десь так:



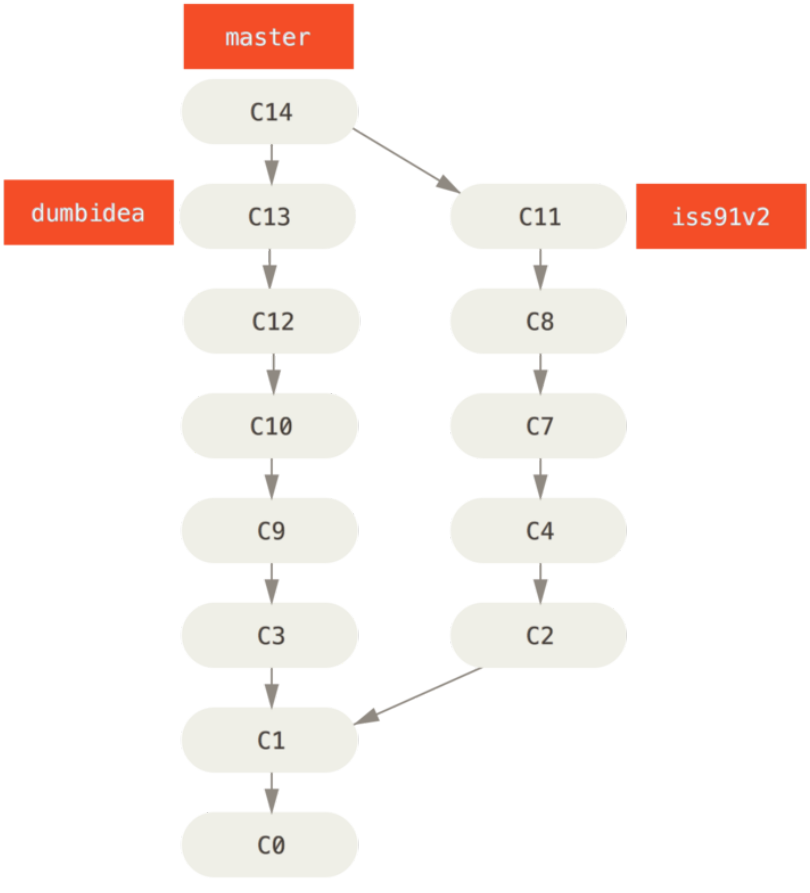
ЗОБРАЖЕННЯ 3-20

Багато тематичних гілок

Тепер, скажімо, вам сподобався підхід номер два (`iss91v2`) і ви показали свою ідею (гілку `dumbidea`) колегам, і вона їм здається геніальною. Ви можете викидати оригінальну `iss91` (втрачаючи коміти `C5` та `C6`) та зливати дві інші гілки. Тепер ваша історія така:

ЗОБРАЖЕННЯ 3-21

Історія після зливання dumbidea та iss91v2



Ми приділимо більше уваги різним можливим робочим процесам ваших Git проектів в **Chapter 5**, тому, перед остаточним рішенням яку ж схему обрати, прочитайте цей розділ.

Пам'ятайте, що всі зміни, що ви робили тут з гілками - повністю локальні. Коли ви створюєте гілки та зливаєте їх - все це відбувається тільки у вашому сховищі-репозиторію, - ніяких змін на сервер чи з нього не надсилається.

Віддалені гілки

Віддалені посилання — це посилання (вказівники) у ваших віддалених сховищах: гілки, теґи тощо. Для повного списку віддалених посилань

виконайте `git ls-remote (remote)`, або `git remote show (remote)` для детальної інформації про віддалені гілки. Проте, найпоширеніше застосування — це віддалено-відслідковувані гілки.

Віддалено-відслідковувані гілки — це вказівники на стан віддалених гілок. Локально ці вказівники не можливо змінити, але вони автоматично рухаються, коли ви виконуєте мережеві операції. Віддалено-відслідковувані гілки слугують свого роду закладками, що нагадують вам про стан віддалених репозиторіїв на момент вашого останнього зв'язку з ними.

Вони мають наступний запис: `(remote)/(branch)`. Наприклад, якщо ви хочете побачити як виглядала гілка `master` з віддаленого сховища `origin`, коли ви востаннє зв'язувалися з ним, перейдіть на гілку `origin/master`. Припустимо, ви працювали з колегами над одним завданням і вони вже виклали свої зміни. У вас може бути своя локальна гілка `iss53`, але на сервері гілка вказуватиме на коміт `origin/iss53`.

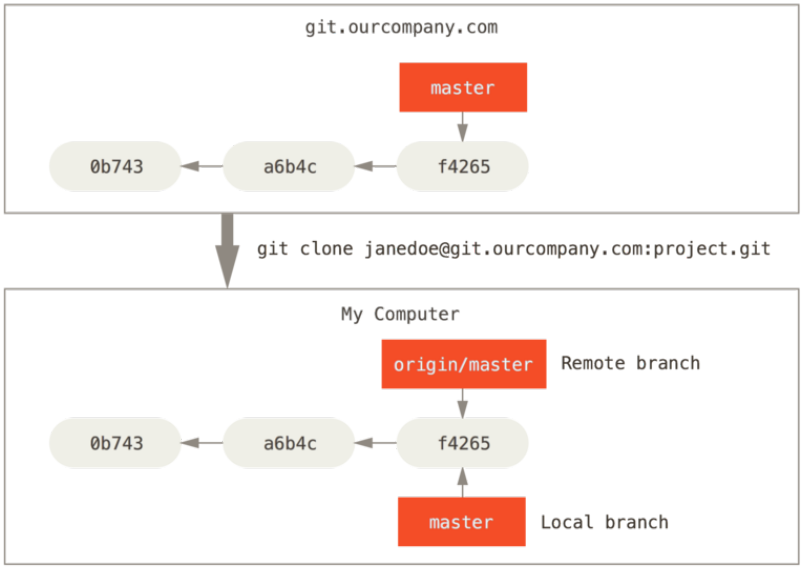
Це може трохи спантеличувати, розгляньмо приклад. Скажімо, ви працюєте з Git сервером, що доступний у вашій мережі за адресою `git.ourcompany.com`. Коли ви склонуєте з нього, команда `clone` автоматично іменує його `origin`, стягує всі дані, створює вказівник на те місце, де зараз знаходиться `master` і локально іменує це посилання `origin/master`, щоб ви могли з чогось почати працювати.

"ORIGIN" НЕ Є ОСОБЛИВИМ

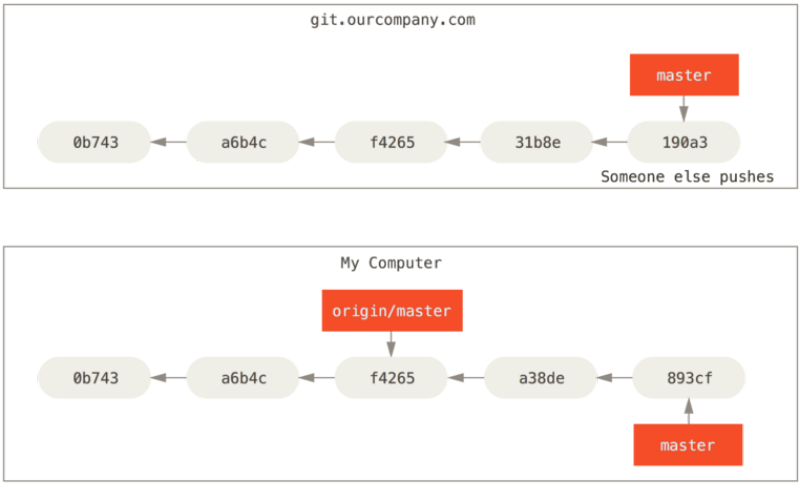
Так само, як назва гілки "master" не має якогось особливого значення для Git, так само й "origin". Просто "master" дається за-замовчуванням для початкової гілки, коли ви запускаєте `git init` — ось чому воно так часто зустрічається, а "origin" — це ім'я за-замовчуванням для віддалених посилань команди `git clone`. Якщо ж ви, натомість, виконаєте `git clone -o booyah`, то отримаєте `booyah/master` своєю гілкою за-замовчуванням.

ЗОБРАЖЕННЯ 3-22

Серверний та локальний репозиторії після клонування



Якщо ви виконали якусь роботу на локальній гілці `master`, і водночас, хтось виклав зміни на `git.ourcompany.com` в `master`, тоді ваші історії прогресують по-різному. Доки ви не синхронізуетесь з сервером, вказівник `origin/master` не буде рухатись.



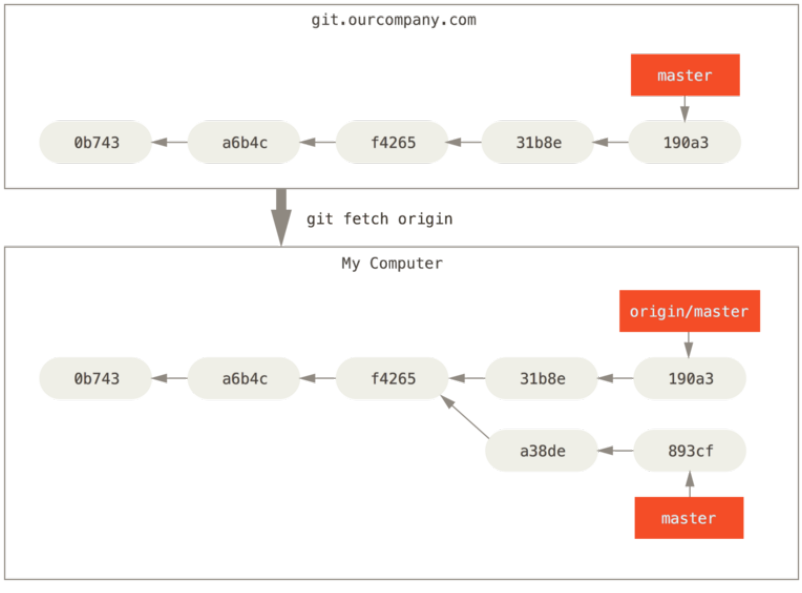
ЗОБРАЖЕННЯ 3-23

Локальна та віддалена робота розійшлися

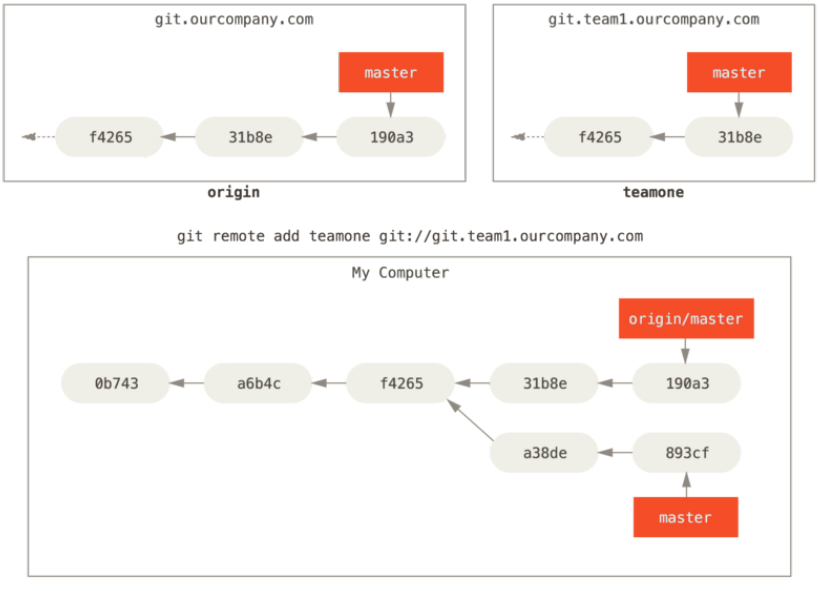
Щоб відновити синхронність, виконайте команду `git fetch origin`. Ця команда шукає який сервер відповідає імені “origin” (у нашому випадку `git.ourcompany.com`), отримує дані, яких ви ще не маєте і оновлює вашу локальну базу даних, переміщаючи вказівник `origin/master` на нову, більш актуальну, позицію.

ЗОБРАЖЕННЯ 3-24

git fetch оновлює віддалені посилання



Щоб продемонструвати роботу з кількома віддаленими серверами і як виглядають віддалені гілки для віддалених проектів, уявімо, що ви маєте ще один внутрішній Git сервер, котрий використовує лиш одна з ваших спринт команд. Сервер розташований за адресою `git.team1.ourcompany.com`. Ви можете додати його як нове віддалене посилання до вашого поточного проекту за допомогою команди `git remote add`, як ми розповідали в **Chapter 2**. Дайте йому ім'я `teamone`, і це буде вашим скороченням для повного URL.



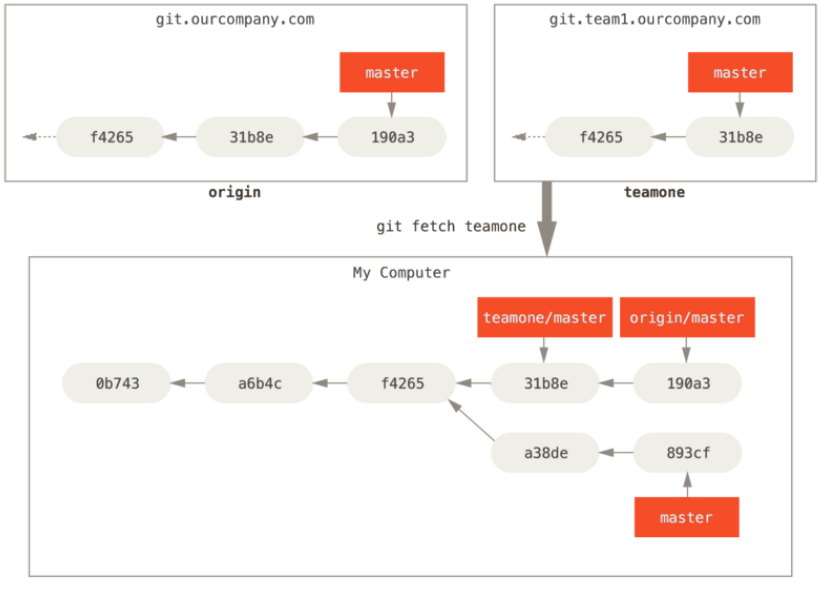
ЗОБРАЖЕННЯ 3-25

Додавання нового віддаленого сервера

Тепер виконайте `git fetch teamone` щоб витягнути з `teamone` всі оновлення. Оскільки `teamone` на даний момент є підмножиною `origin`, то Git не отримує нових даних і нічого не оновлює, а просто ставить віддалено-відслідковувану гілку `teamone/master` вказувати на коміт, на котрому зараз знаходиться гілка `master` для сервера `origin`.

ЗОБРАЖЕННЯ 3-26

Віддалено-відслідковувана гілка `teamone/master`



Надсилання

Коли ви хочете поділитися гілкою з навколишнім світом, надішліть (push) її на віддалене посилання, до якого маєте право запису. Ваші локальні гілки не синхронізуються з віддаленими автоматично — потрібно явно надсилати гілки, якими хочете поділитися. Таким чином, можете користуватися приватними гілками для роботи, якою не збираєтеся ділитися, а надсилати зміни тільки в тематичні гілки, над якими співпрацюєте.

Щоб поділитися з іншими своєю гілкою з назвою `serverfix`, надсилайте її так само як це робили із першою гілкою. Виконайте `git push (remote) (branch)`:

```
$ git push origin serverfix
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]      serverfix -> serverfix
```

Це трохи скорочено. Git автоматично замінює назву гілки `serverfix` з `refs/heads/serverfix:refs/heads/serverfix`, що означає "Візьми мою локальну гілку serverfix та надішли її для оновлення віддаленої гілки serverfix." Ми детальніше розглянемо `refs/heads/` в **Chapter 10**, але, взагалі, можете зараз це пропустити. Також ви можете виконати `git push origin serverfix:serverfix`, що зробить те саме, тобто скаже "Бери мій serverfix та шли на віддалений serverfix." Користуйтеся таким підходом щоб надсилати локальні гілки, що називаються інакше ніж віддалені. Якщо ви не хочете мати віддалену назву `serverfix`, виконайте натомість `git push origin serverfix:awesomebranch`, щоб надіслати локальну `serverfix` на віддалену з назвою `awesome-branch`.

НЕ ВВОДЬТЕ СВІЙ ПАРОЛЬ ЩОРАЗУ

Якщо ви користуєтеся HTTPS для надсилання змін, Git сервер запитуватиме ваші ім'я користувача та пароль для аутентифікації. За замовчуванням Git питатиме вас цю інформацію в терміналі для того, щоб перевірити правомірність надсилання змін.

Щоб не вводити це щоразу, налаштуйте "credential cache". Найпростіше — дозволити тримати цю інформацію в пам'яті протягом кількох хвилин, для чого виконайте `git config --global credential.helper cache`.

Більше інформації про способи кешування ідентифікації користувачів у секції **"Збереження посвідчення (credential)"**.

Наступного разу, коли співпрацівники будуть отримувати зміни з сервера, — отримають вказівник на поточний стан серверного `server-fix` у віддалену гілку-посилання `origin/serverfix`:

```
$ 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]      serverfix    -> origin/serverfix
```

Варто зауважити, що, коли ви отримуєте зміни (за допомогою `fetch`), — насправді це нові віддалено-відслідковувані гілки, а не автоматичні локальні копії цих гілок, які ви можете редагувати. Іншими словами, в цьому випадку, ви не маєте нової гілки `serverfix`, а просто вказівку на `origin/serverfix`, яку не можете змінювати.

Щоб злити ці отримані зміни в вашу поточну робочу гілку виконайте `git merge origin/serverfix`. Якщо хочете мати свою власну гілку `serverfix` і працювати над нею, можете створити її, базуючись на віддалено-відслідковуваній гілці:

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

За допомогою цього отримаєте гілку, де ви можете працювати, та котра відображає стан `origin/serverfix`.

Відслідковувані гілки

Перемикання на локальну гілку з віддалено-відслідковуваної автоматично створює так звану відслідковувану гілку “tracking branch” (або “upstream branch”). Відслідковувані гілки — це локальні гілки, що мають безпосередній зв’язок з віддаленою гілкою. Якщо ви знаходитеся на відслідковуваній гілці і потягнете зміни, виконуючи `git pull`, Git відразу знатиме з якого сервера брати та з якої гілки зливати зміни.

Коли ви клонуєте репозиторій, Git автоматично створює гілку `master`, яка слідкує за `origin/master` Проте, ви можете налаштувати й інші відслідковувані гілки – такі, що слідкують за іншими віддаленими посиланнями, або за гілкою, відмінною від `master`. Як у випадку, що ви бачили в прикладі, виконуючи `git checkout -b [branch] [remote-name]/[branch]`. Це досить поширена дія, Git має опцію `--track` для скороченого запису:

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

Щоб дати локальній гілці назву, що відрізняється від серверної, виконайте попередню повну команду, вказуючи бажане ім’я гілки:

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

Тепер локальна гілка `sf` буде автоматично витягувати зміни з `origin/serverfix`.

Якщо ж у вас вже є локальна гілка і ви хочете прив'язати її до віддаленої, чи змінити віддалену (upstream) гілку, можете використовувати опції `-u` чи `--set-upstream-to` до команди `git branch`.

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

СКОРОЧЕНЕ ЗВЕРТАННЯ ДО **UPSTREAM**

Для того, щоб звертатися до відслідковуваної гілки, використовуйте скорочений запис `@{upstream}` чи `@{u}`. Тобто, при бажанні, знаходячись на `master`, що слідкує за `origin/master`, користуйтеся чимось на зразок `git merge @{u}` замість повного `git merge origin/master`.

Опція `-vv` до `git branch` дозволяє дізнатися, які у вас налаштовані відслідковувані гілки. Результатом буде список локальних гілок та інформація про них, включаючи, які гілки відслідковуються та деталі про те, чи вони випереджають чи відстають від локальних (чи те й інше).

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Тут ми бачимо, що гілка `iss53` слідкує за `origin/iss53` та випереджає її ("ahead") на два, тобто ми маємо локально два коміти, які ще не надіслані на сервер. Також ми бачимо, що `master` слідкує за `origin/master` та її стан є актуальним. Далі бачимо, що `serverfix` слідкує за гілкою `server-fix-good` з сервера `teamone` та випереджає його на три й відстає на один, тобто існує один коміт на сервері, який ми ще не злили та три коміти локально, які ми ще не надіслали. Насамкінець бачимо, що локальна гілка `testing` не слідкує за жодною віддаленою.

Варто зауважити, що ці числа відображають стан віддалених гілок на час останнього оновлення з кожного сервера. Сама по собі команда не сягає серверів, вона просто відображає те, що збережено

про ці сервери локально. Якщо ж ви хочете отримати найостаннішу інформацію про випередження чи відставання гілок, оновіть спочатку всі віддалені посилання. Можете це зробити ось як: `$ git fetch --all; git branch -vv`

Витягування змін

Команда `git fetch`, під час виконання, отримує всі оновлення, яких ви ще не маєте, але, зовсім не змінює вашу робочу директорію. Вона просто отримує дані для того, щоб ви могли самотужки злити зміни. Існує команда `git pull`, яка, по своїй суті та в більшості випадків, є послідовним виконанням команд `git fetch` та `git merge`. Якщо у вас є відслідковувана гілка, як ми показували у попередній секції, — створена та самостійно налаштована, чи як результат `clone` чи `checkout`, команда `git pull` буде звертатися до відслідковуваних сервера та віддаленої гілки, отримувати оновлення і тоді робити спробу зливання.

Переважно простіше користуватися `fetch` та `merge` явно, оскільки магічний `git pull` часом може збивати з пантелику.

Видаляння віддалених гілок

Уявіть, що ви закінчили з віддаленою гілкою — тобто всі співпрацівники завершили свій вклад у нову функціональність та гілку було злито з віддаленою `master` (чи яка там у вас стабільна лінія коду). Для видалення віддаленої гілки користуйтеся опцією `--delete` до `git push`. Щоб видалити вашу `serverfix` виконайте таке:

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

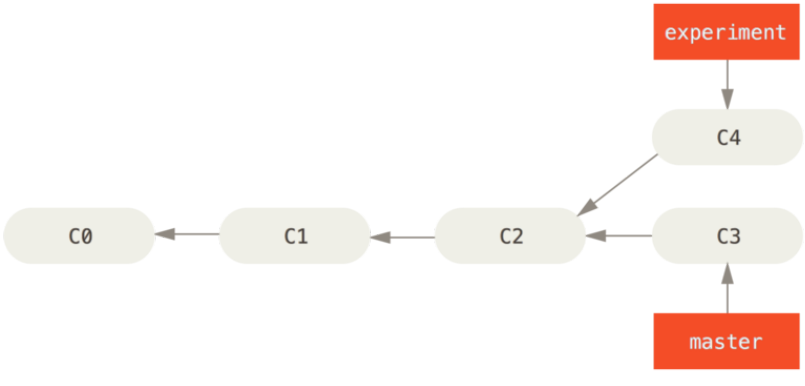
Все, що відбулося, це видалення вказівника на сервері. Git сервер триматиме самі дані ще деякий час, до наступного збирання сміття (garbage collection), тому, якщо щось було видалено випадково, часто це досить легко відновити.

Перебазовування

Git має два основні способи інтегрувати зміни з гілки в гілку: `merge` (зливання) та `rebase`(перебазовування). У цій секції вивчатимете що таке перебазовування, як його виконувати, чому воно вважається таким чудовим інструментом та випадки, коли його не варто застосовувати.

Просте перебазовування

Поверніться до попереднього прикладу з “Основи зливання”, де видно, що ваша робота розгалузилася в комітах по двох різних гілках.



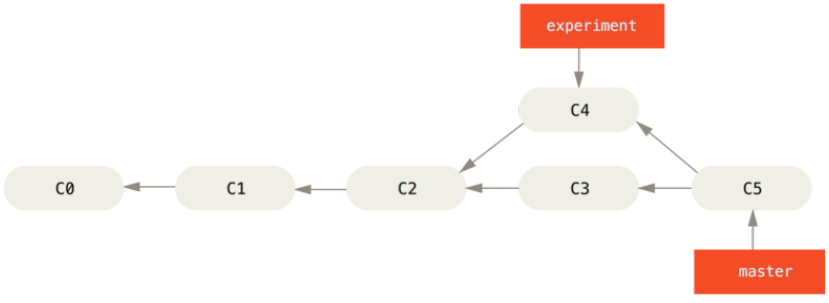
ЗОБРАЖЕННЯ 3-27

Просте розгалуження історії

Найпростіше інтегрувати гілки, як ми вже розглянули, за допомогою команди `merge`. Вона виконає трьох-точкове зливання між двома останніми знімками гілок (`C3` і `C4`) та їх найближчим спільним предком (`C2`), створивши новий знімок (і коміт).

ЗОБРАЖЕННЯ 3-28

Інтеграція розгалуженої історії за допомогою зливання



Проте, існує інший спосіб: можете взяти латку (patch) змін з C4 і накласти її поверху C3. У Git це називають *перебазовуванням* (rebasing). За допомогою `rebase`, ви можете взяти всі зміни, що були в комітах одної гілки та відтворити їх на іншій гілці.

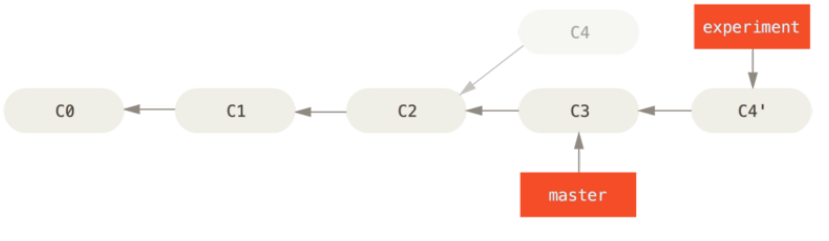
Для цього прикладу, виконайте наступне:

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

Працює це так: шукається спільний предок двох гілок (поточної та тої, на котру перебазовуєтесь); отримуються відмінності (diff), спричинені кожним комітом поточної гілки; відмінності зберігаються в тимчасові файли; поточна гілка встановлюється (reset) на коміт гілки, на яку перебазовуєтесь; та нарешті, застосовується кожна зміна.

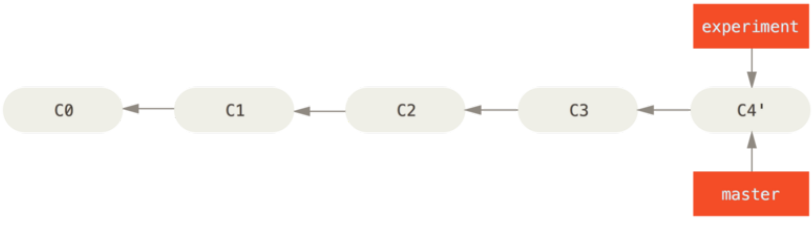
ЗОБРАЖЕННЯ 3-29

Перебазовування зміни з C4 на C3



Тепер ви можете повернутися на `master` та зробити злиття перемотуванням (fast-forward merge).

```
$ git checkout master
$ git merge experiment
```



ЗОБРАЖЕННЯ 3-30

Злиття перемотуванням гілки master

Тепер знімок, що вказує на `C4'` є точнісінько таким, на який вказував `C5` із нашого прикладу зі зливанням. Кінцевий результат інтеграції змін нічим не відрізняється, проте, у випадку перебазовування отримуємо чистішу історію. Якщо подивитеся на журнал перебазованої гілки, то він буде лінійним: здається, що вся робота була послідовною, незважаючи на те, що насправді, вона виконувалася паралельно.

Ви часто використовуватимите це, щоб переконатися в тому, що ваші коміти накладалися на віддалені гілки чисто – наприклад, якщо хочете зробити внесок до проекту, яким самі не керуєте. У цьому випадку, ви працюватимете в гілці й потім перебазуєтеся на `origin/master`, коли будете готові надіслати свої латки до основного проекту. У свою чергу, керуючий проектом не змушений робити ніякої додаткової інтеграційної роботи – просто перемотати гілку або чисто накласти зміни.

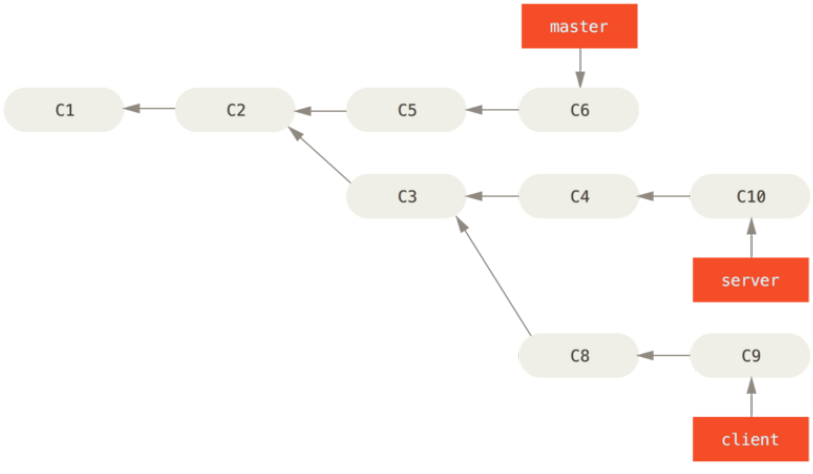
Зауважте, що відбиток, на який вказує фінальний коміт, чи він є останнім з перебазовуваних комітів у випадку перебазовування, чи це фінальний коміт зливання, є одинаковим – відрізняється лише історія. Операція перебазовування відтворює зміни з однієї лінії роботи на іншій лінії в тій же послідовності, в якій ці зміни зустрічалися, у той час, як зливання бере кінцеві точки та зливає їх докупи.

Цікавіше перебазовування

Під час перебазовування ви можете програвати зміни не лише на цільовій гілці. Для прикладу візьміть історію **Figure 3-31**. Ви створили тематичну гілку (`server`) для того, щоб додати певну серверну функціональність до свого проекту та додали туди коміт. Потім створили гілку для змін на клієнтській стороні (`client`), де встигли зробити кілька комітів. Нарешті, повернулися до гілки з серверними змінами та додали кілька комітів там.

ЗОБРАЖЕННЯ 3-31

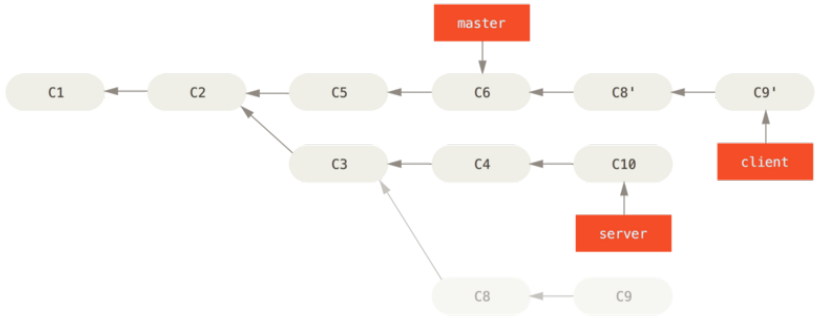
Історія з тематичною гілкою, відгалуженою від іншої тематичної гілки



Допустимо, ви вирішили злити зміни клієнтської сторони до головної лінії коду для релізу, але хочете зачекати з серверними, доки вони будуть більше відтестовані. Для цього можете взяти клієнтські зміни, які ще не є на сервері (`C8` і `C9`), та відтворити їх на `master`, користуючись опцією `--onto` команди `git rebase`:

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

Що означає “Перейди на клієнтську гілку, знайди латки, починаючи із спільного предка гілок `client` та `server`, та відтвори їх на `master`.” Це трохи складно, але результат досить класний.

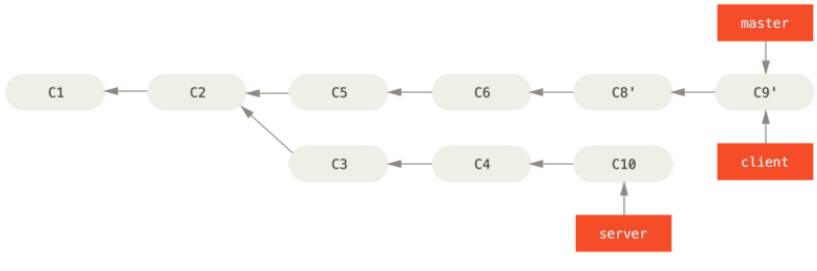


ЗОБРАЖЕННЯ 3-32

Перебазовування тематичної гілки, що відгалужена від іншої гілки

Тепер можете перемотати свою основну гілку `master` (дивіться **Figure 3-33**):

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



ЗОБРАЖЕННЯ 3-33

Перемотування гілки `master`, для залучення в неї змін з клієнтської гілки

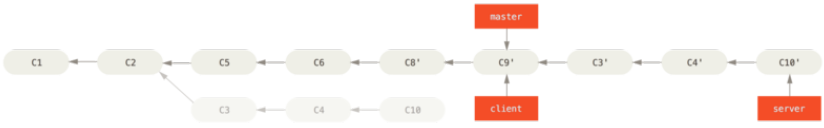
Скажімо, тепер ви також вирішили прийняти зміни з серверної гілки. Ви можете перебазувати серверну гілку на `master` без переходу на неї, користуючись командою `git rebase [базова гілка] [тематична гілка]`, що перейде на тематичну гілку (в нашому випадку `server`) та застосує її зміни на базовій гілці (`master`):

```
$ git rebase master server
```

Це відтворить вашу зроблену роботу з `server` поверху `master`, як відображено в **Figure 3-34**.

ЗОБРАЖЕННЯ 3-34

Перебазовування серверної гілки поверху `master`



Тоді, можете перемотати базову гілку (`master`):

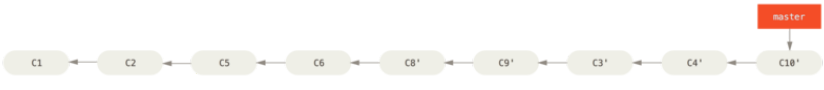
```
$ git checkout master
$ git merge server
```

Гілки `client` та `server` можна вилучати, оскільки вся робота з них вже інтегрована і вам вони більше не знадобляться, що зробить вашу історію такою як **Figure 3-35**:

```
$ git branch -d client
$ git branch -d server
```

ЗОБРАЖЕННЯ 3-35

Остаточна історія комітів



Небезпеки перебазовування

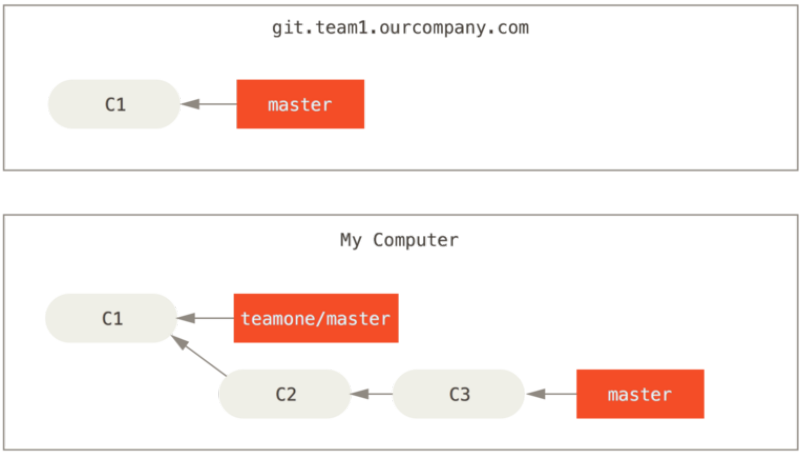
Ох, але все блаженство перебазовування не позбавлене й своїх недоліків, які можна підсумувати одним рядком:

Не перебазовуйте коміти, що існують в інших репозиторіях.

Якщо будете слідувати цьому правилу, то все буде гаразд. Якщо ні, то люди ненавидітимуть вас, а сім'я та друзі зневажатимуть.

Коли ви перебазовуєте, то втрачаєте існуючі коміти, а натомість створюєте нові, які є схожі, проте інші. Якщо ви виклали (push) коміти та інші забрали (pull) їх, базуючи на них свою роботу, а потім ви перезаписали ці коміти з допомогою `git rebase` та виклали (push) їх знову, учасники проекту будуть змушені знову зливати свою роботу, а потім у вас буде плутанина, коли ви будете тягнути (pull) їхню роботу назад у свій репозиторій.

Погляньмо на приклад того, як перебазовування змін, які вже зробили публічними, може створювати проблеми. Нехай ви зклонували репозиторій з центрального сервера і трохи в ньому попрацювали. Історія комітів виглядає десь так:



ЗОБРАЖЕННЯ 3-36

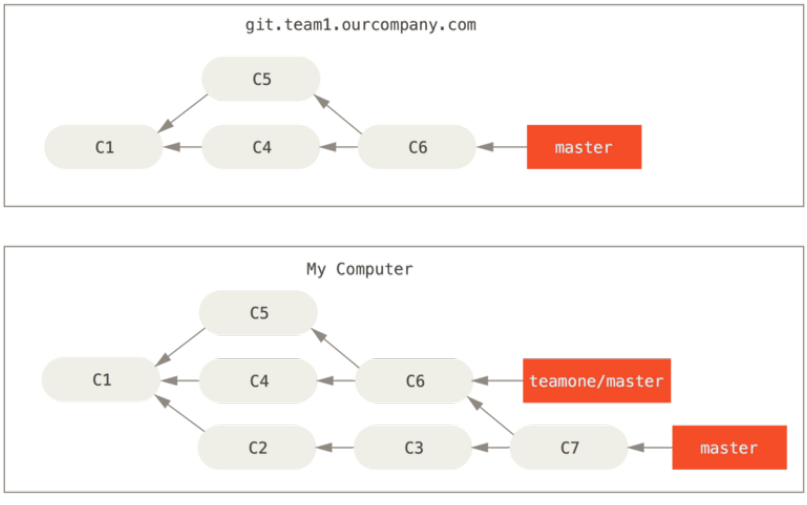
Склонували репозиторій та трохи в ньому попрацювали

Тепер ще хтось додав змін, включно із зливанням, та відправив ці зміни на центральний сервер.

Ви отримали ці зміни та злили нову віддалену гілку зі своєю роботою, отже ваша історія виглядає десь так:

ЗОБРАЖЕННЯ 3-37

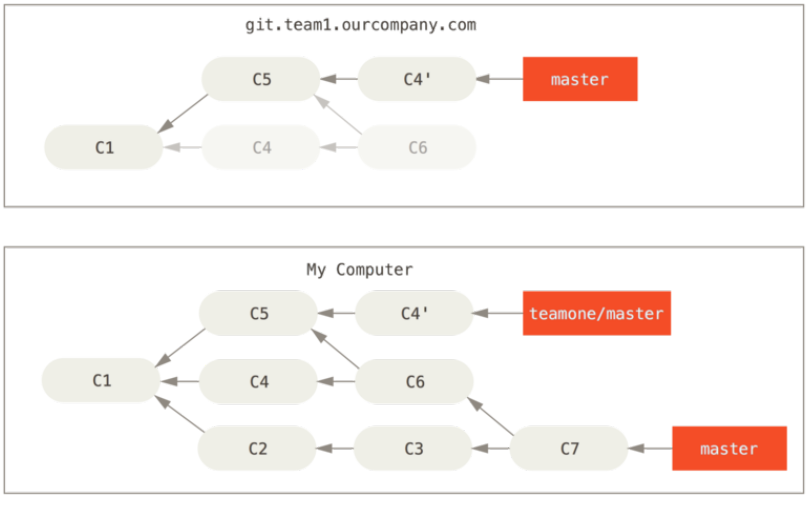
Отримали ще коміти та злили їх із своєю роботою



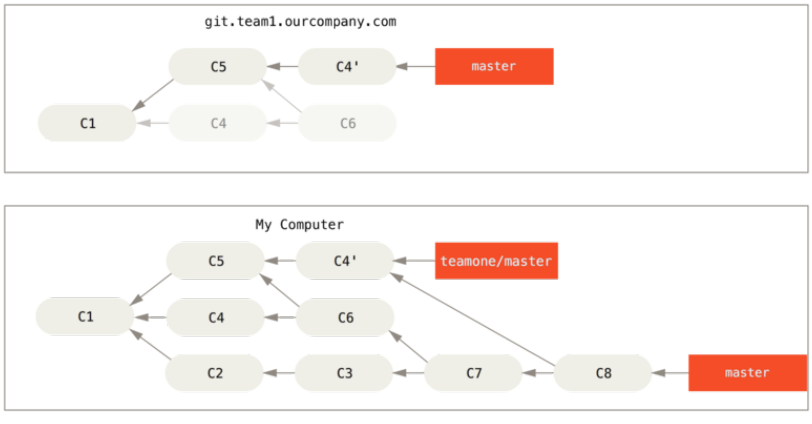
Далі, ті, хто викладали та зливали зміни, вирішили повернутися та перебазувати зроблену роботу, виконавши `git push --force` для того, щоб переписати історію на сервері. Коли ви оновитися (`fetch`), то отримаєте з сервера нові коміти.

ЗОБРАЖЕННЯ 3-38

Хтось виклав перебазовані коміти, зневажаючи коміти, на яких ви базували свою роботу



Тепер ви обидвоє в неприємній ситуації. Якщо виконаєте `git pull`, то створите новий коміт злиття, який включатиме обидві лінії історії, а ваш репозиторій виглядатиме так:



ЗОБРАЖЕННЯ 3-39

Ви знову виконали зливання тої самої роботи в новий коміт зливання

Коли ви виконаєте `git log` для такої історії, то побачите два коміти, що мають того ж автора, дату та повідомлення, що є досить заплутаним. Більше того, коли ви відправите (push) таку історію, ви відновите всі перебазовані коміти на центральному сервері, що й надалі заплутуватиме інших. Мабуть, інші розробники не хочуть мати `C4` та `C6` в історії, власне чому вони й ініціювали перебазовування.

Перебазовуйся коли перебазовуєшся

Якщо ж ви **опинилися** в схожій ситуації, Git має деякі чари, що можуть допомогти. Коли хтось у вашій команді примусово надіслав зміни, що перетерли роботу, на якій ви базувалися, — вам доведеться розібратися котрі зміни ваші, а котрі перетерто.

Виявляється, що крім контрольної SHA-1 суми коміту Git також вираховує контрольну суму, що базується на латці (patch), спричиненою комітом. Вона називається “patch-id”.

Якщо ви візьмете роботу, що була перетертою, та перебазуєте її поверх нових комітів ваших колег, Git часто може сам розібратися що є унікально вашим та застосувати це поверху нової гілки.

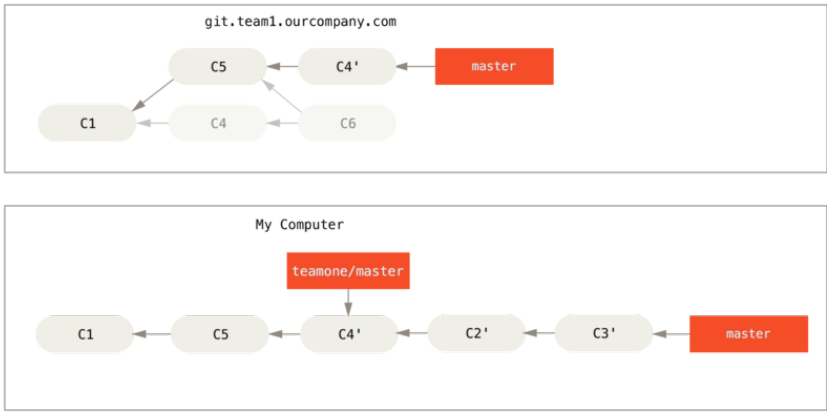
Скажімо, для попереднього прикладу, якщо, будучи на **Figure 3-38**, ми, замість зливання, виконаємо `git rebase teamone/master`, Git

зробить таке: * Визначить які зміни є унікальні для нашої гілки (C2, C3, C4, C6, C7) * Визначить що не є комітами злиття (C2, C3, C4) * Визначить які коміти не були переписані в цільовій гілці (лише C2 та C3, оскільки C4 має таку ж латку як і C4') * Застосує ті коміти поверху `teamone/master`

Отже, замість результатів, які ми бачили в **Figure 3-39**, ми матимемо щось більш схоже на **Figure 3-40**.

ЗОБРАЖЕННЯ 3-40

Перебазовування поверх примусово надісланих (force-pushed) змін.



Це спрацювало лише тому, що C4 та C4' що ви та ваш колега зробили мають майже однакові латки. Інакше перебазовування не могло б визначити що це є дублікат і додало б ще одне C4-подібну латку (яка, швидше за все не могла б успішно застосуватися, оскільки зміни вже були б у хоча б одному місці).

Ви також можете спростити процес виконуючи `git pull --rebase` замість звичайного `git pull`. Або "вручну" виконати послідовно `git fetch` та `git rebase teamone/master`.

Якщо ви використовуєте `git pull` та бажаєте щоб `--rebase` виконувався за замовчуванням, можете відповідно встановити конфігураційну опцію `pull.rebase`, виконавши щось типу `git config --global pull.rebase true`.

Якщо ви сприймаєте перебазовування як спосіб очистки комітів перед надсиланням (push) та якщо ви перебазовуєте лише коміти, що не були досі опубліковані, тоді все буде гаразд. Якщо ж ви перебазовуєте коміти, що вже були опубліковані, а інші вже базують свої зміни на них, то легко можете опинитися в жахливих неприємностях та зневажанні колег.

Якщо ж ви, чи ваші колеги, все ж мають таку необхідність, з тих чи інших причин, упевніться що інші виконують `git pull --rebase` для того, щоб трохи полегшити неприємну ситуацію.

Перебазовування чи зливання

Побачивши перебазовування та зливання в дії, у вас може виникнути запитання: котре з них краще. Перед тим, як відповісти, повернімося до того, що для нас означає історія змін.

Одна з точок зору, що історія репозиторію, це — **запис того, що власне трапилося.** Це історичний документ сам по собі, та його не можна підробляти. Під цим кутом, переписування історії є майже блюзнірством; ви *брешете* про те, що ж відбулося насправді. Ну то й що, як там серія заплутаних комітів зливання? Саме так все сталося, нехай репозиторій тримає це для прозорості нащадків.

Протилежна точка зору — сприймати історію як **оповідання про те, як створювали проект.** Ви ж не будете опубліковувати чернетки вашого творіння, також інструкція з підтримки вашого програмного забезпечення заслуговує детальної редакції. Представники цього табору активно використовують такі інструменти як перебазовування та фільтрація гілок для представлення історії в спосіб, найзручніший майбутнім читачам.

Повертаючись до питання що краще: перебазовування чи зливання, — сподіваємося ви й самі бачите що воно не таке й просте. Git є могутнім інструментом та дозволяє робити багато речей з вашою історією, проте кожна команда та проект є різними. І тепер, коли ви вже познайомилися з тим, як працюють обидва підходи, рішення про те, який кращий для вашої ситуації, залишається за вами.

Взагалі, щоб отримати найкраще з двох світів, дотримуйтеся правила перебазовувати ті локальні зміни, якими ви ще не поділилися, перед надсиланням для того, щоб тримати свою історію чистішою, але ніколи не перебазовувати те, що вже було кудись надіслано.

Підсумок

Ми ознайомили вас з основами галуження та зливання в Git. Тепер ви повинні вміти створювати гілки та перемикатися між ними, зливати зміни в локальних гілках. Також ви навчилися як ділитися своїми гілками, надсилаючи їх на спільний сервер, працювати на спільних гілках та перебазовувати свої гілки перед тим, як ділитися з іншими.

Далі ми розглянемо як налаштувати Git сервер для сховищ вашого власного коду.

Git на сервері 4

Наразі, ви маєте бути в змозі виконувати більшість повсякденних задач, які вам зустрінуться при використанні Git. Втім, щоб мати можливість співпрацювати за допомогою Git, вам треба мати віддалене сховище Git. Хоч ви й можете викладати та забирати зміни зі сховищ кожної людини, це не рекомендується, адже так дуже легко заплутатися в тому, хто що робить, якщо не бути дуже обережним. До того ж, якщо ви бажаєте, щоб ваші колеги мали доступ до сховища навіть коли ваш комп'ютер поза мережею, мати більш надійне спільне сховище зазвичай розумно. Отже, зазвичай для співпраці з кимнебудь налаштовують проміжне сховище, до якого ви обидва маєте доступ, та викладаєте до і забираєте з нього.

Запустити Git сервер доволі просто.

Спершу вам треба обрати протокол, яким ви бажаєте щоб ваш сервер спілкувався. У першій секції цього розділу ми розповімо про доступні протоколи, переваги та недоліки кожного. Наступна секція пояснить деякі типові схеми використання цих протоколів та як змусити ваш сервер з ними працювати. В останній, ми поговоримо про деякі опції хостингу, якщо ви не проти зберігати ваш код на чужому сервері, та не бажаєте мати клопіт зі встановленням та підтримкою вашого власного серверу.

Якщо вас не цікавить запуск власного серверу, ви можете відразу перейти до останньої секції розділу, щоб побачити деякі варіанти налаштування хостингу та переходити до наступного розділу, де ми розглянемо різноманітні деталі роботи в середовищі розподіленої системи контролю коду.

Віддалене сховище зазвичай *очищене сховище* - сховище Git, що не має робочої теки. Адже сховище використовується тільки як місце для співпраці, нема підстав мати копію знімку на диску. Там просто дані Git. Найпростішими словами, очищене сховище містить тільки вміст теки `.git` вашого проекту та більше нічого.

Протоколи

Git може використовувати чотири основних протоколи для передачі даних: Локальний, HTTP, Secure Shell (SSH) та Git. Тут ми розглянемо що вони таке та у яких обставинах ви бажаєте (чи не бажаєте) їх використовувати.

Локальний протокол

Локальний протокол найпростіший, він потребує щоб віддалене сховище було в іншій теці на диску. Він часто використовується, якщо всі з вашої команди мають доступ до розподіленої файлової системи на кшталт NFS , чи в менш імовірному випадку, що всі заходять на один комп*ютер. Останній варіант далекий від ідеалу, адже усі копії вашого сховища перебувають на одному комп*ютері, що підвищує ймовірність катастрофічної втрати.

Якщо у вас спільна файлова система, то ви можете клонувати, викладати до, та забирати з локального файлового сховища. Щоб зробити клон такого сховища чи додати як віддалене сховище до існуючого проекту, достатньо використати шлях до сховища в якості URL. Наприклад, щоб зробити клон локального сховища, ви можете виконати щось таке:

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

Чи зробити так:

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

Git діє трошки по іншому, якщо ви явно задаєте `file://` на початку URL. Якщо ви вкажете тільки шлях, Git спробує використати тверде посилання (hardlink), або просто скопіює теку, якщо потрібно. Якщо ви вкажете `file://`, Git запустить процеси, що він зазвичай використовує для передачі даних через мережу, що зазвичай є набагато менш ефективним методом передачі даних. Зазвичай префікс `file://` використовують, якщо бажають отримати чисту копію сховища без зовнішніх посилань чи об'єктів – зазвичай після імпорту з іншої системи контролю версій, чи чогось подібного (дивіться **Chapter 10** щодо завдань підтримки). Ми будемо користуватись звичайним шляхом, адже це майже завжди швидше.

Щоб додати локальне сховище до існуючого проекту під контролем Git, ви можете виконати щось таке:

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

Після цього ви можете викладати до та забирати з цього віддаленого сховища, ніби ви робите це через мережу.

ПЕРЕВАГИ

Перевага локальних сховищ в тому, що вони прості та використовують існуючи права доступу до файлів та доступу до мережі. Якщо у вас вже є спільна файлова система, до якої має доступ уся ваша команда, налаштувати сховище дуже просто. Ви кладете очищену копію свого сховища в якесь місце, до якого всі мають доступ, та налаштовуєте права читання/запису, як і для будь-якої іншої спільної теки. Ми розглянемо як експортувати очищену копію сховища для цього в **“Отримання Git на сервері”**.

Це також гарний варіант, щоб швидко взяти працю з іншого робочого сховища. Якщо ваш колега працює з вами над одним проектом та ви бажаєте щось перевірити, виконати команду схожу на `git pull /home/taras/project` часто легше, ніж щоб вони викладали до віддаленого серверу, а ви з нього забирали.

НЕДОЛІКИ

Головний недолік цього методу в тому, що налаштувати спільний доступ з декількох вузлів зазвичай складніше, ніж простий мережевий доступ. Якщо ви бажаєте викладати з вашого ноутбуку, коли ви вдома, ви маєте примаунити віддалений диск, що може бути складно та повільно порівняно зі основаним на мережі доступом.

Також важливо зазначити, що це не обов’язково найшвидша опція, якщо ви використовуєте спільний диск якогось типу. Локальне сховище швидке тільки якщо у вас є швидкий доступ до даних. Сховище на NFS зазвичай повільніше, ніж сховище через SSH на тому ж сервері, що дозволяє Git працювати з локальними дисками обох систем.

Протоколи HTTP

Git може спілкуватись через HTTP у двох режимах. До версії 1.6.6 у Git був тільки один метод, що був дуже простим та зазвичай тільки для читання. У версії 1.6.6 та новіших, був доданий кмітливіший протокол, що включає можливість Git проводити розумну передачу даних, схожу на те, як він працює через SSH. В останні декілька років, цей новий протокол HTTP став дуже розповсюдженим, адже він простіший для користувача та кмітливиший щодо методу передачі. Новішу версію часто називають “Розумним” HTTP протоколом, а старішу “Тупим” HTTP Ми спочатку поговоримо про “розумний” HTTP протокол.

РОЗУМНИЙ HTTP

“Розумний” HTTP діє дуже схоже на те, як працює SSH чи Git протоколи, проте працює через звичайні HTTP/S порти та може використовувати різноманітні механізми авторизації HTTP, отже часто цей метод простіший для користувачів, ніж SSH, адже можна використовувати логін/пароль базової авторизації замість налаштування ключів SSH.

Розумний HTTP напевно став найпопулярнішим методом використання Git, адже його можна налаштувати як для анонімної праці, як протокол `git://`, та до нього можна викладати з авторизацією та шифрування, як і з SSH протоколом. Замість налаштування різних URLів для цих речей, ви можете використовувати для них однин URL. Якщо ви спробуєте викласти, а сховище вимагає авторизації (як зазвичай і повинно бути), сервер запитає логін та пароль. Те ж саме стосується і доступу на читання.

Насправді, для таких сервісів як GitHub, URL, що ви використовуєте для перегляду сховища в мережі (наприклад, “***https://github.com/schacon/simplegit***”) збігається з URLом, що ви можете використовувати для клонування, та, якщо у вас є доступ, викладати до нього.

ТУПИЙ HTTP

Якщо сервер не відповідає на розумний сервіс HTTP, клієнт Git спробує відкотитись до простішого “тупого” HTTP протоколу. Тупий протокол очікує, що очищене сховище Git буде обслуговуватись як звичайні файли на веб сервері. Краса тупого протоколу HTTP в простоті його налаштування. Вам треба просто викласти очищене сховище Git під коренем документів HTTP та встановити потрібний гачок (хук, hook) `post-update`, ось і все (Дивіться **“Git Hooks”**). Після цього, усі, в кого є доступ до веб серверу, на котрий ви скопіювали своє сховище, можуть

зробити його клон. Щоб дати доступ на читання вашого сховища через HTTP, зробіть щось таке:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Ось і все. Гачок `post-update` входить у стандартну поставку Git, та виконує відповідні команди (`git update-server-info`), щоб HTTP діставання та клонування працювало правильно. Ця команда виконується, коли ви викладаєте до цього сховища (можливо через SSH). Тоді інші люде люди зможуть клонувати за допомогою

```
$ git clone https://example.com/gitproject.git
```

Саме у цьому випадку, ми використовуємо шлях `/var/www/htdocs`, що є звичайним для серверів Apache, проте ви можете використовувати статичний веб сервер – просто покладіть очищене сховище до нього. Дані Git видаються як прості статичні файли (дивіться **Chapter 10** для докладнішої розповіді про те, як саме вони видаються) Зазвичай ви або виберете Розумний HTTP сервер з можливістю читання/запису, або просто організуєте доступ до файлів з правом тільки на читання за допомогою Тупої опції. Дуже рідко ці два варіанти суміщають.

ПЕРЕВАГИ

Ми зосередимось на перевагах Розумної версії HTTP протоколу.

Простота використання одного URL для всіх типів доступу та те, що сервер потребує авторизації тільки коли потрібно робить користування сервером дуже простим для користувача. Можливість авторизації з логіном та паролем також велика перевага над SSH, оскільки користувачі не мають генерувати SSH ключі локально та завантажувати публічний ключ до серверу до того, як вони зможуть взаємодіяти з ним. Для менш досвідчених користувачів, чи користувачів, що використовують системи де SSH менш розповсюджений, це може стати головною перевагою в зручності. Це також дуже швидкий та ефективний протокол, схожий на SSH.

Ви також можете надавати доступ до вашого сховища тільки на читання через HTTPS, тобто ви можете шифрувати зміст передачі. Чи ви можете навіть змусити клієнтів використовувати специфічні підписані SSL сертифікати.

Ще одна гарна властивість HTTP/S в тому, що це такі розповсюджені протоколи, що корпоративні мережеві екрани зазвичай налаштовані дозволяти передачу через ці порти.

НЕДОЛІКИ

Git через HTTP/S може бути трохи складнішим у порівнянні з SSH на деяких серверах. Окрім цього, інші протоколи мають дуже мало переваг над “Розумним” HTTP протоколом для надавання доступу до Git.

Якщо ви використовуєте HTTP для авторизованого викладання, посвідчення вашого акаунту іноді може буди складнішим, ніж за допомогою ключів через SSH. Втім існує декілька утиліт для кешування даних входу, наприклад Keychaing access на OSX та Credential Manager на Windows, що дозволяє уникнути цієї проблеми. Прочитайте **???** щоб дізнатись як налаштувати безпечне кешування HTTP паролю на вашій системі.

Протокол SSH

SSh доволі поширений протокол передачі для Git при самостійному хостінгу. Причина в тому, що доступ SSH до серверу в більшості випадків вже налаштовано – а якщо ні, це дуже легко зробити. SSH також має авторизацію та працює у мережі. Оскільки він повсюдний, зазвичай його легко налаштовувати та використовувати.

Щоб зробити клон Git сховища через SSH, ви можете задати URL ssh:// ось так:

```
$ git clone ssh://user@server/project.git
```

Чи ви можете використати скорочений синтаксис (подібний до scp) для SSH протоколу:

```
$ git clone user@server:project.git
```

Ви також можете не задавати користувача, тоді Git використає поточного користувача.

ПЕРЕВАГИ

Є багато переваг використання SSH. По-перше, SSH відносно легко налаштувати – демони SSH є повсюди, багато мережевих адміністраторів мають з ними досвід, та багато дистрибутивів поставляються з ними та навіть мають утиліти щоб ними керувати. Далі, доступ через SSH безпечний – усі дані передачі зашифровані та авторизовані. Наостанок, як і HTTP/S, Git та Локальний протоколи, SSH є ефективним, робить дані якомога компактними до відправки.

НЕДОЛІКИ

Від'ємний бік SSH у тому, що за його допомогою неможливо налаштувати анонімний доступ до вашого сховища. Користувачі зобов'язані мати доступ до вашої машини через SSH, навіть тільки для читання, через що SSH доступ не є продуктивним для проектів з відкритим кодом. Якщо ви використовуєте його тільки в межах вашої корпоративної мережі, SSH може бути єдиним протоколом, що вам потрібен. Якщо ви бажаєте дозволити анонімний доступ тільки для читання до ваших проектів та також бажаєте використовувати SSH, вам доведеться налаштувати SSH для викладання, проте щось інше щоб інші могли діставати дані.

Протокол Git

Наступним є протокол Git. Це спеціальний демон, що входить до пакету Git. Він слухає на виділеному порту (9418), що надає сервіс схожий на SSH протокол, проте без жодної авторизації. Щоб надати доступ до сховища за допомогою протоколу Git, ви маєте створити файл `git-daemon-export-ok` – демон відмовляється роздавати сховище без цього файлу – проте ніяких інших засобів безпеки не надає. Або сховище Git доступно всім, або нікому. Це означає, що зазвичай через цей протокол забороняється викладання. Ви можете ввімкнути доступ до викладання. Проте за відсутності авторизації, якщо ви це зробите, будь-хто в мережі, хто знайде ваше сховище, зможе до нього викладати. Достатньо сказати, що таке має сенс доволі рідко.

ПЕРЕВАГИ

Git протокол часто є найшвидшим доступним протоколом передачі в мережі. Якщо ви передаєте великі обсяги даних для відкритого проекту чи роздаєте дуже великий проект, що не вимагає авторизації для читання, вірогідно ви забажаєте налаштувати Git демон для обслуговування вашого проекту. Він використовує той самий механізм передачі даних, що й SSH протокол, проте без додаткових витрат на шифрування та авторизацію.

НЕДОЛІКИ

Головним недоліком протоколу Git є відсутність авторизації. Зазвичай небажано щоб протокол Git був єдиним протоколом доступу до вашого проекту. Зазвичай, його використовують у парі з SSH чи HTTPS доступом для декількох розробників, що мають право викладати (писати), а усі інші використовують `git://` для читання. Також це мабуть найскладніший для налаштування протокол. Для нього має працювати власний демон, що вимагає щось подібне до конфігурації `xinetd`, що не завжди схоже на прогулянку по парку. Також це вимагає щоб мережеві екрани дозволяли доступ до порту 9418, що не є стандартним дозволеним корпоративним портом. За великими корпоративними мережевими екранами, цей незрозумілий порт зазвичай є заблокованим.

Отримання Git на сервері

Тепер ми розглянемо налаштування сервісу Git з цими протоколами на вашому власному сервері.

Тут ми продемонструємо команди та кроки, що потрібні для базового, спрощеного встановлення на базовому Linux сервері, хоча ці сервіси також можуть працювати під Mac або Windows серверами. Насправді налаштування виробничого серверу у вашій інфраструктурі безперечно буде накладати відмінності у міри безпеки чи в роботі з системними утилітами, проте сподіваємось, що ця секція дасть вам загальне уявлення про процес.

Щоб налаштувати сервер Git, спочатку треба експортувати існуюче сховище до нового очищеного сховища – сховища, що не містить робочої теки. Це доволі просто зробити. Щоб зробити клон вашого сховища та створити нове очищене сховище, треба виконати команду

clone з опцією --bare. За умовною домовленістю, теки очищених сховищ закінчуються на .git, наприклад:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

Тепер у вас є копія даних теки Git у теці my_project.git.
Це майже рівнозначно виконанню

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

Є декілька незначних відмінностей у файлі конфігурації, проте для наших цілей, це майже одне й те саме. Ми беремо саме сховище Git, без робочої теки, та створюємо теку спеціально і тільки для нього.

Розміщення очищеного сховища на сервер

Тепер, коли у вас є очищена копія вашого сховища, все, що вам потрібно зробити – це розмістити його на сервері та налаштувати протоколи. Припустимо, ви налаштували сервер за назвою git.example.com, до якого у вас є SSH доступ, та ви бажаєте зберігати усі ваші сховища Git під текою /opt/git. Якщо тека /opt/git вже існує на сервері, ви можете налаштувати своє нове сховище, просто копіюванням вашого очищеного сховища:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

Наразі, інші користувачі що мають SSH доступ до цього серверу, та мають доступ тільки на читання теки /opt/git можуть зробити клон вашого сховища, виконавши

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

Якщо користувач зайде через SSH до серверу та має доступ на запис до теки /opt/git/my_project.git, він автоматично матиме право викладати зміни.

Git автоматично додасть групові права на запис до сховища вірно, якщо ви виконаєте команду `git init` з опцією `--shared`.

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

Бачите як просто взяти сховище Git, створити його очищену версію, та розмістити її на сервері, до якого ви та ваші колеги мають SSH доступ. Тепер ви готові до співпраці над одним проектом.

Важливо зазначити, що це літерально все, що вам потрібно щоб отримати працюючий корисний Git сервер, до якого мають доступ декілька людей – просто додайте SSH акаунти на сервері, та закиньте кудись очищене сховище, до якого всі користувачі мають доступ на читання та запис. Ось і все – більше нічого не треба.

У декількох наступних секціях, ви побачите як перейти до більш витончених схем. Ми розглянемо як не створювати акаунти для кожного користувача, як додати відкритий доступ на читання сховища, налаштування веб інтерфейсу та інше. Втім, пам'ятайте, що для співпраці з декількома людьми над закритим проектом, усе що *треба* – це SSH сервер та очищене сховище.

Маленькі проекти

Якщо ви маленька контора, або ви просто випробовуєте Git у вашій організації, та маєте тільки кількох програмістів, усе дуже просто. Один з найскладніших аспектів налаштування серверу Git – це керування користувачами. Якщо ви бажаєте, щоб якісь сховища були доступні на читання деяким користувачам, а на читання та запис іншим, налаштувати доступ та права може бути трохи складніше.

SSH ДОСТУП

Якщо у вас є сервер, до якого всі ваші програмісти вже мають SSH доступ, зазвичай найпростіше встановити ваше перше сховище там, адже вам майже нічого не доведеться робити (як ми побачимо у наступній секцій). Якщо вам потрібен складніший контроль над доступом та правами до вашого сховища, ви можете з ними впоратися за допомогою звичайних прав файлової системи, на якій розташоване ваше сховище.

Якщо ви бажаєте розмістити ваші сховища на сервері, до якого не у всіх з вашої команди, у кого має бути доступ на запис, є акаунти, вам треба налаштувати SSH доступ для них. Ми далі припускаємо, що на вашому сервері SSH сервер вже є встановленим, та ви саме так заходите на сервер.

Є декілька способів надати доступ усій вашій команді. Ви можете просто додати акаунти всім, що є найпростішим варіантом, проте може бути трудомістким варіантом. Ви можете не захотіти виконувати `adduser` та встановлювати тимчасові паролі для кожного користувача.

Другий метод – це створити єдиного користувача *git*, попросити кожного користувача, у котрого має бути доступ на запис відправити вам публічний ключ SSH, та додати всі ключі до файлу `~/.ssh/authorized_keys` вашого нового користувача *git*. Після цього, усі будуть мати доступ до машини через користувача *git*. Це ніяким чином не впливає на дані фіксації – користувач SSH, з якого ви підключаєтесь, не впливає на фіксації, що ви записуєте.

Інший спосіб це зробити щоб SSH сервер авторизувався з LDAP серверу чи іншого централізованого джерела авторизації, що у вас може бути вже встановленим.

Генерація вашого публічного ключа SSH

Як вже було сказано, багато Git серверів авторизуються за допомогою публічних ключів SSH. Щоб надати публічний ключ, кожен користувач вашої системи має його згенерувати, якщо в них його досі нема. Цей процес однаковий незалежно від операційної системи. Спочатку треба переконатися, що у вас ще нема ключа. Без додаткових опцій, ключі SSH користувача зберігаються у теці `~/.ssh`. Ви можете легко побачити, чи ви вже маєте ключ, якщо перейдете до теки та подивитесь на її зміст:

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

Шукайте пару файлів з назвою схожою на `id_dsa` чи `id_rsa` та відповідний файл з розширенням `.pub`. Файл `.pub` і є вашим публічним ключем, а другий файл – це ваш приватний ключ. Якщо у вас немає цих файлів (або ви навіть не маєте теки `.ssh`), ви можете створити їх

за допомогою програми ssh-keygen, яка поставляється разом з пакетом SSH на Linux/Mac системах та входить до пакету MSysGit під 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
```

Спочатку вона дізнається, де ви бажаєте зберегти ключ (.ssh/id_rsa), а потім двічі питає passphrase, яку ви можете залишити пустою, якщо не бажаєте при кожному використанні ключа набирати пароль.

Тепер кожен користувач має це зробити та відправити свій публічний ключ до вас чи до того, хто адмініструє Git сервер (у разі використання SSH серверу що вимагає публічних ключів). Все що їм треба зробити – скопіювати вміст файлу .pub та відправити його електронною поштою. Публічні ключі виглядають приблизно так:

```
$ 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
```

Для докладнішої інструкції по використанню SSH ключів на багатьох операційних системах, дивіться посібник GitHub щодо SSH ключів за адресою ***https://help.github.com/articles/generating-ssh-keys***.

Налаштування Серверу

Розглянемо покрокове налаштування SSH доступу на сервері. У цьому прикладі, ми використаємо метод авторизованих ключів (authorized_keys) для авторизації ваших користувачів. Ми знови

припустимо, що на вашому сервері стандартний дистрибутив Лінукс, наприклад Ubuntu. Спершу, створемо користувача `git` та теку `.ssh` для цього користувача.

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

Далі, треба додати публічні SSH ключі якогось програміста до файлу `authorized_keys` користувача `git`. Припустимо, що у вас є перевірені публічні ключі та ви їх зберегли до тимчасових файлів. Нагадаємо, публічні ключі виглядають схоже на:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Ви просто долучаєте їх до файлу `authorized_keys` користувача `git` в його теці `.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
```

Тепер ви можете налаштувати порожнє сховище для них за допомогою `git init` з опцією `--bare`, яка створює сховище без робочої теки:

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

Після цього, Джон, Джосі та Джесіка можуть викласти першу версію свого проекту до того сховища, якщо додадуть його як віддалене сховище та викладуть до нього гілку. Завважте що хтось має заходити до машини та створювати очищене сховище щоразу, коли ви хочете додати проект. Використаймо `gitserver` як ім*я хосту (hostname) серверу, на якоми ви налаштували користувача `git` та сховище. Якщо ви ваш сервер працює у внутрішній мережі, та ви налаштуєте DNS щоб ім*я `gitserver` вказувало на ваш сервер, то ви можете використовувати команди як і раніше (припустимо, що `myproject` це існуючий проект з файлами):

```
# на машині Джона
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

Після цього, усі інші можуть зробити клон цього сховища, та викладати зміни назад так само легко:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

За допомогою цього метода, ви можете швидко отримати працюючий сервер Git з доступом на читання та запис для декількох програмістів.

Зауважте, що наразі усі ці користувачі також можуть заходити на сервер та отримують доступ до оболонки (shell) як користувач `git`. Якщо ви не хочете цього дозволяти, вам треба змінити програму оболонки на щось інше у файлі `passwd`.

Ви легко можете обмежити користувача `git` до виключно активності Git за допомогою утиліти `git-shell`, що входить до поставки Git. Якщо ви вкажете її для вашого користувача `git`, як програму, що запускається при вході, то користувач `git` не зможе здобути нормальний доступ до вашого серверу. Щоб цим скористатись, задайте `git-shell` замість bash чи csh як оболонку при вході для

вашого користувача. Щоб це зробити, треба спочатку додати `git-shell` до `/etc/shells`, якщо його там ще нема:

```
$ cat /etc/shells   # перевірте, може `git-shell` вже є у файлі. Якщо ні...
$ which git-shell   # переконайтесь, що git-shell присутній на вашій системі.
$ sudo vim /etc/shells  # та додайте шлях до git-shell за допомогою останньої команди
```

Тепер ви можете відредагувати оболонку для користувача за допомогою `chsh <ім'я користувача>`:

```
$ sudo chsh git  # та введіть шлях до git-shell, зазвичай: /usr/bin/git-shell
```

Тепер користувач `git` може використовувати з'єднання SSH виключно щоб викладати та забирати сховища Git, та не може зайти до машини. Якщо ви спробуєте, ви побачите відмову від входу, схожу на:

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

Тепер мережеві команди Git продовжать працювати без проблем, проте користувачі не зможуть отримати оболонку. Як і стверджує вивід, ви тепер можете створити теку в домашній теці користувача `git`, що трохи змінити поведінку команду `git-shell` Наприклад, ви можете обмежити команди Git, які прийме сервер, або ви можете змінити повідомлення, яке бачать користувачі, якщо вони спробують зайти через SSH. Виконайте `git help shell` для докладнішої інформації про це.

Демон Git

Тепер встановимо демон, що надає доступ до сховищ через протокол "Git". Це поширений вибір для швидкого, неавторизовоного доступу до ваших даних Git. Пам'ятайте, що оскільки він не авторизований, все до чого ви надаєте доступ за допомогою цього протоколу стає загальнодоступним у його мережі.

Якщо сервер працює за межами вашого мережевого екрану, він має використовуватись виключно для проектів, що є загальнодоступними для світу. Якщо ж сервер працює в межах мережевого екрану, ви можете використовувати його для проектів, до яких велика кількість людей або комп'ютерів (наприклад сервери компіляції чи інтеграції) мають доступ тільки на читання, але ви не хочете додавати ключ SSH для кожного з них.

У будь-якому разі, протокол Git легко налаштувати. Вам просто треба виконати цю команду так, щоби вона працювала як демон:

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

`--reuseaddr` дозволяє серверу себе перезапускати без очікування, доки старі з'єднання спливуть (time out), опція `--base-path` дозволяє клонувати сховища за допомогою шляху відносно значення цієї опції замість повного шляху. Якщо у вас працює мережевий екран, ви також маєте дозволити передачу через порт 9418 машини з Git сервером.

Запустити цю команду як демон можна декількома шляхами, в залежності від вашої операційної системи. Під Ubuntu, ви можете використати скрипт Upstart. Отже, у файл

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

додайте такий скрипт:

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

З причин безпеки, ми заохочуємо вас запускати цей демон тільки з користувача, що має доступ тільки на читання сховищ – це легко зробити, створивши нового користувача *git-ro* та запустивши демон з нього. Для простоти ми просто запустимо його з нашого користувача `git`, з якого працює і `git-shell`.

Коли ви перезавантажите свою машину, демон Git автоматично запуститься та буде сам перезапускатися, якщо він впаде. Щоб запустити його без перезапуску, ви можете виконати:

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

На інших системах, ви можете використати `xinetd`, скрипт у теці системи `sysvinit`, чи щось інше – головне щоб команда запускалась як демон, та можна було перевіряти її статус.

Далі, вам треба сказати Git, до яких сховищ дозволяти неавторизований доступ за допомогою сервера Git. Ви можете зробити це в кожному сховищі за допомогою створення файлу під назвою `git-daemon-export-ok`.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

Присутність цього файлу каже Git, що він може роздавати цей проект без авторизації.

Розумний HTTP

Ми вже маємо авторизований доступ через SSH та неавторизований через `git://`, проте існує ще протокол, котрий може працювати в обох варіантах одночасно. Налаштування Розумного HTTP потрибує лише ввімкнути CGI скрипт, що входить в пакет Git під назвою `git-http-backent`, на вашому сервері. Цей CGI буде читати шлях та заголовки, що їх відправили `git fetch` чи `git push` до HTTP URL та визначати, чи клієнт в змозі спілкуватися через HTTP (що є правдою для будь-якого клієнту після версії 1.6.6). Якщо CGI визначив, що клієнт розумний, він буде з ним спілкуватися розумно, інакше доведеться використовувати тупу поведінку (отже він зворотньо сумісний для читання з старіших клієнтів).

Давайте покроково розглянемо дуже просту схему налаштування. Ми використаємо Apache в якості CGI серверу. Якщо у вас немає Apache, ви можете його встановити на Linux машині приблизно так:

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

Ця команда також вмикає модулі `mod_cgi`, `mod_alias` та `mod_env`. Вони всі необхідні для правильної роботи розумного HTTP.

Далі нам треба додати дещо до конфігурації Apache, щоб `git-http-backend` обробник для будь-якого шляху на вашому веб сервері, що закінчується на `/git`.

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

Якщо ви пропустите змінну середовища `GIT_HTTP_EXPORT_ALL`, Git погодиться роздавати неавторизованим користувачам тільки сховища з файлом `git-daemon-export-ok`, як робить і демон Git.

Тепер вам треба сказати Apache дозволяти запити до цього шляху, якось так:

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

Наостанок, ви бажаєте, щоб для запису була потрібна якась авторизація, можливо схожий на цей блок Auth:

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

Щоб це працювало необхідно створити файл `.htaccess` з паролями дозволених користувачів. Ось приклад того, як можна додати користувача “schacon” до цього файлу:

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

Існує безліч методів авторизації в Apache, вам доведеться вибрати та використати один з них. У цій секції ми навели найпростіший приклад, який тільки могли вигадати. Також ви безсумнівно захочете налаштувати шифрування даних через SSL.

Ми не бажаємо занадто багато розглядати конфігурацію Apache, адже цілком можливо, що ви будете використовувати інший сервер, чи у вас інші потреби авторизації. Суть в тому, що Git поставляє CGI скрипт `git-http-backent`, який може відправляти та приймати дані Git через HTTP. Він не реалізує авторизацію сам, проте авторизацію легко контролювати на рівні веб серверу, що його викликає. Це дозволяє майже кожен веб сервер, що підтримує CGI, отже використовуйте той, що ви найкраще знаєте.

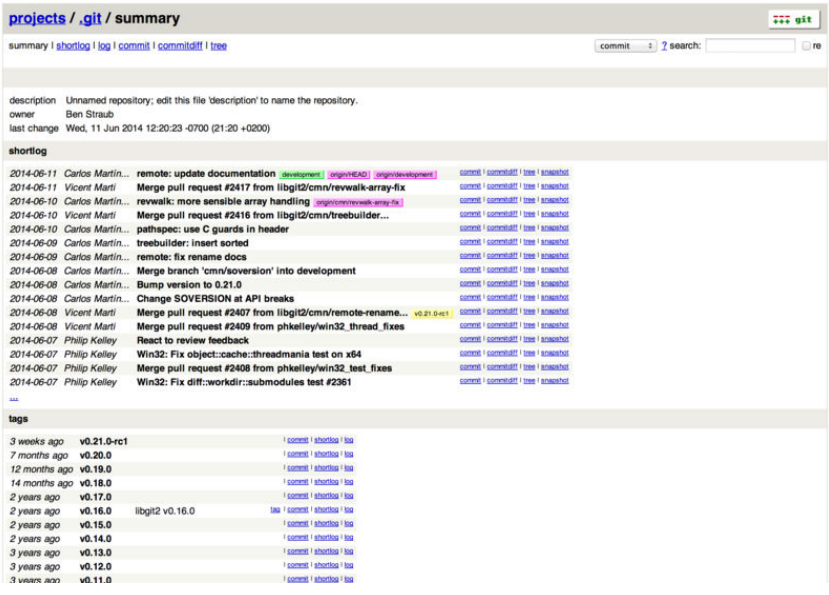
За докладнішою інформацією щодо конфігурації авторизації в Apache, дивитесь документацію Apache за адресою: ***http://httpd.apache.org/docs/current/howto/auth.html***

GitWeb

Тепер, коли у вас є базовий доступ до вашого проекту з правом читання та з правами на читання та запис, можливо ви бажаєте налаштувати простий візуалізатор через веб. Git поставляє CGI скрипт GitWeb, який іноді для цього використовують.

ЗОБРАЖЕННЯ 4-1

Веб інтерфейс користувача за допомогою GitWeb



If you want to check out what GitWeb would look like for your project, Git comes with a command to fire up a temporary instance if you have a light-weight server on your system like `lighttpd` or `webrick`. Якщо ви бажаєте подивитись, як виглядає GitWeb для вашого проекту, Git має команду для запуску тимчасового прикладу, якщо у вас є легкий сервер на системі, наприклад `lighttpd` або `webrick`. На машинах Linux часто є `lighttpd`, отже ви може швидко його запустити, якщо наберете `git instaweb` у теці вашого проекту. Якщо ви використовуєте Mac, Leopard одразу має Ruby, отже `webrick` має бути найпростіше запустити. Щоб запустити `instaweb` не з lighttpd сервером, передайте бажане ім'я серверу опції `--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]
```

Ця команда запускає HTTPD сервер на порту 1234 та автоматично запускає веб-оглядач, що відразу покаже цю сторінку. Вам майже нічого не треба робити. Коли ви закінчили та бажаєте вимкнути сервер, ви можете виконати таку ж команду з опцією `--stop`:

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

Якщо ви бажаєте, щоб веб інтерфейс працював на сервері весь час для вашої команди чи для проекту з відкритим кодом, для якого ви керуєте хостингом, вам треба встановити CGI скрипт, щоб він працював на вашому звичайному веб сервері. Деякі дистрибутиви Linux мають пакет `gitweb`, який ви можливо можете встановити за допомогою `apt` або `yum`, отже можливо варто спочатку спробувати ці команди. Ми дуже швидко розглянемо встановлення GitWeb вручну. Спочатку треба отримати програмний код Git, в якому є і код GitWeb, та згенерувати власний CGI скрипт:

```
$ 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/
```

Зауважте, що ви маєте вказати цій команді, де розташовані ваші сховища Git за допомогою змінної змінної `GITWEB_PROJECTROOT`. Тепер вам треба щоб Apache використовувати CGI для цього скрипту, що можна зробити за допомогою 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>
```

Як і попередній CGI скрипт, GitWeb може працювати на будь-якому здатному на CGI або Perl веб сервері. Якщо ви хочете використовувати

щось інше, налаштування має бути легким. Тепер ви можете зайти до `http://gitserver/` та побачити ваші сховища.

GitLab

Проте GitWeb дуже простий. Якщо ви шукаєте більш сучасний Git сервер з багатшим функціоналом, існує декілька рішень з відкритим кодом, які ви можете встановити замість GitWeb. Оскільки GitLab є одним з найпопулярніших, ми розглянемо його інсталяцію та використання як приклад. Це трохи складніше, ніж варіант GitWeb, та напевно вимагає більше роботи, проте пропонує набагато більший функціонал.

Інсталяція

GitLab є веб програмою, що використовує базу даних для зберігання даних, отже його інсталяція вимагає більше знань, ніж деякі інші git сервери. На щастя, є дуже детальний опис цього процесу.

Є декілька методів, як ви можете досягнути інсталяції GitLab. Щоб швидко отримати щось працююче, ви можете завантажити відбиток віртуальної машини чи інсталятор в один клік з ***https://bitnami.com/stack/gitlab***, та підправити конфігурацію до особливих потреб вашого середовища. Ще одна приємна деталь, що її додала Bitnami, це екран входу (до якого можна перейти, набравши alt-→). Він повідомляє вам IP адресу та логін/пароль до GitLab.

Екран входу віртуальної машини Bitnami GitLab. image::images/bitnami.png[Екран входу віртуальної машини Bitnami GitLab.]

За іншою інформацією звертайтеся до GitLab Community Edition readme, що можна знайти за адресою ***https://gitlab.com/gitlab-org/gitlab-ce/tree/master***. Там ви знайдете підтримку щодо інсталяції GitLab за допомогою рецептів Chef, віртуальної машини на Digital Ocean, та з RPM або DEB пакетів (які, на момент написання, у бета-версії). Є також “неофіційне” керівництво по запуску GitLab з нестандартними операційними системами та базами даних, скрипт інсталяції повністю вручну, та багато іншого.

Адміністрування

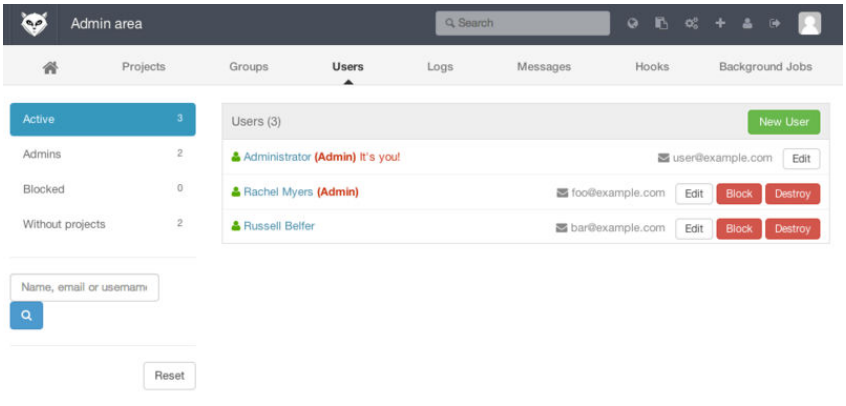
GitLab пропонує веб-інтерфейс для адміністрування. Просто направте ваш оглядач на ім’я хоста або IP адресу, де ви встановили GitLab, та заходьте як користувач admin. Після інсталяції ім’я користувача

`admin@local.host`, а пароль `5iveL!fie` (який вас попросять змінити щойно ви зайдете). Після входу, натисніть іконку меню “Admin area” нагорі праворуч.

Пункт “Admin area” в меню GitLab. image::images/gitlab-menu.png[Пункт “Admin area” в меню GitLab.]

КОРИСТУВАЧІ

Користувачі в GitLab є обліковими записами, що відповідають людям. Облікові записи не дуже складні. Переважно це збір персональної інформації, прив’язаний до даних логіну. Кожен користувач має **простір імен**, що є логічним групуванням проектів, що належать цьому користувачу. Якщо користувач `hanna` мала проект під назвою `project`, то url цього проекту буде: ***http://server/hanna/project***.



ЗОБРАЖЕННЯ 4-2

Екран адміністрування користувачів GitLab.

Видалити користувача можна двома методами. “Блокування” користувача не дає їм заходити на ваш GitLab, проте усі дані з їх простіру імен буде збережено, та фіксації з поштовою адресою цього користувача будуть продовжувати вказувати на його профіль.

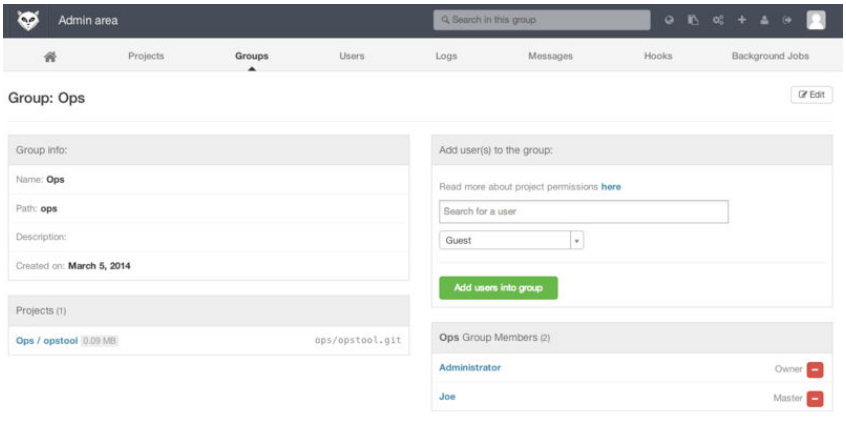
“Винищення” користувача, з іншого боку, повністю видаляє їх з бази даних та файлової системи. Усі проекти та дані в їх просторі імен видаляються, та будь-які групи, що їм належать, також будуть видалені. Це вочевидь більш деструктивна та незмінна дія, тому її використовують зрідка.

ГРУПИ

Група в GitLoab --- це збірка проектів, разом з даними про доступ користувачів до цих проектів. Кожна група має простір імен проекту (так само, як користувачі), отже якщо група `training` має проект `materials`, його url буде ***http://server/training/materials***.

ЗОБРАЖЕННЯ 4-3

Екран адміністрування груп GitLab.



Кожна група пов'язана з декількома користувачами, кожен з яких має свій рівень доступу до проектів групи та до самої групи. Вони різняться від "Guest" (гість, тільки завдання (issues) та чат) до "Owner" (власник, повний контроль над групою, її користувачами та проектами). Типи прав доступу занадто чисельні щоб наводити їх тут, проте GitLab пропонує корисне посилання на екрані адміністрування.

ПРОЕКТИ

Проект GitLab приблизно відповідає одному сховищу git. Кожен проект належить до одного простору імен, що може належати користувачу або групі. Якщо проект належить користувачу, власник проекту має прямий контроль над правами доступу до проекту. Якщо проект належить групі, права доступу групи рівня користувачів також будуть враховані.

Кожен проект також має рівень доступу до перегляду, який контролює в кого є права на читання сторінок та сховища проекту.

Якщо проект *Private* (приватний), то власник проекту має окремо надати права кожному користувачу. *Internal* (внутрішній) проект може

бачити будь-який користувач, що здійснив вхід, а *Public* (публічний) проект може переглядати будь-хто. Завважте, що це контролює і доступ до команди “fetch”, і доступ до сторінок веб-інтерфейсу цього проекту.

ХУКИ

GitLab також підтримує хуки, як на рівні проекту, так і на рівні системи. Для кожного з них, сервер GitLab виконає HTTP POST з детальним описом у форматі JSON, щоразу трапляється якась релевантна подія. Це чудовий спосіб з’єднати ваші сховища git та вашу копію GitLab з рештою ваших інструментів розробки, наприклад сервери безперервної інтеграції, кімнати чатів або утиліти розгортання.

Базове користування

Щойно ви встановите GitLab, ви забажаєте створити новий проект. Щоб це здійснити, вам треба клікнути на значок “+” з панелі інструментів. У вас запитають назву проекту, до якого простору імен він має належати, та який в нього має бути рівень доступу до перегляду. Більшість заданих тут значень легко змінити потім за допомогою інтерфейсу налаштувань. Натисніть “Create Project” (створити проект) коли ви будете готові.

Once the project exists, you’ll probably want to connect it with a local Git repository. Після створення проекту, ви напевно захочете з’єднати його з локальним сховищем Git. Кожен проект є доступним через HTTPS чи SSH, їх обох можна використати для додавання видаленого Git сховища. Ви можете побачити URL’и нагорі домашньої сторінки проекту. Щоб додати видалене сховище під назвою `gitlab` до вже існуючого локального сховища:

```
$ git remote add gitlab https://server/namespace/project.git
```

Якщо у вас немає локальної копію сховища, ви можете просто виконати:

```
$ git clone https://server/namespace/project.git
```

Веб інтерфейс надає доступ до декількох корисних представлень сховища. Домашня сторінка кожного проекту показує нещодавню

діяльність, та посилання нагорі проведуть вас до переглядів файлів проекту та журналу фіксацій.

Співпраця

Найпростіший метод працювати над проектом GitLab разом - це надати іншому користувачу безпосередній доступ на викладання до сховища git. Ви можете додати користувача до проекту, якщо перейдете до секції "Members" налаштувань цього проекту, та задасте новому користувачу рівень доступу (різні рівні доступу трошки розглянуті в **"Групи"**). Якщо ви дасте користувачу рівень доступу "Developer" або вищій, то користувач зможе безкарно викладати фіксації та гілки безпосередньо до сховища.

Інший, менш зчеплений шлях співпраці - це використання запитів на злиття (`merge requests`). Він дозволяє будь-якому користувачу, що бачить ваш проект, робити внесок до нього під вашим наглядом. Користувачі з безпосереднім доступом можуть просто створити гілку, викласти до неї зміни, та відкрити запит на злиття з їхньої гілки назад до `master` чи будь-якої іншої гілки. Користувачі, що не мають прав викладати зміни до сховища, можуть "форкнути" його (створити власну копію), викласти фіксації до *тієї* копії, та відкрити запит на злиття з їхнього форку назад до головного проекту. Ця модель дозволяє власнику повністю контролювати, що потрапить до сховища та коли, і в той же час дозволяє приймати внески від неперевірених користувачів.

Запити на злиття та завдання (`issues`) є головними предметами довгих обговорень в GitLab. Кожен запит на злиття дозволяє порядкове обговорення пропонованих змін (що підтримує легку версію перевірки коду), а також загальне обговорення всього запиту. Обидва можуть бути призначені користувачам, чи організовані у віхи (`milestone`).

Ця секція переважно розглядала функціонал GitLab пов'язаний з Git, проте це зрілий проект, він пропонує багато іншого функціоналу для допомоги вашій команді, наприклад вікі проекту та утиліти по підтримці системи. Одна з переваг GitLab в тому, що після налаштування та запуску серверу, вам тільки зрідка доведеться змінювати файл конфігурації або заходити до серверу через SSH. Більшість адміністративних та загальних завдань доступні через веб інтерфейс.

Варіанти стороннього хостингу

Якщо ви не хочете витрачати час на всю роботу, що пов'язана зі встановленням вашого власного серверу Git, у вас є декілька варіантів хостингу ваших проектів Git на зовнішньому виділеному хостингу. Це надає чисельні переваги: зазвичай хостинг швидко налаштувати, на ньому легко почати проекти, та не треба підтримувати чи слідкувати за сервером. Навіть якщо ви налаштували та запустили свій власний внутрішній сервер, ви все одно можете використати публічний хостинг для вашого відкритого коду – так спільноті зазвичай легше його знайти та допомогти з вашим проектом.

Нині у вас є величезний вибір хостингів, кожен зі своїми перевагами та недоліками. Ви можете побачити оновлюваний список на сторінці GitHosting, на головній вікі Git за адресою ***https://git.wiki.kernel.org/index.php/GitHosting***

Ми в подробицях розглянемо використання GitHub у розділі **Chapter 6**, адже це найбільший хостинг Git, та вам може бути потрібно взаємодіяти з проектами, що викладені на нього, проте є ще десятки варіантів, з яких ви можете вибирати, якщо не бажаєте налаштовувати власний Git сервер.

Підсумок

У вас декілька варіантів, як отримати працююче віддалене Git сховище, щоб співпрацювати з іншими або надати доступ до своєї праці. Використання власного серверу дає вам повний контроль та дозволяє налаштовувати ваш власний мережевий екран (firewall), проте такий сервер зазвичай вимагає немало вашого часу для налаштування та підтримки. Якщо ви розмістите ваші дані на серверу хостера, його легко налаштувати та підтримувати. Проте, вам доведеться зберігати код на чужому сервері, та деякі організації цього не дозволяють.

Має бути доволі просто визначити, яке рішення чи комбінація рішень влаштовують вас або вашу організацію.

5 Розподілений Git

Тепер, коли ви маєте віддалений Git репозиторій, налаштований як місце, де всі розробники можуть ділитись своїм кодом, та знайомі з базовими командами Git в локальному процесі роботи, ви дізнаєтесь як використовувати деякі розподілені робочі процеси, що Git надає вам.

В цьому розділі, ви побачите, як працювати з Git в розподіленому середовищі як учасник та інтегратор. Таким чином, ви дізнаєтесь, як успішно вносити код до проекту та зробити це якомога простішим для вас та супроводжувача, а також, як успішно супроводжувати проект з великою кількістю розробників.

Розподілені процеси роботи

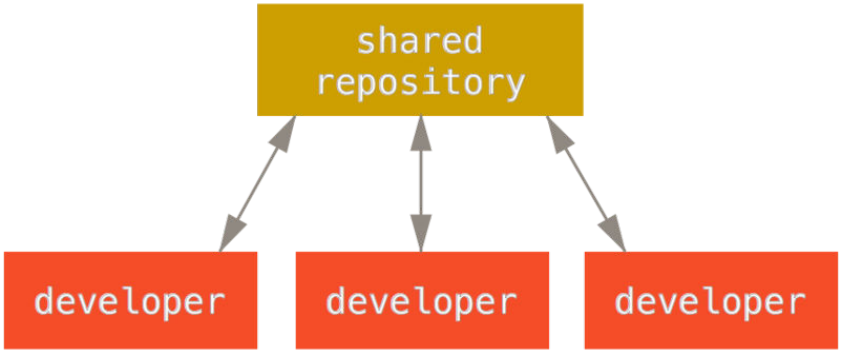
На відміну від Централізованих Систем Керування Версіями (ЦСКВ), розподілена сутність Git дозволяє вам бути набагато більш гнучким щодо того, як розробники співпрацюють над проектами. У централізованих системах кожен розробник є одним з вузлів, які працюють більш менш на рівних над централізованим розподілювачем (hub). У Git, втім, кожен розробник є потенційно і вузлом, і сервером – тобто, кожен розробник може як надсилати код до інших репозиторіїв, як і супроводжувати публічний репозиторій, на якому інші можуть базувати свою роботу, та до якого вони можуть надсилати зміни. Це відкриває безмежні можливості процесу роботи для вашого проекту та/або вашої команди, отже розглянемо декілька поширених парадигм, щоб скористатись цією гнучкістю. Ми дізнаємось про переваги та можливі недоліки кожного методу; ви можете вибрати якийсь один, або змішати та сумістити елементи кожного.

Централізований процес роботи

У централізованих системах, взагалі існує єдина модель співпраці — централізований процес роботи. Один центральний розподілювач, або репозиторій, може приймати код, і всі синхронізують свою роботу з ним. Усі розробники є вузлами – споживачами цього розподілювача – та синхронізуються виключно з ним.

ЗОБРАЖЕННЯ 5-1

Централізований порядок роботи.



Це означає, що, якщо два розробники зроблять клон з розподілювача та обидвоє щось змінять, перший, хто надішле свої зміни назад, зможе це зробити без проблем. Другий розробник змушений зливати роботу першого до надсилання змін, щоб не переписати зміни першого розробника. Ця концепція працює в Git так само, як і в Subversion (чи будь-якій ЦСКВ), і ця модель чудово працює в Git.

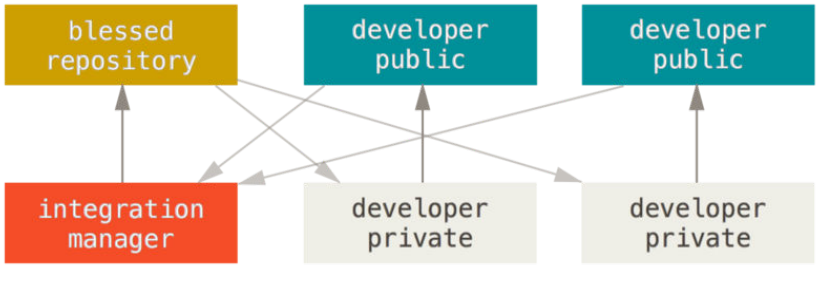
Якщо ви вже звикли до централізованого процесу роботи в компанії чи команді, то можете безклопітно продовжувати його використовувати з Git. Просто налаштуйте окреме сховище, та надайте всім у вашій команді доступ на запис; Git не дозволить користувачам переписувати один одного. Припустимо, Джон та Джесіка обидвоє починають працювати одночасно. Джон завершує свою зміну та надсилає її до сервера. Потім Джесіка намагається надіслати свої зміни, проте сервер відхиляє їх. Їй повідомляють, що вона намагається надіслати зміни, що не є перемотуванням вперед, та їх не можна надсилати, доки вона не отримає та не зіллє зміни. Цей процес роботи є принадним для багатьох, адже ж ця парадигма багато кому знайома та зручна.

Вона також може використовуватись не лише маленькими командами. За допомогою моделі галуження Git, сотні розробників одночасно можуть її використовувати успішно для роботи над одним проектом за допомогою десятків гілок.

Процес роботи з менеджером інтеграції

Оскільки Git дозволяє вам мати декілька віддалених сховищ, можливо мати процес роботи, в якому кожен розробник має право на запис до власного публічного репозиторію та доступ на читання до репозиторіїв інших розробників. Цей сценарій зазвичай включає канонічне сховище, яке представляє “офіційний” проект. Щоб зробити внесок до такого проекту, треба створити власний публічний клон проекту та надіслати зміни до нього. Потім, ви можете надіслати запит до супроводжувача головного проекту, щоб він злив ваші зміни. Супроводжувач тоді може додати ваше сховище як віддалене, перевірити ваші зміни локально, злити їх до своєї гілки, та надіслати до свого репозиторію. Процес працює наступним чином (дивіться **Figure 5-2**):

1. Супроводжувач проекту надсилає зміни до свого власного репозиторію.
2. Розробник клонує цей репозиторій та робить зміни.
3. Розробник надсилає зміни до своєї власної публічної копії.
4. Розробник надсилає супроводжувачу листа, в якому просить злити його зміни.
5. Супроводжувач додає сховище розробника як віддалене та зливає зміни локально.
6. Супроводжувач надсилає злиті зміни до головного репозиторію.



ЗОБРАЖЕННЯ 5-2

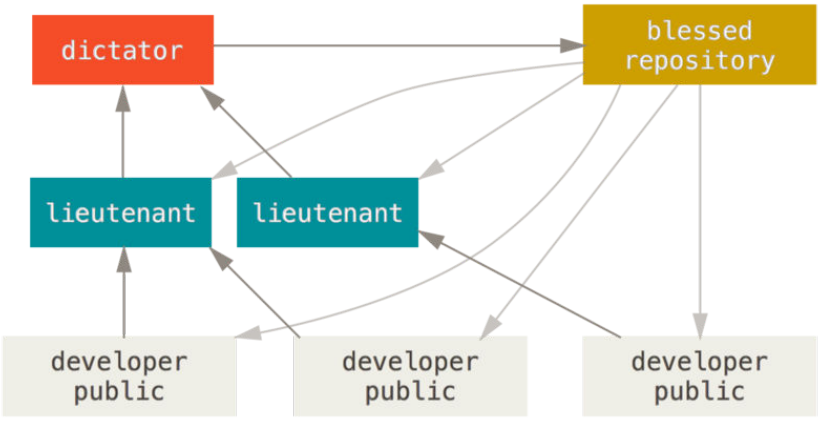
Процес роботи з менеджером інтеграції.

Це дуже поширений процес роботи для інструментів, що базуються на розподілювачах, таких як GitHub та GitLab, в яких дуже легко створити форк проекту та надсилати до нього зміни, які всі можуть бачити. Однією з головних переваг цього підходу є те, що ви можете продовжувати працювати, а супроводжувач головного репозиторію може злити ваші зміни будь-коли. Розробники не змушені чекати на проект, щоб вбудувати свої зміни – кожна сторона може працювати у своєму власному темпі.

Процес роботи з диктатором та лейтенантами

Це варіація процесу роботи з багатьма репозиторіями. Зазвичай він використовується у величезних проектах з сотнями учасників: одним славетним прикладом є ядро Linux. Різноманітні менеджери інтеграції відповідають за окремі частини репозиторію; їх називають лейтенантами. Всі лейтенанти мають одного менеджера інтеграції, відомого як доброзичливий диктатор. Репозиторій доброзичливого диктатора слугує зразковим сховищем, з якого всі розробники мають отримувати зміни. Процес працює наступним чином (дивіться **Figure 5-3**):

1. Звичайні розробники працюють у власній тематичній гілці та перебазовують свою роботу поверх `master`. Гілка `master` та, що в диктатора.
2. Лейтенанти зливають тематичні гілки розробників до власних гілок `master`.
3. Диктатор зливає гілки лейтенантів `master` до гілки диктатора `master`.
4. Диктатор надсилає гілку `master` до зразкового репозиторію, щоб інші розробники могли перебазуватись поверху неї.



ЗОБРАЖЕННЯ 5-3

Процес роботи з доброзичливим диктатором.

Такий процес роботи не є поширеним, проте може бути корисним у дуже великих проектах, або в дуже ієрархічних середовищах. Він дозволяє керівнику проекту (диктатору) делегувати чимало праці та збирати великі частини коду з декількох місць перед тим, як інтегрувати їх.

Підсумок щодо процесів роботи

Це деякі широковживані процеси роботи, які можливі в розподілених системах, таких як Git, проте, як ви бачите, існує багато можливостей, щоб влаштувати реальний процес роботи саме для вас. Тепер, коли ви (сподіваємось) можете визначити, яка комбінація процесів роботи спрацює для вас, ми розглянемо більш конкретні приклади того, як досягнути головних ролей, необхідних для різноманітних процесів. У наступній секції ви дізнаєтесь про декілька поширених шаблонів внесення змін до проекту.

Contributing to a Project

The main difficulty with describing how to contribute to a project is that there are a huge number of variations on how it's done. Because Git is very flexible, people can and do work together in many ways, and it's problematic to describe how you should contribute – every project is a bit different. Some of the variables involved are active contributor count, chosen workflow, your commit access, and possibly the external contribution method.

The first variable is active contributor count – how many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects. For larger companies or projects, the number of developers could be in the thousands, with hundreds or thousands of commits coming in each day. This is important because with more and more developers, you run into more issues with making sure your code applies cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by work that is merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your commits valid?

The next variable is the workflow in use for the project. Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?

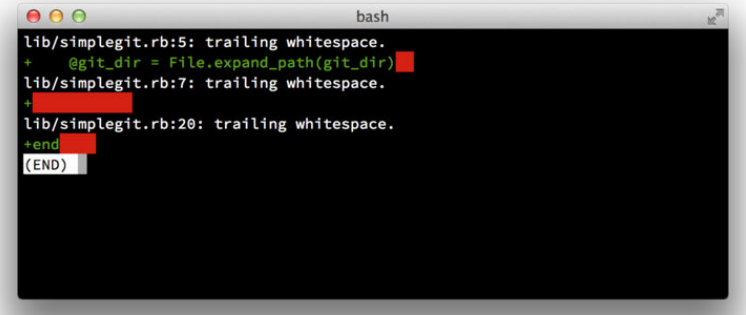
The next issue is your commit access. The workflow required in order to contribute to a project is much different if you have write access to the project than if you don't. If you don't have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to you. We'll cover aspects of each of these in a series of use cases, moving from simple to more complex; you should be able to construct the specific workflows you need in practice from these examples.

Commit Guidelines

Before we start looking at the specific use cases, here's a quick note about commit messages. Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. The Git project provides a document that lays out a number of good tips for creating commits from which to submit patches – you can read it in the Git source code in the `Documentation/SubmittingPatches` file.

First, you don't want to submit any whitespace errors. Git provides an easy way to check for this – before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.



ЗОБРАЖЕННЯ 5-4

Output of `git diff --check`.

If you run that command before committing, you can tell if you're about to commit whitespace issues that may annoy other developers.

Next, try to make each commit a logically separate changeset. If you can, try to make your changes digestible – don't code for a whole weekend on five different issues and then submit them all as one massive commit on Monday. Even if you don't commit during the weekend, use the staging area on Monday to split your work into at least one commit per issue, with a useful message per commit. If some of the changes modify the same file, try to use `git add --patch` to partially stage files (covered in detail in "Інтерактивне індексування"). The project snapshot at the tip of the branch is identical whether you do one commit or five, as long as all the changes are added at some point, so try to make things easier on your fellow developers when they have to review your changes. This approach also makes it easier to pull out or revert one of the changesets if you need to later. "Переписування історії" describes a number of useful Git tricks for rewriting history and interactively staging files – use these tools to help craft a clean and understandable history before sending the work to someone else.

The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier. As a general rule, your messages should start with a single line that's no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation. The Git project requires that the more detailed explanation include your motivation for the change and contrast its implementation with previous behavior – this is a good guideline to follow. It's also a good idea to use the imperative present tense in these messages. In other words, use commands. Instead of "I added tests for"

or “Adding tests for,” use “Add tests for.” Here is a template originally written by Tim Pope:

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to
about 72 characters or so.  In some contexts, the first
line is treated as the subject of an email and the rest of
the text as the body.  The blank line separating the
summary from the body is critical (unless you omit the body
entirely); tools like rebase can get confused if you run
the two together.

Further paragraphs come after blank lines.

  - Bullet points are okay, too

  - Typically a hyphen or asterisk is used for the bullet,
    preceded by a single space, with blank lines in
    between, but conventions vary here
```

If all your commit messages look like this, things will be a lot easier for you and the developers you work with. The Git project has well-formatted commit messages – try running `git log --no-merges` there to see what a nicely formatted project-commit history looks like.

In the following examples, and throughout most of this book, for the sake of brevity this book doesn’t have nicely-formatted messages like this; instead, we use the `-m` option to `git commit`. Do as we say, not as we do.

Private Small Team

The simplest setup you’re likely to encounter is a private project with one or two other developers. “Private,” in this context, means closed-source – not accessible to the outside world. You and the other developers all have push access to the repository.

In this environment, you can follow a workflow similar to what you might do when using Subversion or another centralized system. You still get the advantages of things like offline committing and vastly simpler branching and merging, but the workflow can be very similar; the main difference is that merges happen client-side rather than on the server at commit time. Let’s see what it might look like when two developers start to work together with a shared repository. The first developer, John, clones the repository, makes a change, and commits locally. (The protocol messages have been replaced with `...` in these examples to shorten them somewhat.)

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The second developer, Jessica, does the same thing – clones the repository and commits a change:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Now, Jessica pushes her work up to the server:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John tries to push his change up, too:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

John isn't allowed to push because Jessica has pushed in the meantime. This is especially important to understand if you're used to Subversion, because you'll notice that the two developers didn't edit the same file. Although

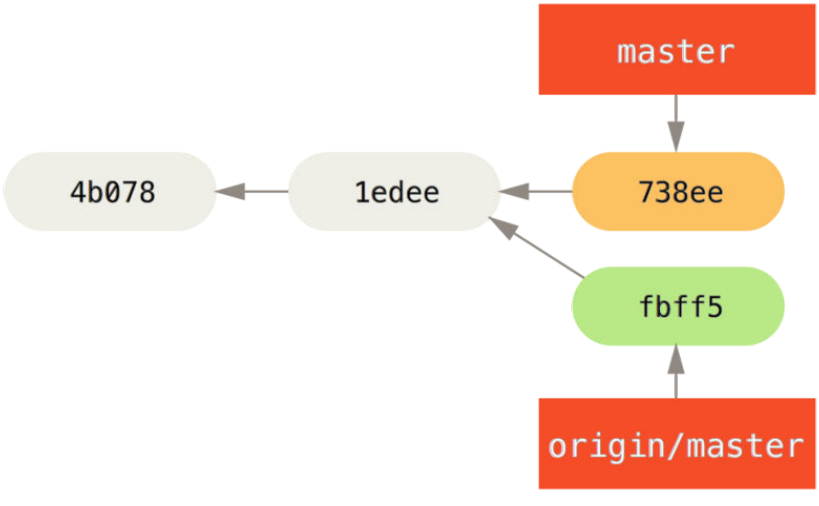
Subversion automatically does such a merge on the server if different files are edited, in Git you must merge the commits locally. John has to fetch Jessica's changes and merge them in before he will be allowed to push:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

At this point, John's local repository looks something like this:

ЗОБРАЖЕННЯ 5-5

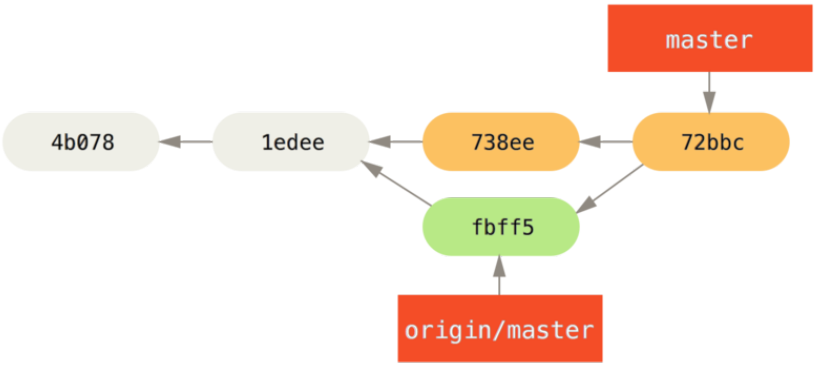
John's divergent history.



John has a reference to the changes Jessica pushed up, but he has to merge them into his own work before he is allowed to push:

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

The merge goes smoothly – John's commit history now looks like this:



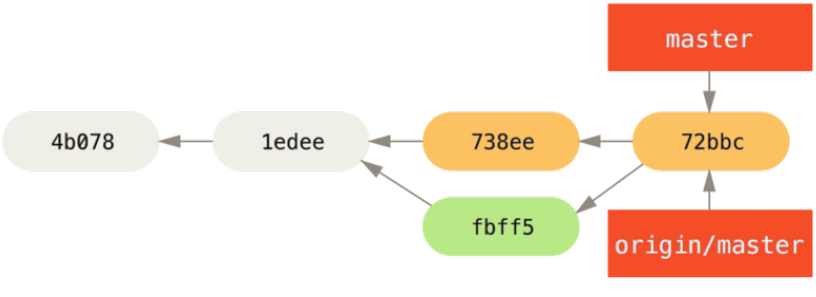
ЗОБРАЖЕННЯ 5-6

John's repository after merging `origin/master`.

Now, John can test his code to make sure it still works properly, and then he can push his new merged work up to the server:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

Finally, John's commit history looks like this:



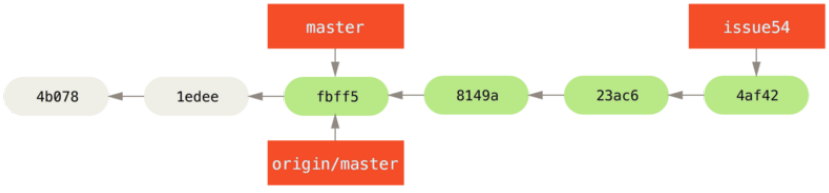
ЗОБРАЖЕННЯ 5-7

John's history after pushing to the `origin` server.

In the meantime, Jessica has been working on a topic branch. She's created a topic branch called `issue54` and done three commits on that branch. She hasn't fetched John's changes yet, so her commit history looks like this:

ЗОБРАЖЕННЯ 5-8

Jessica's topic branch.



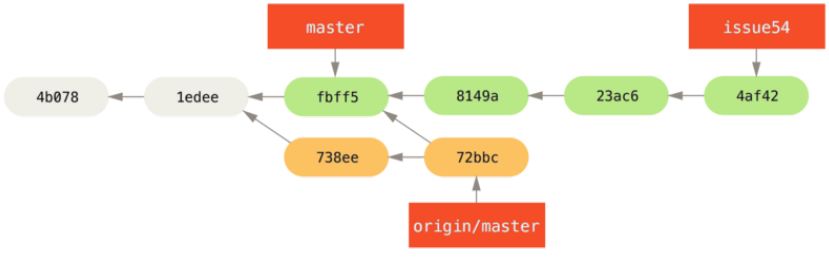
Jessica wants to sync up with John, so she fetches:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

That pulls down the work John has pushed up in the meantime. Jessica's history now looks like this:

ЗОБРАЖЕННЯ 5-9

Jessica's history after fetching John's changes.



Jessica thinks her topic branch is ready, but she wants to know what she has to merge into her work so that she can push. She runs `git log` to find out:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   removed invalid default value
```

The `issue54..origin/master` syntax is a log filter that asks Git to only show the list of commits that are on the latter branch (in this case `origin/master`) that are not on the first branch (in this case `issue54`). We'll go over this syntax in detail in **"Інтервали комітів"**.

For now, we can see from the output that there is a single commit that John has made that Jessica has not merged in. If she merges `origin/master`, that is the single commit that will modify her local work.

Now, Jessica can merge her topic work into her master branch, merge John's work (`origin/master`) into her `master` branch, and then push back to the server again. First, she switches back to her master branch to integrate all this work:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

She can merge either `origin/master` or `issue54` first – they're both upstream, so the order doesn't matter. The end snapshot should be identical no matter which order she chooses; only the history will be slightly different. She chooses to merge in `issue54` first:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

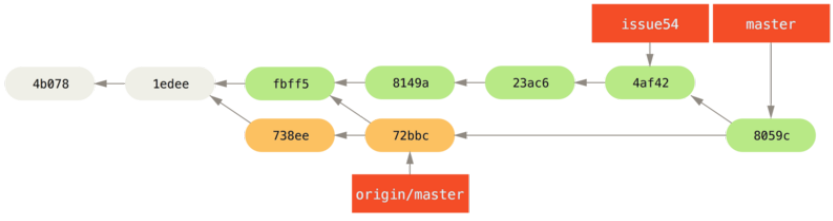
No problems occur; as you can see it was a simple fast-forward. Now Jessica merges in John's work (`origin/master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Everything merges cleanly, and Jessica's history looks like this:

ЗОБРАЖЕННЯ 5-10

Jessica's history after merging John's changes.



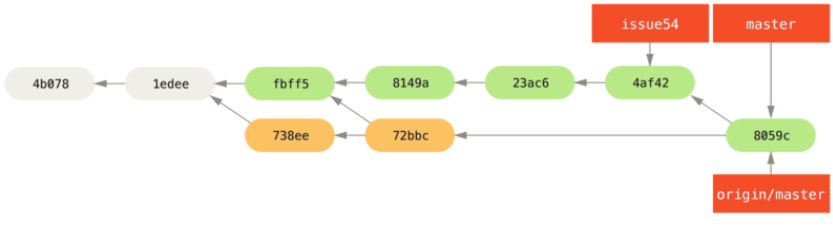
Now `origin/master` is reachable from Jessica's `master` branch, so she should be able to successfully push (assuming John hasn't pushed again in the meantime):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

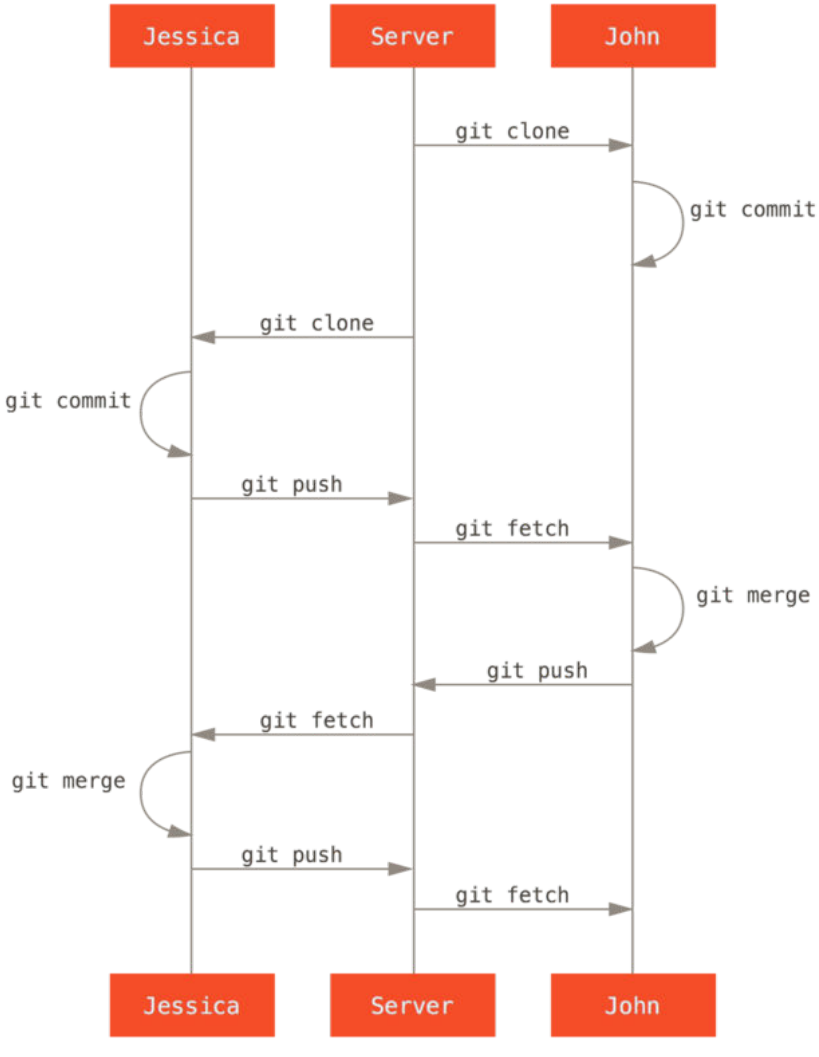
Each developer has committed a few times and merged each other's work successfully.

ЗОБРАЖЕННЯ 5-11

Jessica's history after pushing all changes back to the server.



That is one of the simplest workflows. You work for a while, generally in a topic branch, and merge into your master branch when it's ready to be integrated. When you want to share that work, you merge it into your own master branch, then fetch and merge `origin/master` if it has changed, and finally push to the `master` branch on the server. The general sequence is something like this:



ЗОБРАЖЕННЯ 5-12

General sequence of events for a simple multiple-developer Git workflow.

Private Managed Team

In this next scenario, you'll look at contributor roles in a larger private group. You'll learn how to work in an environment where small groups collaborate on features and then those team-based contributions are integrated by another party.

Let's say that John and Jessica are working together on one feature, while Jessica and Josie are working on a second. In this case, the company is using a

type of integration-manager workflow where the work of the individual groups is integrated only by certain engineers, and the `master` branch of the main repo can be updated only by those engineers. In this scenario, all work is done in team-based branches and pulled together by the integrators later.

Let's follow Jessica's workflow as she works on her two features, collaborating in parallel with two different developers in this environment. Assuming she already has her repository cloned, she decides to work on `featureA` first. She creates a new branch for the feature and does some work on it there:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

At this point, she needs to share her work with John, so she pushes her `featureA` branch commits up to the server. Jessica doesn't have push access to the `master` branch – only the integrators do – so she has to push to another branch in order to collaborate with John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica e-mails John to tell him that she's pushed some work into a branch named `featureA` and he can look at it now. While she waits for feedback from John, Jessica decides to start working on `featureB` with Josie. To begin, she starts a new feature branch, basing it off the server's `master` branch:

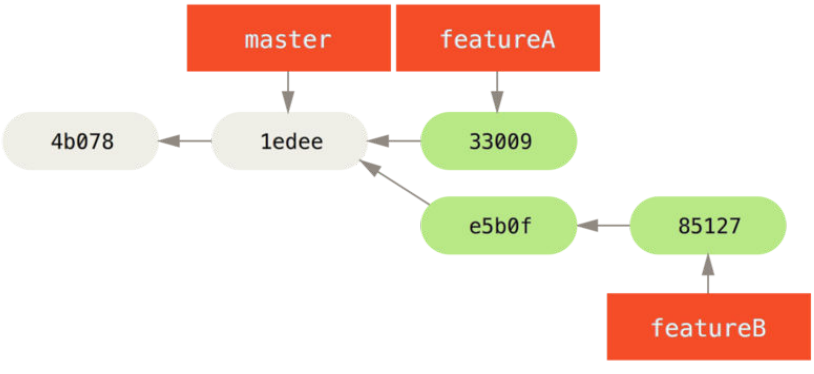
```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Now, Jessica makes a couple of commits on the `featureB` branch:

```
$ 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(-)
```

Jessica's repository looks like this:



ЗОБРАЖЕННЯ 5-13

Jessica's initial commit history.

She's ready to push up her work, but gets an e-mail from Josie that a branch with some initial work on it was already pushed to the server as `featureBee`. Jessica first needs to merge those changes in with her own before she can push to the server. She can then fetch Josie's changes down with `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica can now merge this into the work she did with `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

There is a bit of a problem – she needs to push the merged work in her `featureB` branch to the `featureBee` branch on the server. She can do so by specifying the local branch followed by a colon (:) followed by the remote branch to the `git push` command:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

This is called a *refspec*. See **"The Refspec"** for a more detailed discussion of Git refspecs and different things you can do with them. Also notice the `-u` flag; this is short for `--set-upstream`, which configures the branches for easier pushing and pulling later.

Next, John e-mails Jessica to say he's pushed some changes to the `featureA` branch and asks her to verify them. She runs a `git fetch` to pull down those changes:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Then, she can see what has been changed with `git log`:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

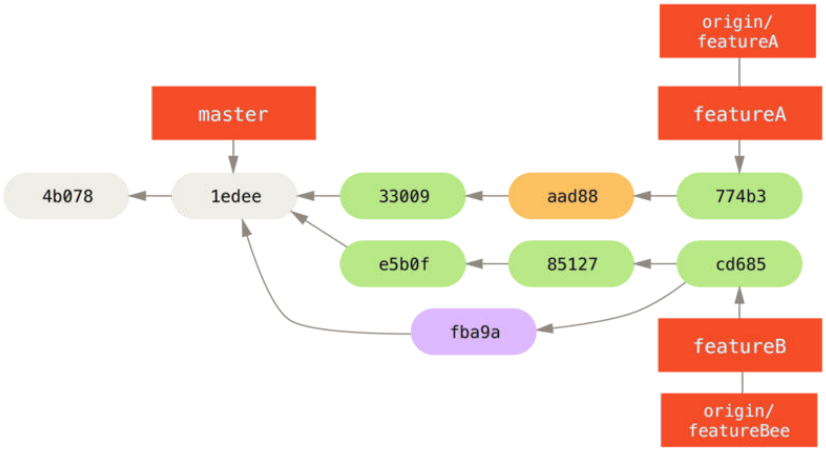
Finally, she merges John's work into her own `featureA` branch:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica wants to tweak something, so she commits again and then pushes this back up to the server:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Jessica's commit history now looks something like this:



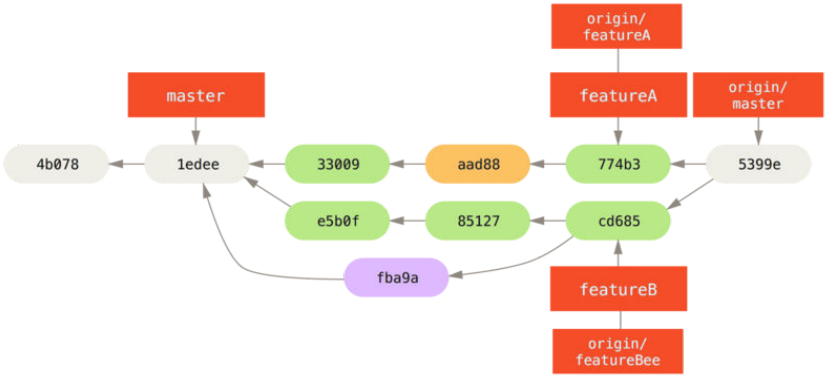
ЗОБРАЖЕННЯ 5-14

Jessica's history after committing on a feature branch.

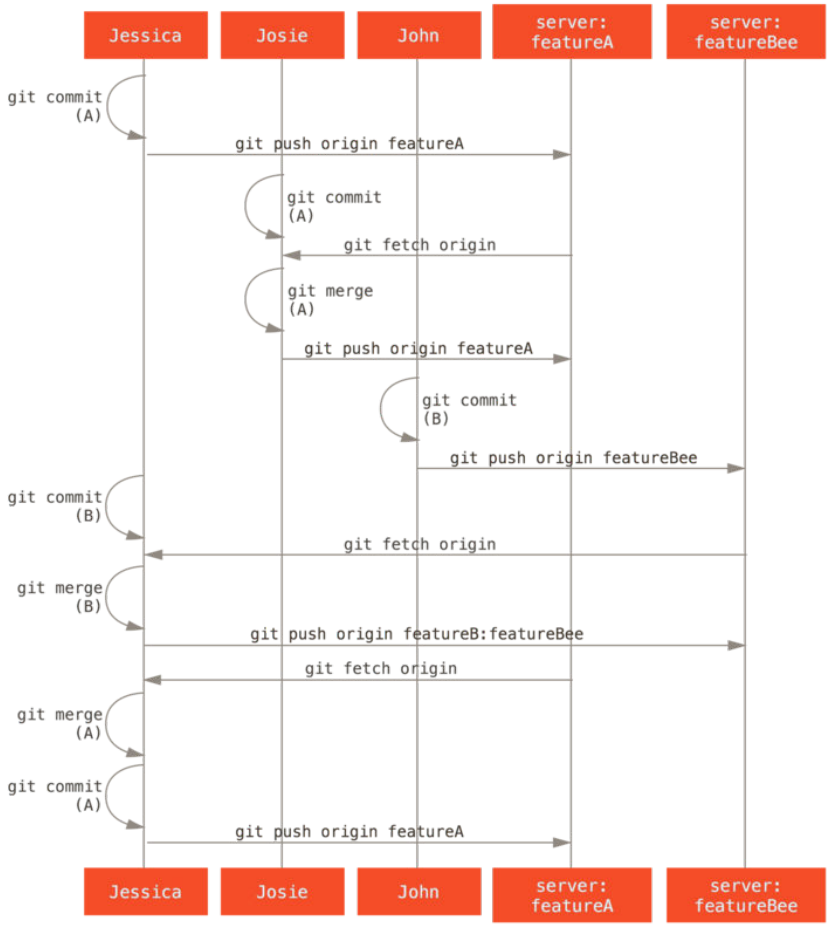
Jessica, Josie, and John inform the integrators that the `featureA` and `featureBee` branches on the server are ready for integration into the mainline. Afterthe integrators merge these branches into the mainline, a fetch will bring down the new merge commit, making the history look like this:

ЗОБРАЖЕННЯ 5-15

Jessica's history after merging both her topic branches.



Many groups switch to Git because of this ability to have multiple teams working in parallel, merging the different lines of work late in the process. The ability of smaller subgroups of a team to collaborate via remote branches without necessarily having to involve or impede the entire team is a huge benefit of Git. The sequence for the workflow you saw here is something like this:



ЗОБРАЖЕННЯ 5-16

Basic sequence of this managed-team workflow.

Forked Public Project

Contributing to public projects is a bit different. Because you don't have the permissions to directly update branches on the project, you have to get the work to the maintainers some other way. This first example describes contributing via forking on Git hosts that support easy forking. Many hosting sites support this (including GitHub, BitBucket, Google Code, repo.or.cz, and others), and many project maintainers expect this style of contribution. The next section deals with projects that prefer to accept contributed patches via e-mail.

First, you'll probably want to clone the main repository, create a topic branch for the patch or patch series you're planning to contribute, and do your work there. The sequence looks basically like this:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

You may want to use `rebase -i` to squash your work down to a single commit, or rearrange the work in the commits to make the patch easier for the maintainer to review – see "Переписування історії" for more information about interactive rebasing.

When your branch work is finished and you're ready to contribute it back to the maintainers, go to the original project page and click the "Fork" button, creating your own writable fork of the project. You then need to add in this new repository URL as a second remote, in this case named `myfork`:

```
$ git remote add myfork (url)
```

Then you need to push your work up to it. It's easiest to push the topic branch you're working on up to your repository, rather than merging into your master branch and pushing that up. The reason is that if the work isn't accepted or is cherry picked, you don't have to rewind your master branch. If the maintainers merge, rebase, or cherry-pick your work, you'll eventually get it back via pulling from their repository anyhow:

```
$ git push -u myfork featureA
```

When your work has been pushed up to your fork, you need to notify the maintainer. This is often called a pull request, and you can either generate it via the website – GitHub has its own Pull Request mechanism that we'll go over in **Chapter 6** – or you can run the `git request-pull` command and e-mail the output to the project maintainer manually.

The `request-pull` command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and outputs a summary of all the changes you're asking to be pulled in. For instance, if Jessica wants to send John a pull request, and she's done two commits on the topic branch she just pushed up, she can run this:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

The output can be sent to the maintainer–it tells them where the work was branched from, summarizes the commits, and tells where to pull this work from.

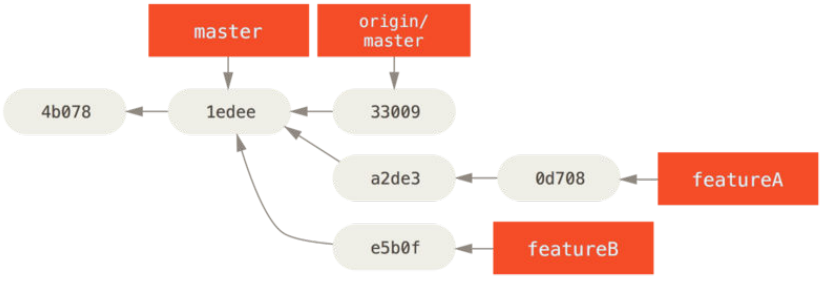
On a project for which you're not the maintainer, it's generally easier to have a branch like `master` always track `origin/master` and to do your work in topic branches that you can easily discard if they're rejected. Having work themes isolated into topic branches also makes it easier for you to rebase your work if the tip of the main repository has moved in the meantime and your commits no longer apply cleanly. For example, if you want to submit a second topic of work to the project, don't continue working on the topic branch you just pushed up – start over from the main repository's `master` branch:

```
$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin
```

Now, each of your topics is contained within a silo – similar to a patch queue – that you can rewrite, rebase, and modify without the topics interfering or interdepending on each other, like so:

ЗОБРАЖЕННЯ 5-17

Initial commit history with `featureB` work.



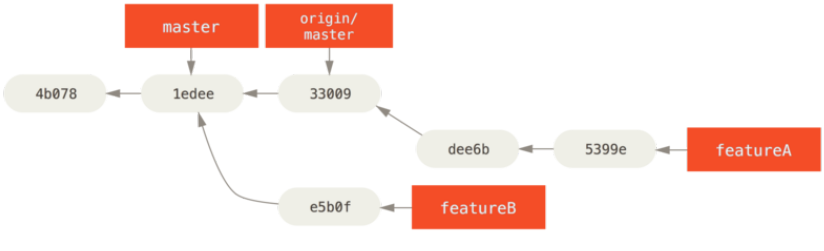
Let's say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges. In this case, you can try to rebase that branch on top of `origin/master`, resolve the conflicts for the maintainer, and then resubmit your changes:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

This rewrites your history to now look like **Figure 5-18**.

ЗОБРАЖЕННЯ 5-18

Commit history after `featureA` work.



Because you rebased the branch, you have to specify the `-f` to your push command in order to be able to replace the `featureA` branch on the server with a commit that isn't a descendant of it. An alternative would be to push this new work to a different branch on the server (perhaps called `featureAv2`).

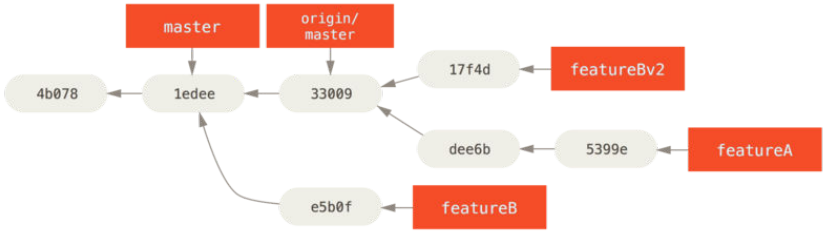
Let's look at one more possible scenario: the maintainer has looked at work in your second branch and likes the concept but would like you to change an implementation detail. You'll also take this opportunity to move the work to be

based off the project's current `master` branch. You start a new branch based off the current `origin/master` branch, squash the `featureB` changes there, resolve any conflicts, make the implementation change, and then push that up as a new branch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

The `--squash` option takes all the work on the merged branch and squashes it into one changeset producing the repository state as if a real merge happened, without actually making a merge commit. This means your future commit will have one parent only and allows you to introduce all the changes from another branch and then make more changes before recording the new commit. Also the `--no-commit` option can be useful to delay the merge commit in case of the default merge process.

Now you can send the maintainer a message that you've made the requested changes and they can find those changes in your `featureBv2` branch.



ЗОБРАЖЕННЯ 5-19

Commit history after `featureBv2` work.

Public Project over E-Mail

Many projects have established procedures for accepting patches – you'll need to check the specific rules for each project, because they will differ. Since there are several older, larger projects which accept patches via a developer mailing list, we'll go over an example of that now.

The workflow is similar to the previous use case – you create topic branches for each patch series you work on. The difference is how you submit them to the project. Instead of forking the project and pushing to your own writable ver-

sion, you generate e-mail versions of each commit series and e-mail them to the developer mailing list:

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

Now you have two commits that you want to send to the mailing list. You use `git format-patch` to generate the mbox-formatted files that you can e-mail to the list – it turns each commit into an e-mail message with the first line of the commit message as the subject and the rest of the message plus the patch that the commit introduces as the body. The nice thing about this is that applying a patch from an e-mail generated with `format-patch` preserves all the commit information properly.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

The `format-patch` command prints out the names of the patch files it creates. The `-M` switch tells Git to look for renames. The files end up looking like this:

```
$ cat 0001-add-limit-to-log-function.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] add limit to log function

Limit log functionality to the first 20

---
 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
```

You can also edit these patch files to add more information for the e-mail list that you don't want to show up in the commit message. If you add text between the `---` line and the beginning of the patch (the `diff --git` line), then developers can read it; but applying the patch excludes it.

To e-mail this to a mailing list, you can either paste the file into your e-mail program or send it via a command-line program. Pasting the text often causes formatting issues, especially with "smarter" clients that don't preserve newlines and other whitespace appropriately. Luckily, Git provides a tool to help you send properly formatted patches via IMAP, which may be easier for you. We'll demonstrate how to send a patch via Gmail, which happens to be the e-mail agent we know best; you can read detailed instructions for a number of mail programs at the end of the aforementioned `Documentation/SubmittingPatches` file in the Git source code.

First, you need to set up the imap section in your `~/.gitconfig` file. You can set each value separately with a series of `git config` commands, or you can add them manually, but in the end your config file should look something like this:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

If your IMAP server doesn't use SSL, the last two lines probably aren't necessary, and the host value will be `imap://` instead of `imaps://`. When that is set up, you can use `git imap-send` to place the patch series in the Drafts folder of the specified IMAP server:

```
$ 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
```

At this point, you should be able to go to your Drafts folder, change the To field to the mailing list you're sending the patch to, possibly CC the maintainer or person responsible for that section, and send it off.

You can also send the patches through an SMTP server. As before, you can set each value separately with a series of `git config` commands, or you can add them manually in the sendemail section in your `~/.gitconfig` file:

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

After this is done, you can use `git send-email` to send your patches:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.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
```

Then, Git spits out a bunch of log information looking something like this for each patch you're sending:

```
(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
```

Summary

This section has covered a number of common workflows for dealing with several very different types of Git projects you're likely to encounter, and introduced a couple of new tools to help you manage this process. Next, you'll see how to work the other side of the coin: maintaining a Git project. You'll learn how to be a benevolent dictator or integration manager.

Maintaining a Project

In addition to knowing how to effectively contribute to a project, you'll likely need to know how to maintain one. This can consist of accepting and applying patches generated via `format-patch` and e-mailed to you, or integrating changes in remote branches for repositories you've added as remotes to your project. Whether you maintain a canonical repository or want to help by verifying or approving patches, you need to know how to accept work in a way that is clearest for other contributors and sustainable by you over the long run.

Working in Topic Branches

When you're thinking of integrating new work, it's generally a good idea to try it out in a topic branch – a temporary branch specifically made to try out that new work. This way, it's easy to tweak a patch individually and leave it if it's not working until you have time to come back to it. If you create a simple branch name based on the theme of the work you're going to try, such as `ruby_client` or something similarly descriptive, you can easily remember it if you have to abandon it for a while and come back later. The maintainer of the Git project tends to namespace these branches as well – such as `sc/ruby_client`, where `sc` is short for the person who contributed the work. As you'll remember, you can create the branch based off your master branch like this:

```
$ git branch sc/ruby_client master
```

Or, if you want to also switch to it immediately, you can use the `checkout -b` option:

```
$ git checkout -b sc/ruby_client master
```

Now you're ready to add your contributed work into this topic branch and determine if you want to merge it into your longer-term branches.

Applying Patches from E-mail

If you receive a patch over e-mail that you need to integrate into your project, you need to apply the patch in your topic branch to evaluate it. There are two ways to apply an e-mailed patch: with `git apply` or with `git am`.

APPLYING A PATCH WITH APPLY

If you received the patch from someone who generated it with the `git diff` or a Unix `diff` command (which is not recommended; see the next section), you can apply it with the `git apply` command. Assuming you saved the patch at `/tmp/patch-ruby-client.patch`, you can apply the patch like this:

```
$ git apply /tmp/patch-ruby-client.patch
```

This modifies the files in your working directory. It's almost identical to running a `patch -p1` command to apply the patch, although it's more paranoid and accepts fewer fuzzy matches than patch. It also handles file adds, deletes, and renames if they're described in the `git diff` format, which `patch` won't do. Finally, `git apply` is an "apply all or abort all" model where either everything is applied or nothing is, whereas `patch` can partially apply patchfiles, leaving your working directory in a weird state. `git apply` is overall much more conservative than `patch`. It won't create a commit for you – after running it, you must stage and commit the changes introduced manually.

You can also use git apply to see if a patch applies cleanly before you try actually applying it – you can run `git apply --check` with the 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
```

If there is no output, then the patch should apply cleanly. This command also exits with a non-zero status if the check fails, so you can use it in scripts if you want.

APPLYING A PATCH WITH AM

If the contributor is a Git user and was good enough to use the `format-patch` command to generate their patch, then your job is easier because the patch contains author information and a commit message for you. If you can, encourage your contributors to use `format-patch` instead of `diff` to generate patches for you. You should only have to use `git apply` for legacy patches and things like that.

To apply a patch generated by `format-patch`, you use `git am`. Technically, `git am` is built to read an mbox file, which is a simple, plain-text format for storing one or more e-mail messages in one text file. It looks something like this:

```
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
```

This is the beginning of the output of the format-patch command that you saw in the previous section. This is also a valid mbox e-mail format. If someone has e-mailed you the patch properly using git send-email, and you download that into an mbox format, then you can point git am to that mbox file, and it will start applying all the patches it sees. If you run a mail client that can save several e-mails out in mbox format, you can save entire patch series into a file and then use git am to apply them one at a time.

However, if someone uploaded a patch file generated via `format-patch` to a ticketing system or something similar, you can save the file locally and then pass that file saved on your disk to `git am` to apply it:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

You can see that it applied cleanly and automatically created the new commit for you. The author information is taken from the e-mail's `From` and `Date` headers, and the message of the commit is taken from the `Subject` and body (before the patch) of the e-mail. For example, if this patch was applied from the mbox example above, the commit generated would look something like this:

```
$ 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
```

The `Commit` information indicates the person who applied the patch and the time it was applied. The `Author` information is the individual who originally created the patch and when it was originally created.

But it's possible that the patch won't apply cleanly. Perhaps your main branch has diverged too far from the branch the patch was built from, or the patch depends on another patch you haven't applied yet. In that case, the `git am` process will fail and ask you what you want to do:

```
$ git am 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
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. You solve this issue much the same way – edit the file to resolve the conflict, stage the new file, and then run `git am --resolved` to continue to the next patch:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

If you want Git to try a bit more intelligently to resolve the conflict, you can pass a `-3` option to it, which makes Git attempt a three-way merge. This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository. If you do have that commit – if the patch was

based on a public commit – then the `-3` option is generally much smarter about applying a conflicting patch:

```
$ 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.
```

In this case, this patch had already been applied. Without the `-3` option, it looks like a conflict.

If you're applying a number of patches from an mbox, you can also run the `am` command in interactive mode, which stops at each patch it finds and asks if you want to apply it:

```
$ 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
```

This is nice if you have a number of patches saved, because you can view the patch first if you don't remember what it is, or not apply the patch if you've already done so.

When all the patches for your topic are applied and committed into your branch, you can choose whether and how to integrate them into a longer-running branch.

Checking Out Remote Branches

If your contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, you can add them as a remote and do merges locally.

For instance, if Jessica sends you an e-mail saying that she has a great new feature in the `ruby-client` branch of her repository, you can test it by adding the remote and checking out that branch locally:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

If she e-mails you again later with another branch containing another great feature, you can fetch and check out because you already have the remote setup.

This is most useful if you're working with a person consistently. If someone only has a single patch to contribute once in a while, then accepting it over e-mail may be less time consuming than requiring everyone to run their own server and having to continually add and remove remotes to get a few patches. You're also unlikely to want to have hundreds of remotes, each for someone who contributes only a patch or two. However, scripts and hosted services may make this easier – it depends largely on how you develop and how your contributors develop.

The other advantage of this approach is that you get the history of the commits as well. Although you may have legitimate merge issues, you know where in your history their work is based; a proper three-way merge is the default rather than having to supply a `-3` and hope the patch was generated off a public commit to which you have access.

If you aren't working with a person consistently but still want to pull from them in this way, you can provide the URL of the remote repository to the `git pull` command. This does a one-time pull and doesn't save the URL as a remote reference:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

Determining What Is Introduced

Now you have a topic branch that contains contributed work. At this point, you can determine what you'd like to do with it. This section revisits a couple of commands so you can see how you can use them to review exactly what you'll be introducing if you merge this into your main branch.

It's often helpful to get a review of all the commits that are in this branch but that aren't in your master branch. You can exclude commits in the master branch by adding the `--not` option before the branch name. This does the same thing as the `master..contrib` format that we used earlier. For example,

if your contributor sends you two patches and you create a branch called `contrib` and applied those patches there, you can run this:

```
$ 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
```

To see what changes each commit introduces, remember that you can pass the `-p` option to `git log` and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

```
$ git diff master
```

This command gives you a diff, but it may be misleading. If your `master` branch has moved forward since you created the topic branch from it, then you'll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you're on and the snapshot of the last commit on the `master` branch. For example, if you've added a line in a file on the `master` branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If `master` is a direct ancestor of your topic branch, this isn't a problem; but if the two histories have diverged, the diff will look like you're adding all the new stuff in your topic branch and removing everything unique to the `master` branch.

What you really want to see are the changes added to the topic branch – the work you'll introduce if you merge this branch with master. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the master branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

However, that isn't convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax. In the context of the `diff` command, you can put three periods after another branch to do a `diff` between the last commit of the branch you're on and its common ancestor with another branch:

```
$ git diff master...contrib
```

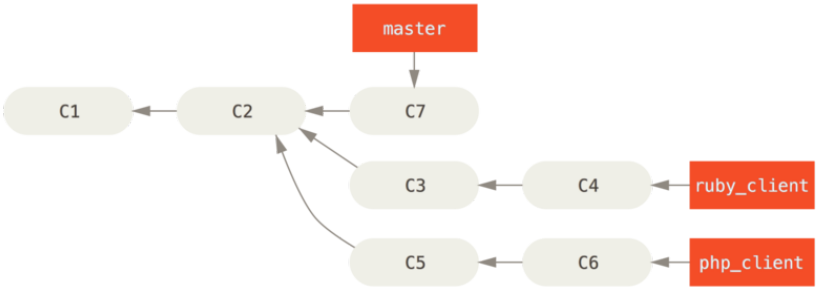
This command shows you only the work your current topic branch has introduced since its common ancestor with master. That is a very useful syntax to remember.

Integrating Contributed Work

When all the work in your topic branch is ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall workflow do you want to use to maintain your project? You have a number of choices, so we'll cover a few of them.

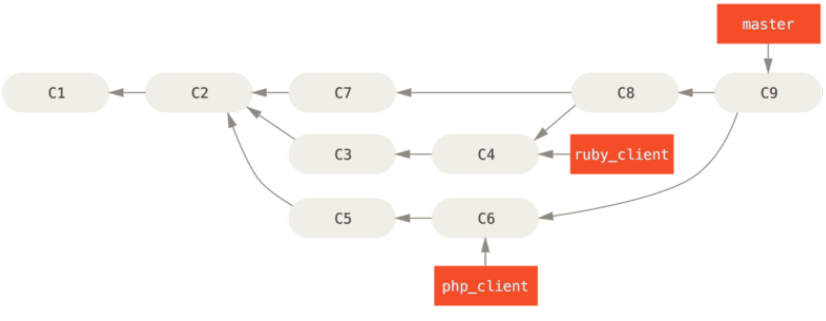
MERGING WORKFLOWS

One simple workflow merges your work into your `master` branch. In this scenario, you have a `master` branch that contains basically stable code. When you have work in a topic branch that you've done or that someone has contributed and you've verified, you merge it into your master branch, delete the topic branch, and then continue the process. If we have a repository with work in two branches named `ruby_client` and `php_client` that looks like **Figure 5-20** and merge `ruby_client` first and then `php_client` next, then your history will end up looking like **Figure 5-21**.



ЗОБРАЖЕННЯ 5-20

History with several topic branches.



ЗОБРАЖЕННЯ 5-21

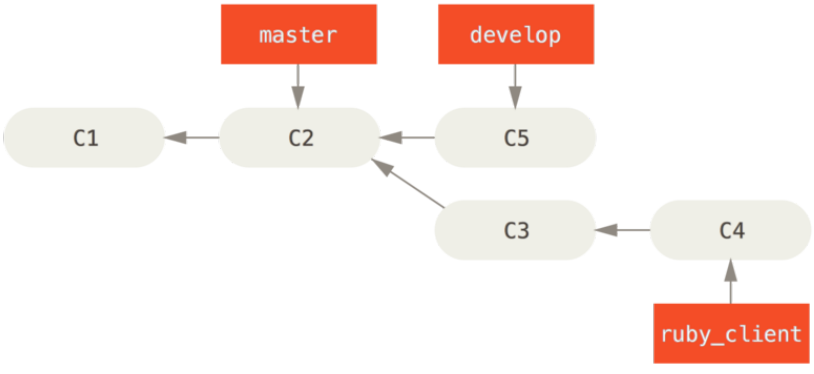
After a topic branch merge.

That is probably the simplest workflow, but it can possibly be problematic if you're dealing with larger or more stable projects where you want to be really careful about what you introduce.

If you have a more important project, you might want to use a two-phase merge cycle. In this scenario, you have two long-running branches, `master` and `develop`, in which you determine that `master` is updated only when a very stable release is cut and all new code is integrated into the `develop` branch. You regularly push both of these branches to the public repository. Each time you have a new topic branch to merge in (**Figure 5-22**), you merge it into `develop` (**Figure 5-23**); then, when you tag a release, you fast-forward `master` to wherever the now-stable `develop` branch is (**Figure 5-24**).

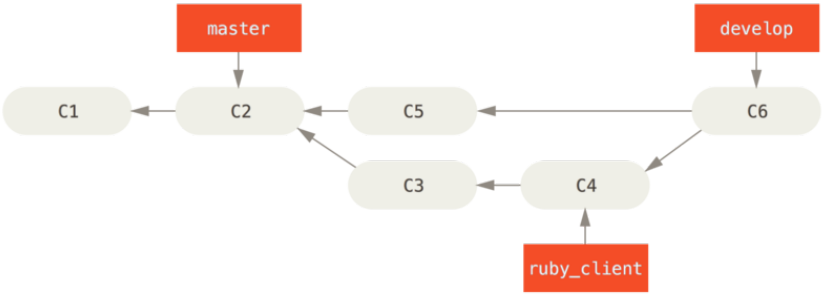
ЗОБРАЖЕННЯ 5-22

Before a topic branch merge.



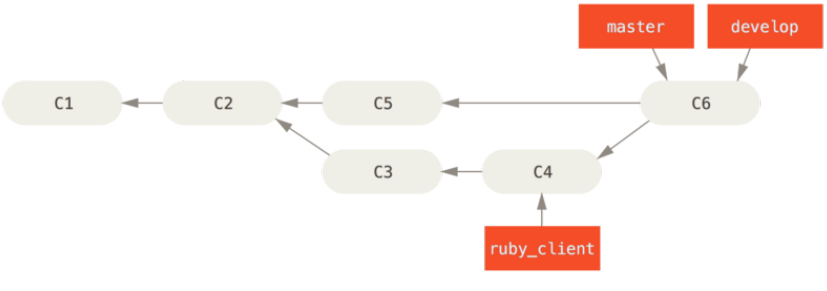
ЗОБРАЖЕННЯ 5-23

After a topic branch merge.



ЗОБРАЖЕННЯ 5-24

After a project release.

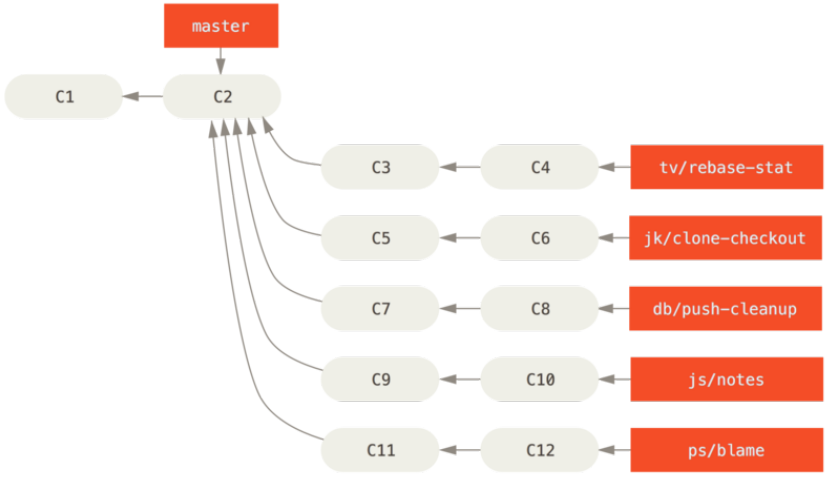


This way, when people clone your project's repository, they can either check out master to build the latest stable version and keep up to date on that easily, or they can check out develop, which is the more cutting-edge stuff. You can

also continue this concept, having an integrate branch where all the work is merged together. Then, when the codebase on that branch is stable and passes tests, you merge it into a develop branch; and when that has proven itself stable for a while, you fast-forward your master branch.

LARGE-MERGING WORKFLOWS

The Git project has four long-running branches: `master`, `next`, and `pu` (proposed updates) for new work, and `maint` for maintenance backports. When new work is introduced by contributors, it's collected into topic branches in the maintainer's repository in a manner similar to what we've described (see **Figure 5-25**). At this point, the topics are evaluated to determine whether they're safe and ready for consumption or whether they need more work. If they're safe, they're merged into `next`, and that branch is pushed up so everyone can try the topics integrated together.



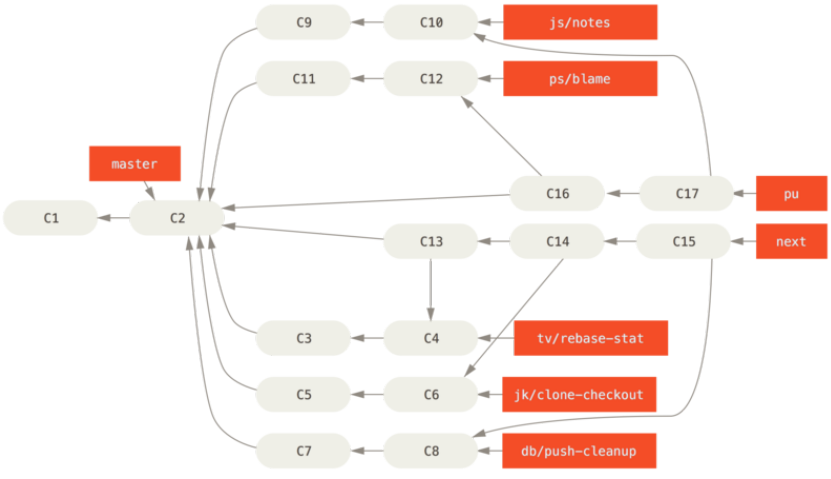
ЗОБРАЖЕННЯ 5-25

Managing a complex series of parallel contributed topic branches.

If the topics still need work, they're merged into `pu` instead. When it's determined that they're totally stable, the topics are re-merged into `master` and are then rebuilt from the topics that were in `next` but didn't yet graduate to `master`. This means `master` almost always moves forward, `next` is rebased occasionally, and `pu` is rebased even more often:

ЗОБРАЖЕННЯ 5-26

Merging contributed topic branches into long-term integration branches.

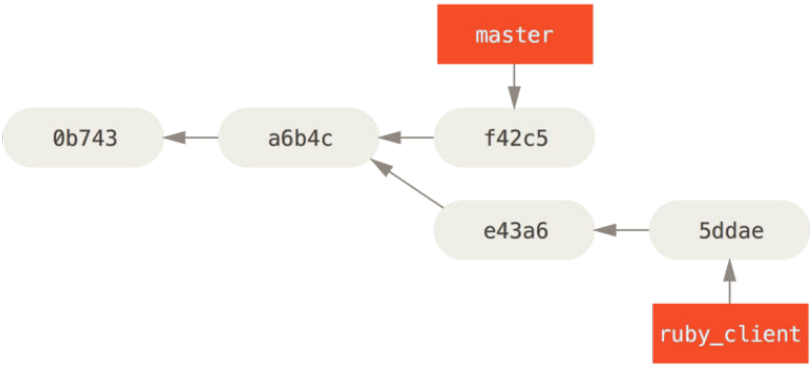


When a topic branch has finally been merged into `master`, it's removed from the repository. The Git project also has a `maint` branch that is forked off from the last release to provide backported patches in case a maintenance release is required. Thus, when you clone the Git repository, you have four branches that you can check out to evaluate the project in different stages of development, depending on how cutting edge you want to be or how you want to contribute; and the maintainer has a structured workflow to help them vet new contributions.

REBASING AND CHERRY PICKING WORKFLOWS

Other maintainers prefer to rebase or cherry-pick contributed work on top of their master branch, rather than merging it in, to keep a mostly linear history. When you have work in a topic branch and have determined that you want to integrate it, you move to that branch and run the rebase command to rebuild the changes on top of your current master (or `develop`, and so on) branch. If that works well, you can fast-forward your `master` branch, and you'll end up with a linear project history.

The other way to move introduced work from one branch to another is to cherry-pick it. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run rebase. For example, suppose you have a project that looks like this:



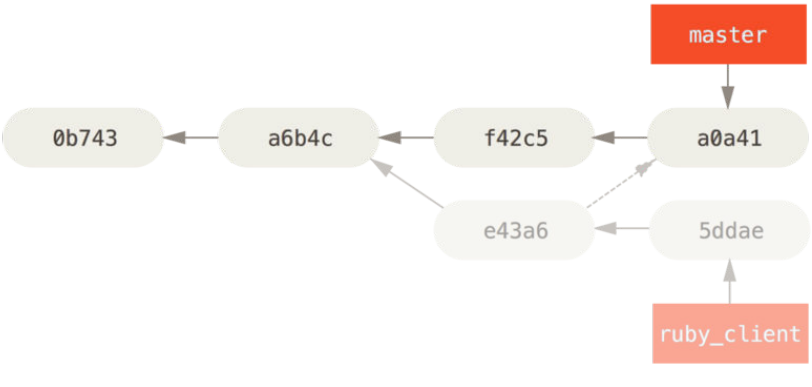
ЗОБРАЖЕННЯ 5-27

Example history before a cherry-pick.

If you want to pull commit e43a6 into your master branch, you can run

```
$ 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(-)
```

This pulls the same change introduced in e43a6, but you get a new commit SHA-1 value, because the date applied is different. Now your history looks like this:



ЗОБРАЖЕННЯ 5-28

History after cherry-picking a commit on a topic branch.

Now you can remove your topic branch and drop the commits you didn't want to pull in.

RERERE

If you're doing lots of merging and rebasing, or you're maintaining a long-lived topic branch, Git has a feature called "rerere" that can help.

Rerere stands for "reuse recorded resolution" – it's a way of shortcutting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there's a conflict that looks exactly like one you've already fixed, it'll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is `rerere.enabled`, and it's handy enough to put in your global config:

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

Now, whenever you do a merge that resolves conflicts, the resolution will be recorded in the cache in case you need it in the future.

If you need to, you can interact with the rerere cache using the `git rerere` command. When it's invoked alone, Git checks its database of resolutions and tries to find a match with any current merge conflicts and resolve them (although this is done automatically if `rerere.enabled` is set to `true`). There are also subcommands to see what will be recorded, to erase specific resolution from the cache, and to clear the entire cache. We will cover rerere in more detail in **"Rerere"**.

Tagging Your Releases

When you've decided to cut a release, you'll probably want to drop a tag so you can re-create that release at any point going forward. You can create a new tag as discussed in **Chapter 2**. If you decide to sign the tag as the maintainer, the tagging may look something like this:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you do sign your tags, you may have the problem of distributing the public PGP key used to sign your tags. The maintainer of the Git project has solved this issue by including their public key as a blob in the repository and then adding a tag that points directly to that content. To do this, you can figure out which key you want by running `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]
```

Then, you can directly import the key into the Git database by exporting it and piping that through `git hash-object`, which writes a new blob with those contents into Git and gives you back the SHA-1 of the blob:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Now that you have the contents of your key in Git, you can create a tag that points directly to it by specifying the new SHA-1 value that the `hash-object` command gave you:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

If you run `git push --tags`, the `maintainer-pgp-pub` tag will be shared with everyone. If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

They can use that key to verify all your signed tags. Also, if you include instructions in the tag message, running `git show <tag>` will let you give the end user more specific instructions about tag verification.

Generating a Build Number

Because Git doesn't have monotonically increasing numbers like *v123* or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run `git describe` on that commit. Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you're describing:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, `git --version` gives you something that looks like this. If you're describing a commit that you have directly tagged, it gives you the tag name.

The `git describe` command favors annotated tags (tags created with the `-a` or `-s` flag), so release tags should be created this way if you're using `git describe`, to ensure the commit is named properly when described. You can also use this string as the target of a checkout or show command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older `git describe` output names were invalidated.

Preparing a Release

Now you want to release a build. One of the things you'll want to do is create an archive of the latest snapshot of your code for those poor souls who don't use Git. The command to do this is `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

If someone opens that tarball, they get the latest snapshot of your project under a project directory. You can also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

You now have a nice tarball and a zip archive of your project release that you can upload to your website or e-mail to people.

The Shortlog

It's time to e-mail your mailing list of people who want to know what's happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or e-mail is to use the `git shortlog` command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named 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
```

You get a clean summary of all the commits since v1.0.1, grouped by author, that you can e-mail to your list.

Підсумок

Ви маєте відчувати себе досить комфортно, пропонуючи зміни до проекту в Git, так само як і супроводжуючи власний проект чи інтегруючи зміни інших користувачів. Вітаємо серед ефективних Git розробників! В наступному розділі ви дізнаєтесь як використовувати найбільший та найпопулярніший Git хостинг — GitHub.

GitHub 6

GitHub - це найбільший хостинг для сховищ Git, та є центром співпраці між мільйонами розробників та проектів. Великий відсоток усіх сховищ Git мають хост на GitHub, та багато проектів з відкритим кодом використовують задля Git хостингу, керування завданнями, перегляду коду та для багато чого іншого. Отже хоч це і не частина вільного проекту Git, ви майже напевно захочете чи вам доведеться колсь взаємодіяти з GitHub під час професійного використання Git.

Цей розділ про ефективне використання GitHub. Ми розглянемо реєстрацію та керування обліковим записом (account), створення та використання сховищ Git, поширені схеми додавання змін до проектів та прийом змін до ваших, програмний інтерфейс GitHub, та багато маленьких порад, що зроблять ваше життя легшим.

Якщо вас не цікавить ані використання GitHub як хостингу ваших власних проектів, ані взаємодія з іншими проектами, що використовують GitHub, ви можете сміливо переходити до **Chapter 7**.

ЗМІНИ ІНТЕРФЕЙСУ

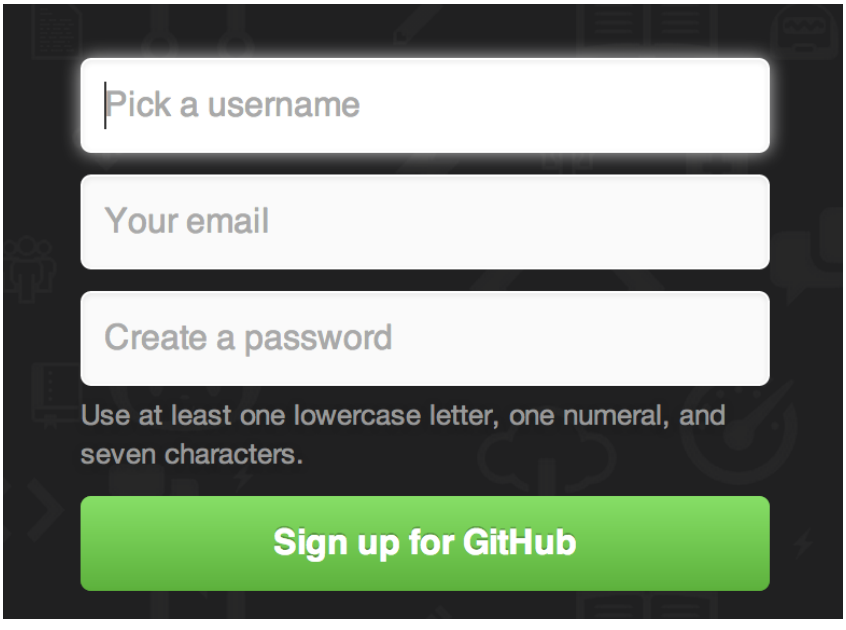
Важливо зауважити, що, як і інші активні сайти, елементи інтерфейсу на знімках екрану безперечно змінюються з часом. Сподіваємось, що головна думка, яку ми намагаємось передати за їх допомогою, досі буде зрозуміла, проте якщо ви хочете більш сучасні версії знімків, онлайн версія цієї книги може містити новіші.

Створення та налаштування облікового запису

Спершу вам треба створити безкоштовний обліковий запис. Просто зайдіть до ***https://github.com***, оберіть ім'я користувача, якого ще ні в кого немає, надайте адресу електронної пошти та пароль, натисніть на велику зелену кнопку "Sign up for Github" (Зареєструватись на Github).

ЗОБРАЖЕННЯ 6-1

Форма реєстрації GitHub.



Далі ви побачите сторінку з розцінками додаткових планів, проте зараз ми можемо це проігнорувати. GitHub відправить вам листа щоб перевірити адресу, що ви надали. Виконайте інструкції в листі - це доволі важливо (як ми потім побачимо).

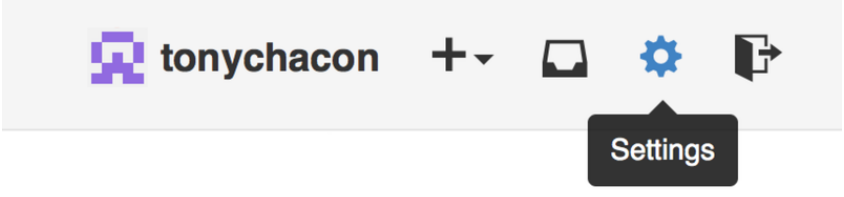
> GitHub надає вам увесь свій функціонал безкоштовно, з обмеженням, що всі ваші проекти повністю публічні (усі мають доступ на читання). Оплачувані плани пропонують фіксоване число приватних проектів, проте ми не будемо це розглядати в даній книзі.

Логотип Octocat зліва зверху проведе вас до вашої дошки керування. Ви тепер готові використовувати GitHub.

SSH доступ

Наразі, ви здатні авторизуватися щойно створеними ім'ям та паролем та зв'язуватися з Git сховищами за допомогою протоколу `https://`. Втім, щоб просто клонувати публічні проекти, вам не треба було навіть реєструватися - щойно створений обліковий запис стає в нагоді, коли ми будемо відокремлювати проекти та викладати до них зміни.

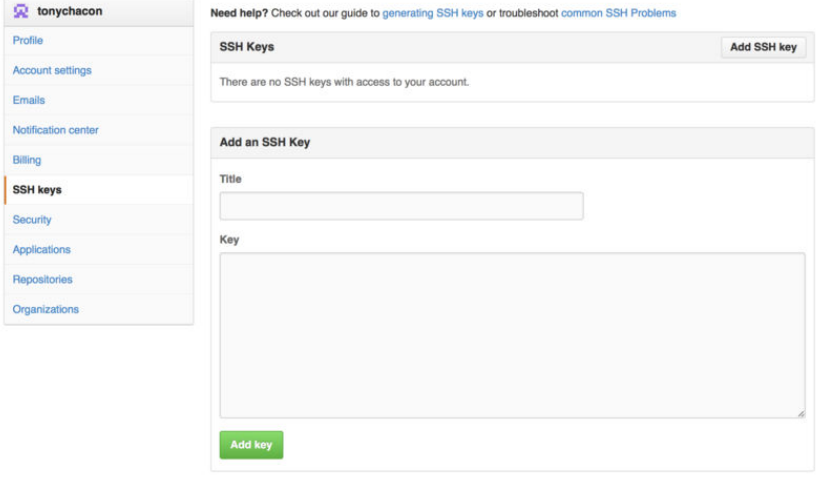
Якщо ви будете використовувати віддалені сховища через SSH, вам потрібно задати публічний ключ. (Якщо у вас його нема, дивіться “Генерація вашого публічного ключа SSH”.) Відкрийте налаштування облікового запису (account settings) за допомогою посилання в правому верхньому куту вікна:



ЗОБРАЖЕННЯ 6-2

Посилання на налаштування облікового запису (“Account settings”).

Оберіть секцію “SSH keys” (SSH ключі) з лівого боку.



ЗОБРАЖЕННЯ 6-3

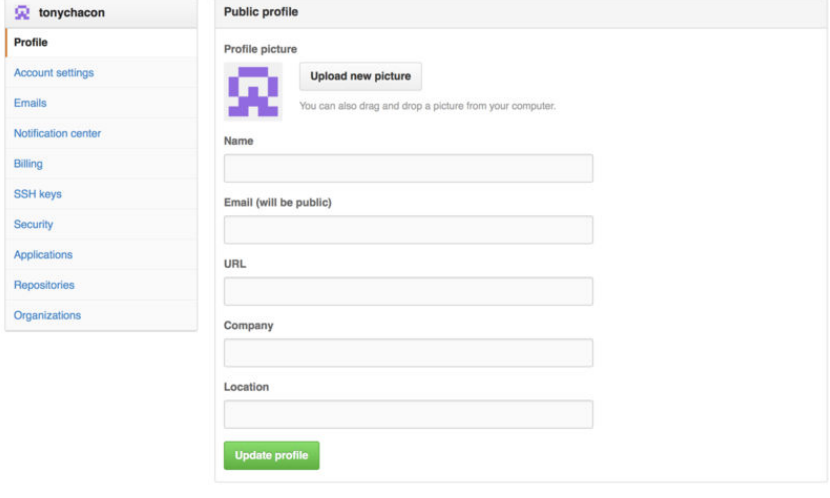
Посилання “SSH keys” (SSH ключі).

Далі, натисніть кнопку “Add an SSH key” (додати SSH ключ), назвіть ваш ключ, та вставте вміст вашого файлу з публічним ключем `~/.ssh/id_rsa.pub` (якщо ви не змінювали назву) до текстового поля, та натисніть “Add key” (додати ключ).

Обов'язково називайте свої SSH ключі розважливо. Ви можете назвати кожен з ваших ключів (наприклад "Мій Ноутбук" чи "Машина на Роботі") так, щоб якщо ви захочете скасувати якийсь потім, ви легко могли знайти потрібний вам ключ.

Ваш аватар

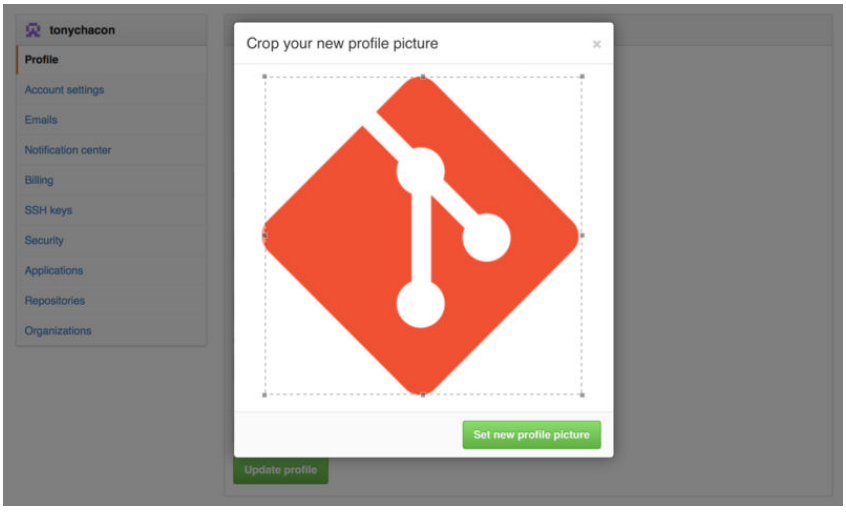
Далі, якщо ви бажаєте, можете змінити згенерований аватар обраним вами зображенням. Спершу перейдіть до вкладки "Profile" (профіль, над вкладкою SSH ключі) та натисніть "Upload new picture" (Відвантажити нове зображення).



ЗОБРАЖЕННЯ 6-4

Посилання на "Profile" (профіль).

Ми оберемо копію логотипу Git, що вже є на нашому жорсткому диску та потім зможемо обрізати зображення.



ЗОБРАЖЕННЯ 6-5

Обрізати ваш аватар

Тепер всюди, де ви будете щось робити на сайті, інші побачать ваш аватар біля вашого імені.

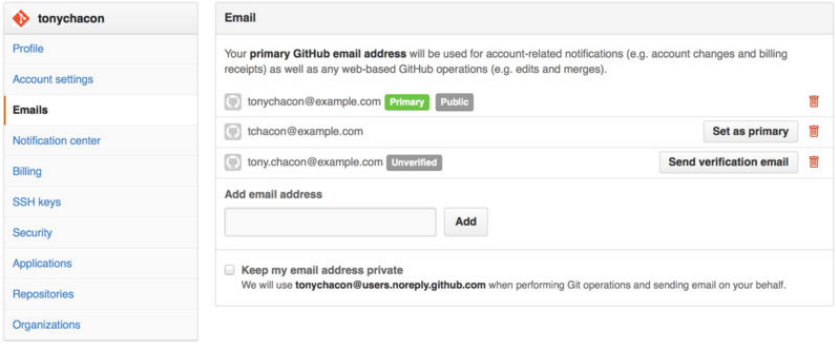
Якщо ж ви відвантажили аватар до популярного сервісу Gravatar (часто використовується для облікових записів Wordpress), то аватар буде використано автоматично, і вам цей крок не потрібен.

Ваші поштові адреси

GitHub визначає, що ваші коміти до Git є вашими через поштову адресу. Якщо ви використовуєте декілька електронних адрес для комітів, та бажаєте, щоб GitHub усіх їх пов'язував з вами, додайте всі адреси, що ви використовуєте до секції Emails в адміністративній секції.

ЗОБРАЖЕННЯ 6-6

Додавання електронних адрес

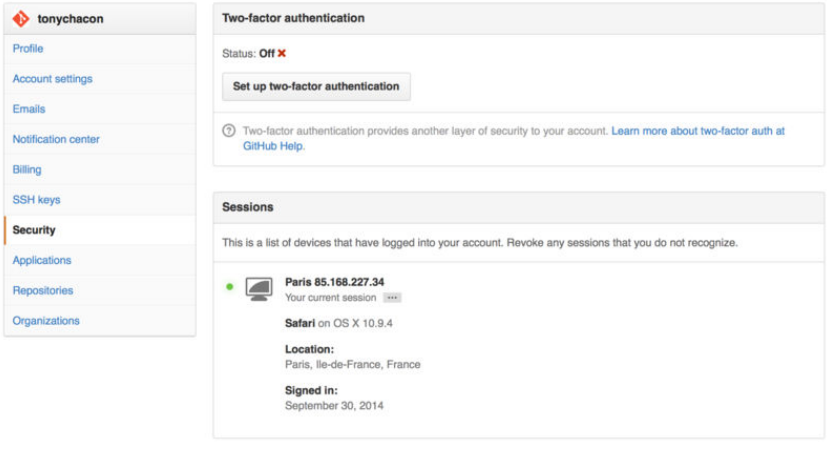


На **Figure 6-6** ми можемо бачити декілька можливих станів адреси. Верхня адреса перевірена та призначена головною, тобто до цієї адреси надсилають усі повідомлення та квитанції. Друга адреса перевірена, отже ви можете її призначити головною, якщо забажаєте їх поміняти місцями. Остання адреса не перевірена, отже ви не можете зробити її головною. Якщо GitHub побачить будь-яку з цих адрес у повідомленні коміту в будь-якому сховищі на сайті, це повідомлення буде пов'язано з вашим користувачем.

Двох-крокова авторизація

Нарешті, для більшої безпеки, вам безумовно слід налаштувати Двох-Крокову Авторизацію чи "2FA" (Two-factor Authentication). Двох-Крокова Авторизація - це механізм авторизації, що наразі стає все більш популярним задля зменшення ризиків крадіжки вашого паролю або того, що хтось буде діяти під вашим обліковим записом. Якщо ви її ввімкнете, GitHub буде потребувати два різних методи авторизації, отже якщо один з них був зламаний, зловмисник все одно не зможе отримати доступ до вашого облікового запису.

Ви можете знайти опції Двох-Крокової Авторизації у вкладці Security (безпека) налаштувань вашого облікового запису.



ЗОБРАЖЕННЯ 6-7

2FA у вкладці Безпеки

Якщо ви клацнете на кнопку “Set up two-factor authentication” (Налаштувати двох-крокову авторизацію), ви перейдете на сторінку конфігурації, де ви зможете вибрати використовувати програму для телефону для генерації другого коду (“time based one-time password” - оснований на часі одноразовий пароль), або GitHub може відправляти вам код через SMS щоразу, коли вам треба ввійти в систему.

Після вибору методу, що вам подобається, та виконання інструкції з налаштування 2FA, ваш обліковий запис буде трохи більш безпечним, та вам доведеться вводити код на додаток до паролю щоразу ви заходите на GitHub.

Як зробити внесок до проекту

Тепер, коли ваш обліковий запис налаштовано, розгляномо деякі подробиці, що можуть стати вам у нагоді при роботі над вже існуючим проектом.

Робимо форк проекту

Якщо ви бажаєте зробити внесок до існуючого проекту, проте у вас немає права на викладання до нього змін, ви можете створити “форк” (fork) проекту. Це означає, що GitHub створить копію проекту, що є повністю вашою. Вона існує в просторі імен вашого користувача, ви можете викладати до неї зміни.

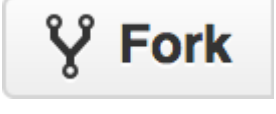
> Історично термін “форк” має дещо негативне забарвлення, зазвичай це означало, що хтось узяв проект з відкритим кодом та повів його в іншому напрямку, іноді створюючи конкуруючий проект та забираючи частину розробників. У GitHub “форк” - це просто той самий проект у вашому власному просторі імен, що дозволяє вам робити зміни до проекту публічно, це засіб зробити внесок у більш відкритій спосіб.

Таким чином, проектам не доводиться піклуватися про надання користувачам права на викладання змін. Кожен може створити форк проекту, викласти до нього зміни, та додати свої зміни до оригінального сховища за допомогою “Запита на Забирання Змін” (Pull Request), про який ми поговоримо далі. Цей запит відкриває нитку дискусії з можливістю переглядати код, власник та автор змін можуть спілкуватися про зміни доки власник не стане ними задоволений, після чого власник може злити зміни до свого сховища.

Щоб зробити форк проекту, зайдіть на сторінку проекту та натисніть на кнопку “Fork” зверху праворуч.

ЗОБРАЖЕННЯ 6-8

Кнопка “Fork”.



Через декілька секунд, ви опинитесь на сторінці вашого нового проекту, з вашою власною копією коду, яку ви можете змінювати.

Потік роботи GitHub

GitHub був спроектований навколо конкретного методу співпраці, в центрі якого Запит на Пул. Цей метод працює і в разі співпраці у маленькій команді в одному спільному сховищі, і в глобальній розподіленій компанії, чи мережі незнайомців, які допомагають проекту через десятки форків. Він базується на процесі роботи **“Тематичні гілки”**, який ми вже розглянули в **Chapter 3**.

Ось як це в цілому працює:

1. Створіть гілку на основі `master`, в ній ви будете робити всі свої зміни.
2. Зробіть якісь коміти, що поліпшують проект.
3. Викладайте цю гілку до свого проекту GitHub.

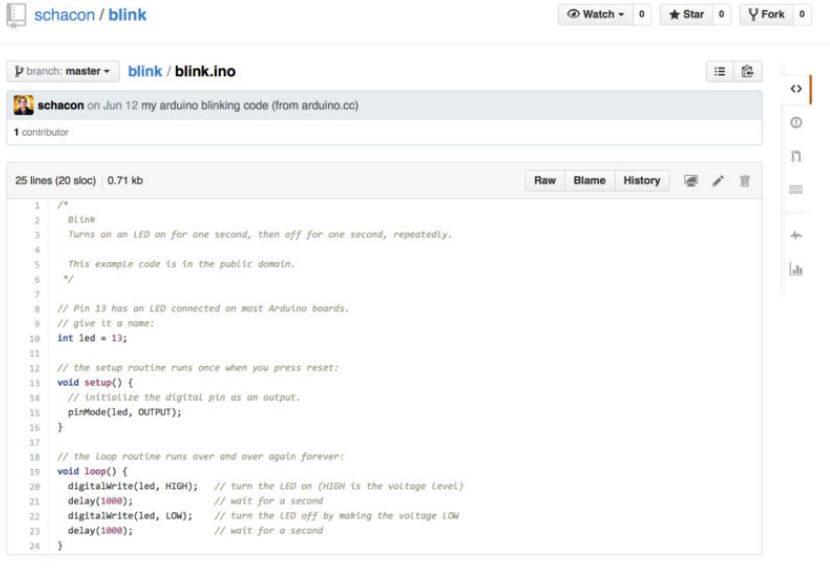
4. Відкрийте Запит на Пул за допомогою GitHub.
5. Обговоріть зміни, можливо зробіть ще декілька комітів.
6. Власник проекту заливає до проекту Запит на Пул, або закриває його.

Це дуже схоже на процес роботи Integration Manager, який ми розглянули в **“Процес роботи з менеджером інтеграції”**, проте замість використання електронної пошти для спілкування та огляду змін, команди використовують інструменти сайту GitHub.

Детально розглянемо приклад того, як запропонувати зміни до проекту з відкритим кодом, що зберігає код на GitHub, за допомогою цієї схеми.

СТВОРЕННЯ ЗАПИТУ НА ПУЛ

Тоні шукає код, який можна запустити на його мікроконтролері на платі Arduino, та знайшов чудовий програмний файл на GitHub за адресою ***https://github.com/schacon/blink***.



ЗОБРАЖЕННЯ 6-9

Проект, до якого ми хочемо зробити внесок.

Єдина проблема в тому, що частота миготіння занадто велика, ми гадаємо, що набагато краще чекати 3 секунди замість 1 між кожною

зміною стану. Отже ми покращуємо програму та надсилаємо назад до проекту як пропоновану зміну.

Спочатку, ми натискаємо на кнопку *Fork*, як ми згадували раніше, щоб отримати нашу власну копію проекту. Наше ім*я користувача тут "tonychacon", отже наша копія проекту має адресу* `*https://github.com/tonychacon/blink* і саме за цією адресою ми можемо її редагувати. Ми зробимо локальний клон, створимо гілку для наших змін, зробимо зміни в коді та нарешті викладемо ці зміни назад до GitHub.

```
$ git clone https://github.com/tonychacon/blink ❶
Cloning into '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
[slow-blink 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 Клонуємо наш форк проекту локально

❷ Створюємо окрему гілку зі зрозумілою назвою

❸ Робимо наші зміни в коді

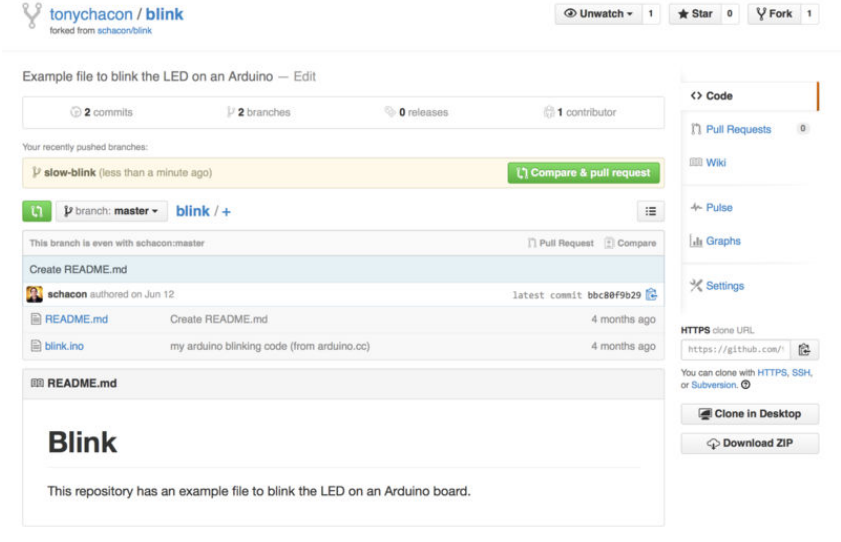
❹ Перевіряємо, що зміни вірні

❺ Робимо коміт наших змін до своєї гілки

❻ Викладаємо нашу нову гілку назад до нашого форку GitHub.

Тепер, якщо ми повернемося до нашого форку на GitHub, ми можемо бачити, що GitHub помітив, що ми виклали нову гілку та виводить нам велику зелену кнопку щоб продивитись наші зміни та відкрити Запит на Пул до оригінального проекту.

Також ви можете перейти до сторінки “Branches” (гілки) за адресою `https://github.com/<user>/<project>/branches`, щоб знайти вашу гілку та відкрити новий Запит на Пул звідти.



ЗОБРАЖЕННЯ 6-10

Кнопка Запит на Пул (Pull Request)

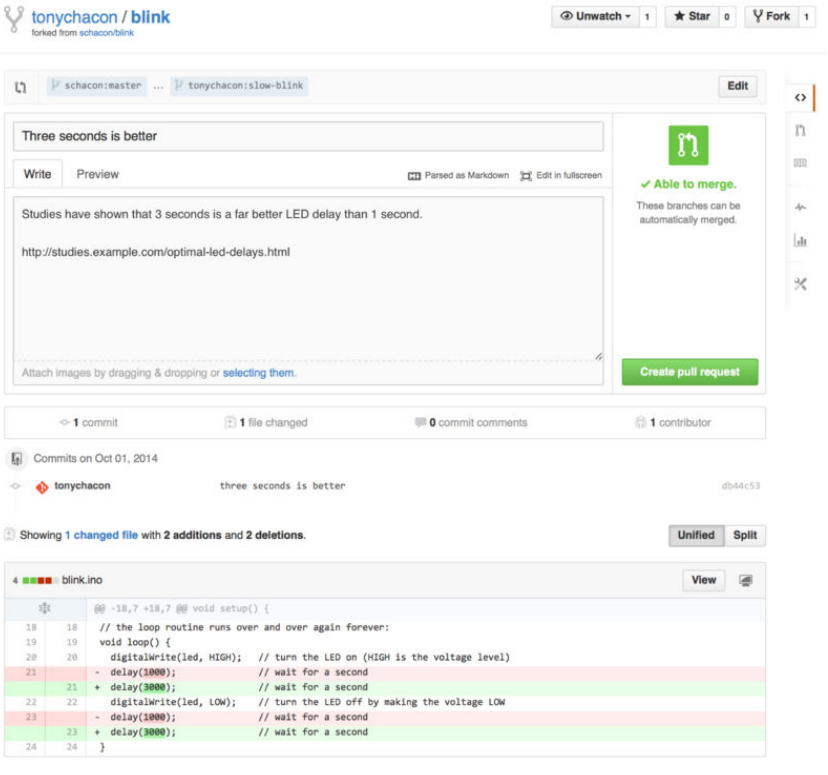
Якщо ми натиснемо на цю зелену кнопку, ми побачимо сторінку, що дозволяє нам задати назву та опис зміни, для якої ми бажаємо створити запит, щоб власник проекту міг одразу зрозуміти, що ці зміни

варто розглянути. Взагалі це слушна ідея, витратити деякі зусилля, щоб зробити опис якомога корисним, щоби автори знали, чому ці зміни їм пропонують, та чому це буде цінним внеском, який варто прийняти.

Ми також бачимо список комітів вашої гілки, які “ahead” (попереду) гілки `master` (у цьому випадку тільки один) та об’єднану різницю (unified diff) від усіх змін, що будуть зроблені, якщо цю гілку зіллє власник проекту.

ЗОБРАЖЕННЯ 6-11

Сторінка створення Запиту на Пул



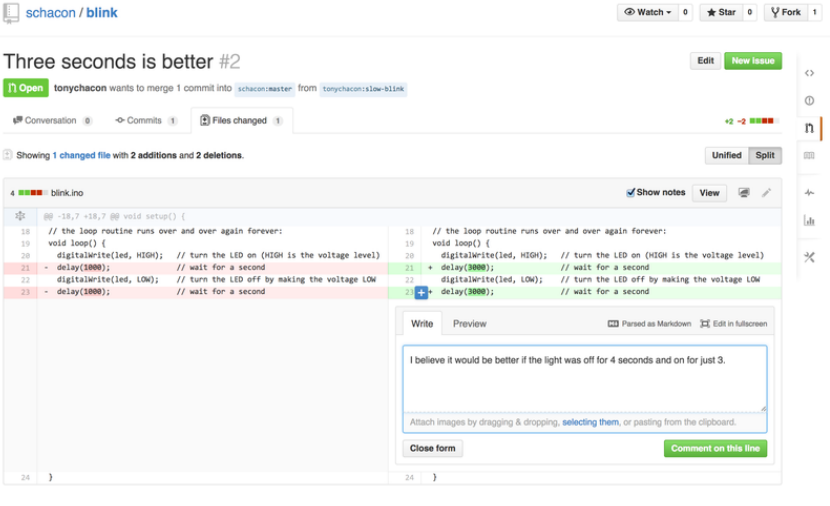
Коли ви натиснете кнопку *Create pull request* (створити запит на пул) на цій сторінці, власник проекту, від якого ви створили форк, отримає повідомлення про те, що хтось пропонує зміни з посиланням на сторінку, що містить усю інформацію про ці зміни.

Хоч Запити на Пул зазвичай використовують у відкритих проектах так, як описано вище - коли зміни вже повністю готові, Запити на Пул також часто використовуються у внутрішніх проектах *на початку* циклу розробки. Оскільки ви можете продовжувати заливати зміни до гілки навіть **після** відкриття Запиту на Пул, нерідко його відкривають завчасно та використовують як спосіб ітеративного доопрацювання в команді, а не відкривають наприкінці процесу.

ІТЕРАТИВНЕ ДООПРАЦЮВАННЯ ЗА ДОПОМОГОЮ ЗАПИТА НА ПУЛ

Тепер власник проекту може подивитися на пропоновані зміни та злити їх з проектом, відмовитись від них або прокоментувати їх. Припустимо, що ідея йому подобається, проте він бажає щоб світлодіод трохи довше був увімкнутим та вимкненим.

У представленому в **Chapter 5** процесі роботи обговорення б проходило через електронну пошту, а на GitHub усе обговорення йде прямо на сайті. Власник проекту може продивлятись об'єднану різницю (unified diff) та залишити коментар, якщо він просто натисне на будь-який з рядків.



ЗОБРАЖЕННЯ 6-12

Коментування окремого рядка коду в Запиті на Пул

Коли супроводжувач (maintainer) зробить цей коментар, людина, що відкрила Запит на Пул (та насправді будь-хто, хто слідкує за сховищем) отримає повідомлення. Ми розглянемо прилаштування

цього пізніше, проте якщо в нього були ввімкнені поштові повідомлення, Тоні отримає такого листа:

ЗОБРАЖЕННЯ 6-13

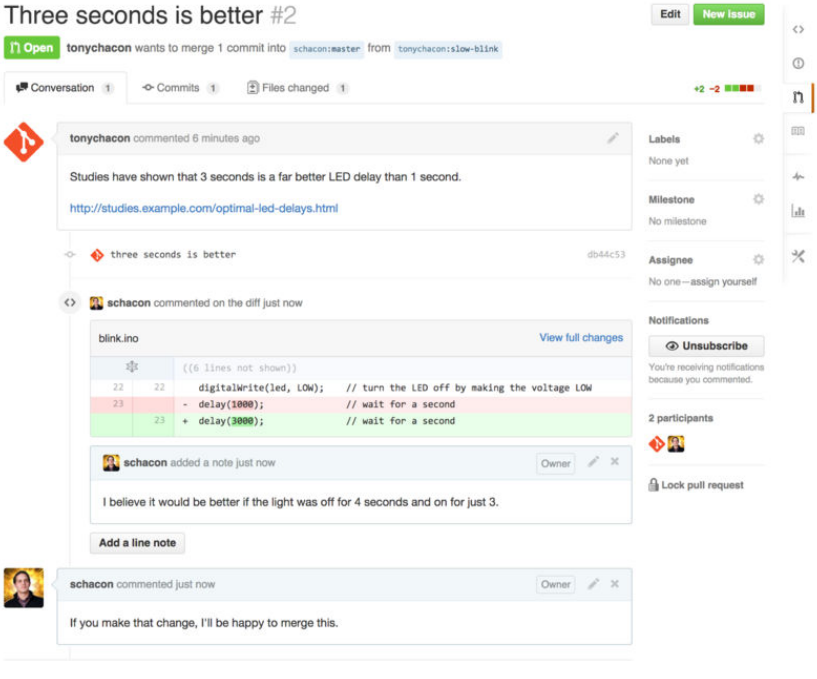
Коментарі, що їх відправили як поштові повідомлення



Хто завгодно може залишати загальні коментарі до Запиту на Пул. У **Figure 6-14** ми можемо побачити приклад того, як власник проекту прокоментував і окремий рядок коду, і залишив загальний коментар у секції обговорення. Ви можете побачити, що коментарі до коду також є частиною обговорення.

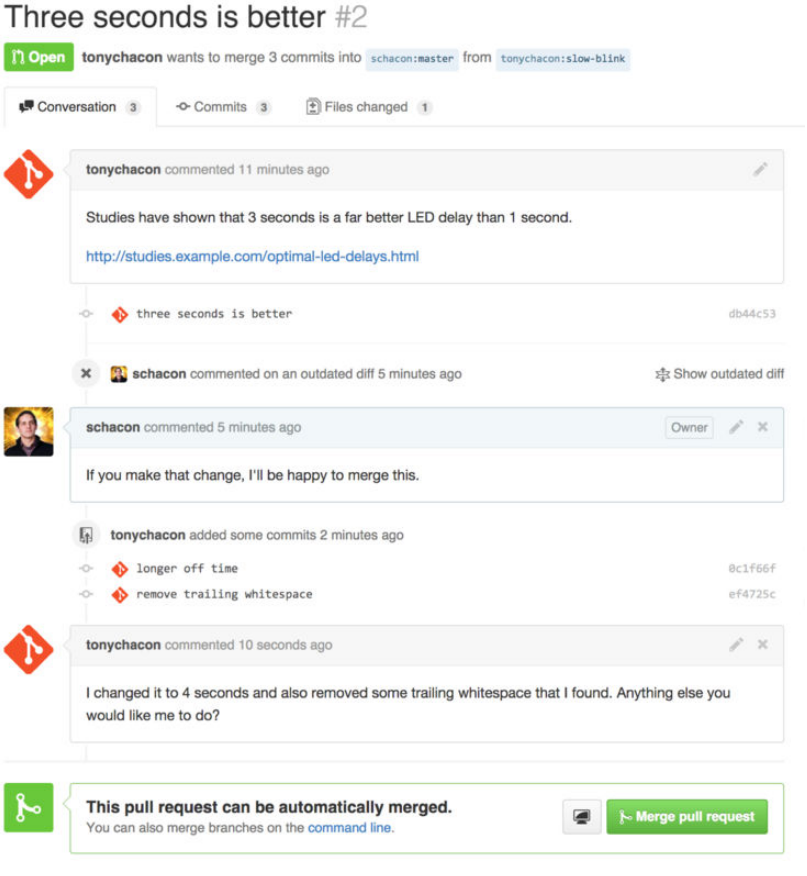
ЗОБРАЖЕННЯ 6-14

Сторінка обговорення Запиту на Пул



Тепер автори змін можуть бачити, що їм треба зробити, щоб їх зміни прийняли. На щастя це дуже просто зробити. Якби ви використовували пошту для спілкування, вам довелося б переробити ваші зміни та знову відправляти їх до поштового списку (mailing list), а за допомогою GitHub ви можете просто знову зробити коміт до своєї гілки та викласти зміни до GitHub.

Якщо автор змін це зробить, то власник проекту знову отримає повідомлення та коли він перейде до сторінки Запиту на Злиття, він побачить, що вона змінена. Насправді, оскільки рядок коду, який був мав коментар, змінився, GitHub це помітив та ховає застарілі зміни.



ЗОБРАЖЕННЯ 6-15

Фінальний Запит на Пул

Зверніть увагу на те, що якщо ви натиснете на вкладку “Files Changed” (змінені файли) цього Запиту на Пул, ви отримаєте

“об’єднану” різницю — тобто, повну агреговану різницю, яка буде додана до вашої головної гілки, якщо гілка зі змінами буде до неї злита. У термінах `git diff`, це просто автоматично виконана команда `git diff master...<гілка>`, для гілки, на якої базується цей Запит на Пул. Зверніться до **“Determining What Is Introduced”** задля докладнішим описом цього типу різниці.

Ще ви помітите, що GitHub сам перевіряє, чи Запит на Пул може бути чисто злитим (без конфліктів) та показує кнопку для злиття прямо на сервері. Ця кнопка відображається тільки якщо у вас є доступ на запис до сховища та просте злиття можливе. Якщо ви на неї натиснете, GitHub зробить злиття “не-швидко-вперед” (`non-fast-forward`), тобто навіть якщо злиття могло бути здійснено швидко-вперед, все одно буде створений коміт злиття.

Якщо ви бажаєте, ви можете просто отримати гілку та злити її локально. Якщо ви зіллєте цю гілку до гілки `master` та викладете її до GitHub, Запит на Пул автоматично стане закритим.

Це базовий процес роботи, який використовує більшість проектів GitHub. Створюєте окрему гілку, одкриваєте Запит на Злиття для неї, виникає обговорення, можливо доводиться допрацювати цю гілку та нарешті запит або закривають, або зливають до проекту.

НЕ ТІЛЬКИ ФОРКИ

Важливо зазначити, що ви можете відкрити Запит на Пул між двома гілками одного сховища. Якщо ви працюєте над функціоналом з кимось та ви обидвоє маєте доступ на запис до проекту, ви можете викласти окрему гілку до сховища та відкрити Запит на Пул з неї до гілки `master` того ж самого проекту, щоб розпочати процес перевірки та обговорення коду. Нема потреби створювати форк.

Глибше про Запити на Пул

Ми розглянули основи того, як робити внесок до проекту на GitHub, розглянемо декілька цікавих порад та хитрощів про Запити на Пул, щоб ви могли користуватись ними ефективніше.

ЗАПИТИ НА ПУЛ ЯК ПАТЧІ

Важливо розуміти, що багато проектів не розглядають Запити на Пул як чергу ідеальних патчів, які мають чисто накладатись саме в цьому порядку, як більшість основаних на поштових списках проектів розглядає послідовність патчів. Більшість проектів GitHub розглядають

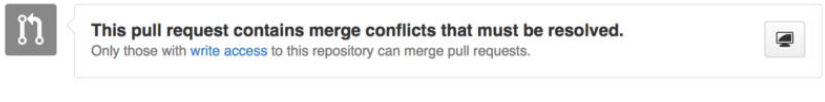
гілки Запитів на Пул як інтерактивну бесіду про запропоновані зміни, їх цікавить тільки результуючі зміни, які додаються до проекту за допомогою злиття.

Це важлива відмінність, адже зазвичай зміни пропонуються до того, як код доводять до ідеалу, що є дуже рідкісним у проектах, які використовують поштові списки з послідовностями патчів для співпраці. Це дозволяє обговоренню з супроводжувачем проекту розпочатись раніше, тому вірне рішення досягається спільними зусиллями. Коли код пропонується за допомогою Запита на Пул, та супроводжувачі або спільнота пропонує зміну, послідовність патчів зазвичай не створюється заново, а замість цього зміни просто викладаються як новий коміт до гілки, що просуває обговорення далі без втрати контексту попередньої праці.

Наприклад, якщо ви повернетесь та знову подивитесь на **Figure 6-15**, ви можете помітити, що автор змін не робив перебазування свого коміту та не відправляв ще один Запит на Пул. Натомість він додав нові коміти та виклав їх до існуючої гілки. Таким чином, якщо ви повернетесь до цього Запиту на Пул у майбутньому, ви легко знайдете весь контекст всіх прийнятих рішень. При натисканні на кнопку “Merge” (злити) на сайті, GitHub навмисно створює коміт злиття з посиланням на Запит на Пул, щоб було легко повернутись та досліджувати обговорення при необхідності.

ЙДЕМО В НОГУ З ОРИГІНАЛЬНИМ ПРОЕКТОМ

Якщо ваш Запит на Пул стає застарілим, чи не може бути чисто злитим через щось інше, ви забажаєте виправити його, щоб супроводжувач проекту міг легко злити ваші зміни. GitHub перевіряє це та повідомляє вам наприкінці кожного Запиту на Пул чи може він бути злитим без конфліктів.



ЗОБРАЖЕННЯ 6-16

Запит на Пул не може бути злитим чисто

Якщо ви бачите щось схоже на **Figure 6-16**, ви захочете виправити свою гілку щоб зображення стало зеленим та супроводжувачу не довелося виконувати зайвої праці.

У вас є два основних варіанти, як це зробити. Ви можете перебазувати вашу гілку поверх цільової гілки (зазвичай це гілка `master` сховища, від якого ви зробили форк), або ви можете злити цільову гілку до вашої гілки.

Більшість розробників на GitHub обирають другий варіант через вже зазначені раніше причини. Важлива історія та фінальне злиття, а перебазування дає нам тільки трохи чистішу історію натомість воно **набагато** складніше та в цьому варіанті легше наробити помилок.

Якщо ви бажаєте злити цільову гілку до вашої, щоб з'явилася можливість чисто злити ваш Запит на Пул, вам слід додати оригінальне сховище як нове віддалене сховище, зробити з нього фетч, злити головну гілку цього сховища до вашої гілки, виправити будь-які проблеми та нарешті залити це до гілки, на яку відкритий Запит на Пул.

Наприклад, припустимо, що в наведеному вище прикладі "tonychacon", автор оригінального проекту зробив зміни, які призводять до конфлікту в Запиті на Пул. Розглянемо що треба робити покроково.

```
$ 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 Додаємо оригінальне сховище як віддалене під ім'ям “upstream”

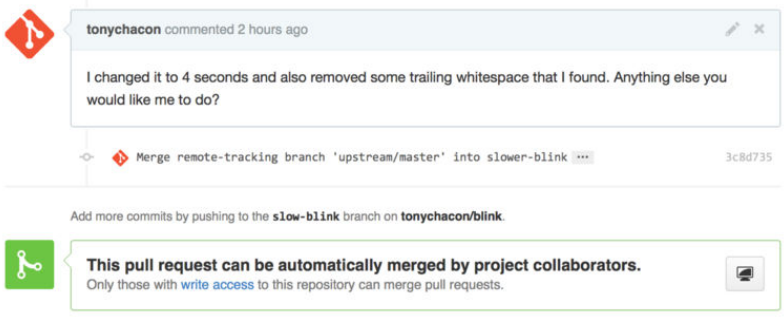
❷ Отримаємо останні зміни з цього сховища

❸ Зливаємо головну гілку до нашої

❹ Виправляємо конфлікти

❺ Заливаємо зміни назад до тієї ж нашої гілки

Щойно ви це зробите, Запит на Пул буде автоматично оновлений та знову пройде перевірка на можливість чистого злиття.



ЗОБРАЖЕННЯ 6-17

Тепер Запит на Злиття можна злити чисто

Одна з чудових рис Git полягає в тому, що ви можете це робити постійно. Якщо у вас дуже довготривалий проект, ви легко можете зливати з цільової гілки знов і знов та вам доведеться вирішувати тільки ті конфлікти, що виникли після останнього злиття, що дозволяє якось впоратися з процесом.

Якщо ви все одно бажаєте перебазувати гілку, щоб її почистити, ви безумовно можете це зробити, проте ми дуже рекомендуємо не заливати зміни насилу (force push) до гілки, для якої ви відкрили Запит на Злиття. Якщо інші люди взяли цю гілку та щось з нею робили, ви наштовхнетеся на багато труднощів, які описані в **“Небезпеки перебазовування”**. Замість цього, залийте перебазовану гілку до нової гілки на GitHub та відкривайте абсолютно новий Запит на Пул з посиланням на попередній, та закривайте перший.

ПОСИЛАННЯ

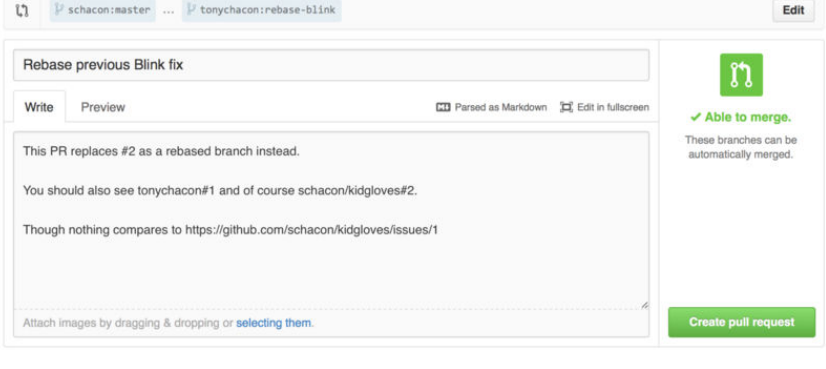
Можливо ваше наступне питання “А як мені зробити посилання на попередній Запит на Пул?”. Виявляється є дуже багато методів зробити посилання на інші речі будь-де, де ви можете писати на GitHub.

Почнемо з перехресних посилань між Запитом на Пул та Завданням (Issue). Усім Запитам на Злиття та Завданням присвоєні номери, що є унікальними в межах проекту. Наприклад, у вас може бути Запит на Пул #3 *та* Завдання #3. Якщо ви бажаєте послатись на будь-який Запит на Злиття чи Завдання з будь-якого іншого, ви можете просто написати #<номер> в будь-якому коментарі чи описі. Ви також можете бути більш детальними, якщо Завдання чи Запит на Пул знаходяться деінде. Пишіть <ім’я користувача>#<номер>, якщо ви хочете послатись на Завдання чи Запит на Пул у форку поточного сховища, або <ім’я користувача>/<назва сховища>#<номер>, щоб послатись на щось в іншому сховищі.

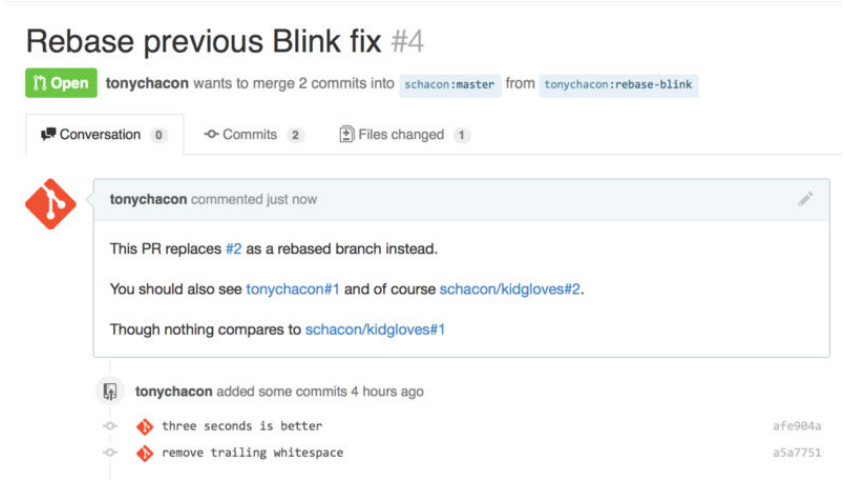
Розглянемо приклад. Припустимо, ми перебазували гілку в попередньому прикладі, створили новий запит на пул для неї, та тепер хочемо послатися на попередній запит на пул з нового. Ми також бажаємо послатись на завдання у форку сховища та на завдання з зовсім іншого проекту. Ми можемо заповнити поле опису як у **Figure 6-18**.

ЗОБРАЖЕННЯ 6-18

Перехресні посилання в Запиті на Пул



Коли ми відправимо цей запит на пул, ми побачимо, що він відображається як **Figure 6-19**.

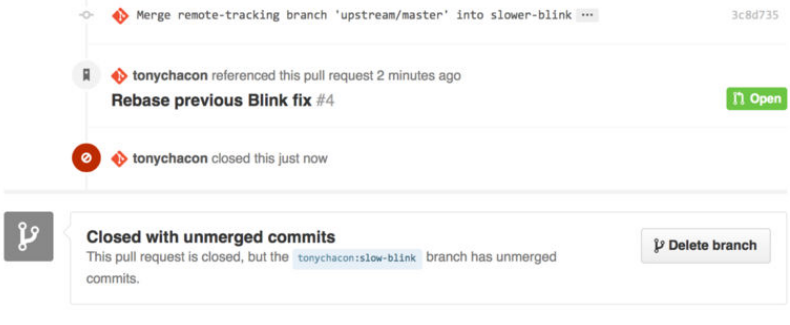


ЗОБРАЖЕННЯ 6-19

Відображення перехресних посилань у Запиті на Пул

Зверніть увагу на те, що повний GitHub URL, що ми ввели, був скорочений саме так, щоб передати всю необхідну інформацію.

Тепер якщо Тоні повернеться назад та закриє оригінальний Запит на Пул, ми зможемо це побачити згадку про це в новому, адже GitHub автоматично створить подію зворотнього стеження (`trackback event`) у хронології попереднього Запиту на Пул. Це означає, що будь-хто, хто зайде до цього Запиту на Пул та побачить, що він закритий, легко зможе перейти до сторінки запиту, що його замінив. Посилання буде виглядати приблизно як **Figure 6-20**.



ЗОБРАЖЕННЯ 6-20

Відображення перехресного посилання в Запиті на Пул.

Крім номерів завдань, ви також можете посилатись на окремий коміт за його SHA-1. Вам доведеться вказати всі 40 символів SHA-1,

проте якщо GitHub побачить це в коментарі, він зробить посилання прямо на коміт. Знову ж таки, ви можете посилатись на коміти у форках чи інших сховищах саме так, як ви це робили із завданнями.

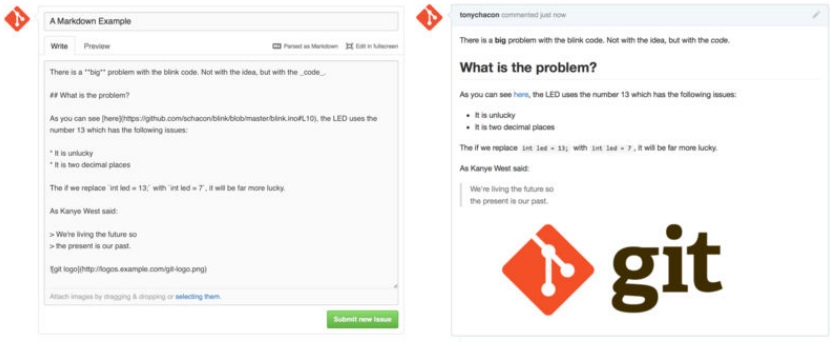
Markdown

Зв'язок з іншими завданнями це тільки мала частина цікавих речей, які ви можете робити майже в будь-якому текстовому полі на GitHub. В описах Завдань та Запитів на Пул, у коментарях, у коментарях до коду та ще багато де, ви можете використовувати так званий "GitHub різновид Markdown" (`GitHub Flavored Markdown`). Markdown це простий текст, який відображається з форматуванням.

Figure 6-21 містить приклад того, як коментарі або текст можуть бути написані та відображені за допомогою Markdown.

ЗОБРАЖЕННЯ 6-21

Приклад написання та відображення Markdown



GITHUB РІЗНОВИД MARKDOWN

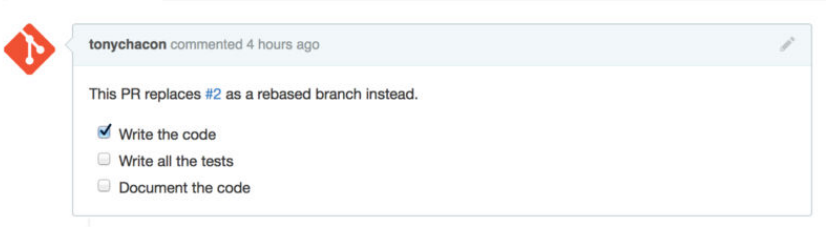
GitHub різновид Markdown додає деякі речі поза базовим синтаксисом Markdown. Вони можуть бути дуже корисними при створенні докладних описів Запитів на Пул або Завдань або коментарів до них.

Перша дійсно корисна специфічна для GitHub функція Markdown, особливо при використанні з Запитами на Пул, це Список Завдань (`Task List`). Список завдань це список прапорців (checkbox) речей, які ви хочете зробити. Зазвичай його додають до Завдання або Запиту на Пул, щоб написати список речей, які мають бути виконані до того, як можна вважати роботу завершеною.

Ви можете створити список завдань так:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

Якщо ми додамо це до опису нашого Запиту на Пул або Завдання, воно буде відображатись як **???**



ЗОБРАЖЕННЯ 6-22

Відображення списку завдань Markdown у коментарі.

Список завдань часто використовується в Запиті на Пул щоб позначити необхідні доопрацювання гілки, до того як Запит на Пул буде готовим до злиття. Дійсно чудове те, що ви можете просто натиснути на прапорець щоб оновити коментар — вам нема потреби редагувати код Markdown щоб змінювати позначку біля задачі.

Більш того, GitHub шукає список задач у ваших Завданнях та Запитах на Пул і відображає їх як метадані на сторінках, що виводять їх список. Наприклад, якщо ви маєте Запит на Пул із задачами та ви дивитесь на оглядову сторінку Запитів на Пул, ви зможете побачити як багато вже зроблено. Це допомагає людям розбити Запити на Пул на дрібніші завдання, та допомагає іншим слідкувати за прогресом гілки. Ви можете подивитись на приклад цього в **Figure 6-23**.



ЗОБРАЖЕННЯ 6-23

Підсумок списку задач у списку Запитів на Пул.

Це неймовірно корисно, коли ви відкриваєте Запит на Пул заздалегідь та використовуєте його, щоб слідкувати за своїм прогресом під час написання нової функції.

Ви також можете додавати уривки коду до коментарів. Це особливо корисно, якщо ви хочете показати щось, що ви *могли б* спробувати до того, як дійсно створювати такий коміт у своїй гілці. Їх також часто використовують щоб додати приклад коду, що не працює, чи коду, який міг би використати цей Запит на Пул.

Щоб додати уривок коду, вам треба “огородити” (fence) його зворотними апострофами.

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

Якщо додати назву мови, як ми зробили зараз (коли написали *java*), GitHub також спробує підсвітити синтаксис уривку.

ЗОБРАЖЕННЯ 6-24

Відображення прикладу відгородженого коду.



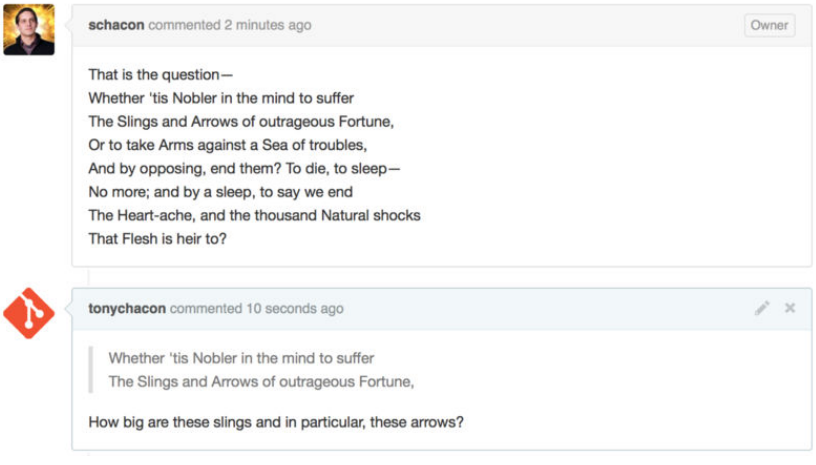
Якщо ви відповідаєте на маленьку частину довгого коментаря, ви можете вибірково процитувати частину іншого коментаря, якщо поставите на початку кожного рядку знак >. Насправді, це настільки поширена та корисна функція, що на клавіатурі для цього є окреме поєднання клавіш. Якщо ви виділите текст у коментарі, на який ви хочете відповісти та натиснете клавішу r, відкриється форма коментарю з вже процитованим для вас текстом.

Цитування виглядають приблизно так:

```
> 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?
```

При відображенні коментар буде виглядати як **Figure 6-25**.

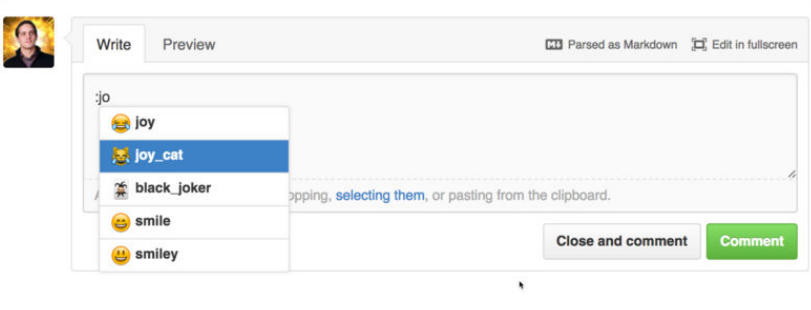


ЗОБРАЖЕННЯ 6-25

Відображення прикладу цитування.

C

Нарешті, ви також можете використовувати емодзі у ваших коментарях. Насправді їх використовують доволі широко в коментарях, їх можна побачити в багатьох Завданнях та Запитах на Пул GitHub. У GitHub навіть існує помічник емодзі.



ЗОБРАЖЕННЯ 6-26

Автодоповнювач Емодзі в дії.

Емодзі мають форму `:<назва>:` будь-де у коментарі. Наприклад, ви можете написати щось таке:

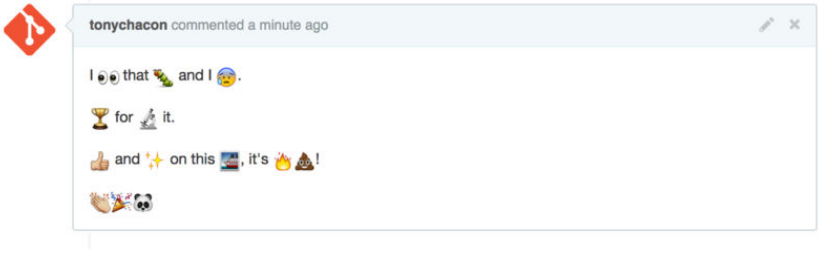
```
I :eyes: that :bug: and I :cold_sweat:.
```

```
:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

При відображенні, це виглядатиме приблизно як **Figure 6-27**.



ЗОБРАЖЕННЯ 6-27

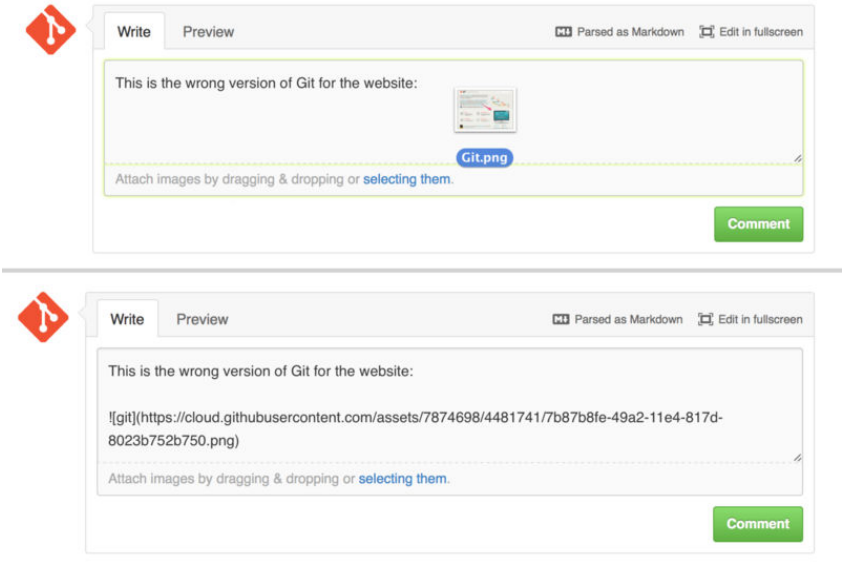
Сповнений емодзі коментар.

Не те щоб це дуже корисно, проте додає елемент розваги та емоцій до носія, в якому доволі важко передати емоції іншим чином.

> Насправді доволі багато веб сервісів використовують символи емодзі. Чудова шпаргалка для пошуку емодзі, що виражає потрібну вам емоцію знаходиться за адресою:
>
> *http://www.emoji-cheat-sheet.com*

C

Технічно це не є частиною Різновиду Markdown GitHUb, проте дуже корисно. На додаток до стандартного додавання посилань на зображення Markdown, що може бути доволі складним через необхідність шукати та вбудовувати URL, GitHub дозволяє вам просто перетягнути зображення до області вводу тексту, щоб вбудувати їх.



ЗОБРАЖЕННЯ 6-28

Перетягування зображень щоб відвантажити та автоматично вбудувати їх.

Якщо ви знову поглянете на **Figure 6-18**, то побачите маленьку підказку “Parsed as Markdown” (оброблено як Markdown) над текстовим полем. Якщо ви натиснете на неї, вам нададуть повну шпаргалку всього, що ви можете робити за допомогою Markdown на GitHub.

Супроводжування проекту

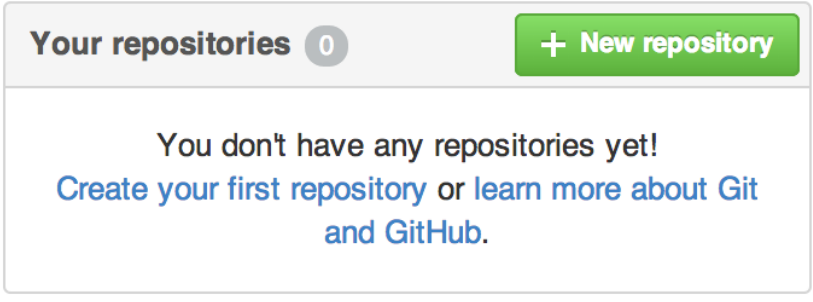
Тепер, коли ми знаємо, як робити внески до проектів, поглянемо з іншого боку: створення, супроводжування та адміністрування вашого власного проекту.

Створення нового сховища

Створимо нове сховище, до якого ми додамо код нашого проекту. Спочатку натиснемо кнопку “New repository” (нове сховище) праворуч панелі керування, чи за допомогою кнопки + у верхній панелі інструментів біля вашого імені користувача, як можна побачити в **Figure 6-30**.

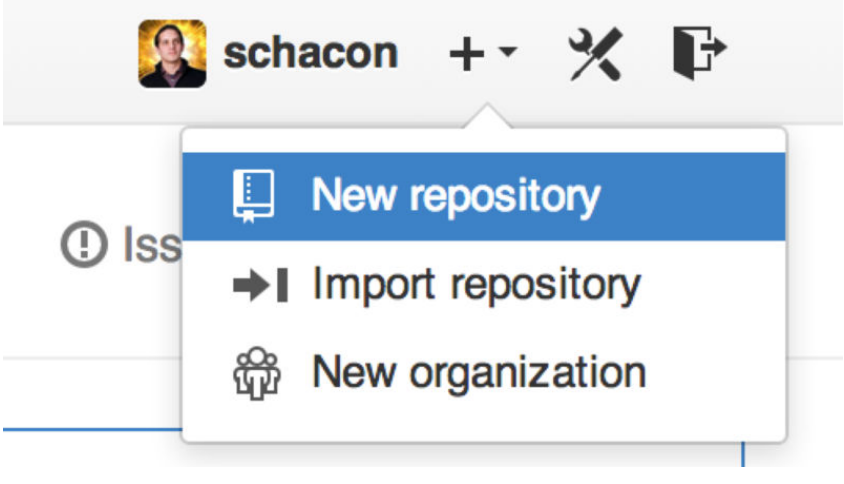
ЗОБРАЖЕННЯ 6-29

Область "Your repositories" (ваші сховища).

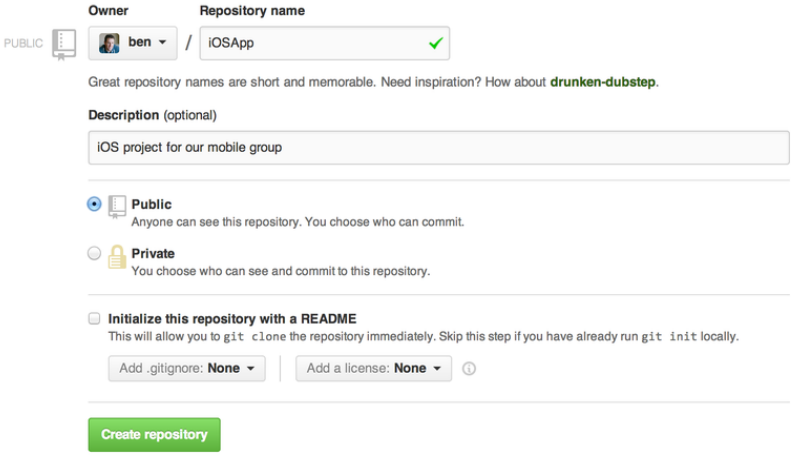


ЗОБРАЖЕННЯ 6-30

"New repository" (нове сховище) у випадному списку.



Тоді ми потрапимо до форми "нове сховище":



ЗОБРАЖЕННЯ 6-31

Форма "нове сховище".

Вам треба лише надати проекту ім'я. Усі інші поля зовсім не обов'язкові. Зараз просто натисніть на кнопку "Create Repository" (створити сховище), і бах – у вас вже є нове сховище на GitHub, під назвою `<ім'я користувача>/<назва проекту>`.

Оскільки у вашому проекті наразі нема коду, GitHub покаже вам інструкції щодо створення абсолютно нового сховища Git, або приєднання існуючого проекту Git. Ми не будемо її тут викладати. Якщо вам необхідно щось з цього пригадати, дивіться **Chapter 2**.

Тепер у вас є проект на GitHub, ви можете дати URL будь-кому, з ким хочете поділитись своїм проектом. Кожен проект на GitHub є доступним через HTTP за адресою `https://github.com/<user>/<project_name>`, та через SSH за адресою `git@github.com:<user>/<project_name>`. Git може отримувати та викладати зміни користуючись обома URL, проте вони мають контроль доступу, що базується на запиті ім'я/паролю користувача.

Часто більш бажано поширювати HTTP URL публічного проекту, адже тоді користувачу не доведеться мати обліковий запис GitHub щоб зробити клон проекту. Користувачам доведеться мати обліковий запис та відвантажений SSH ключ щоб мати доступ до вашого проекту через SSH. Посилання HTTP ще можна просто вставити до вашого веб-оглядача, щоб побачити там ваш проект.

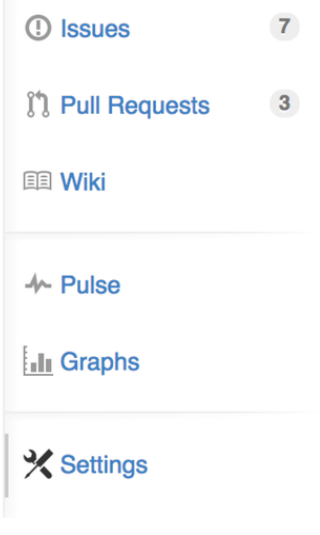
Додавання співпрацівників

Якщо ви працюєте з іншими людьми, та бажаєте надати їм право робити коміти, ви маєте додати їх до “співпрацівників” (`collaborators`). Якщо Бен, Джефф та Луїза усі мають облікові записи на GitHub, та ви бажаєте надати їм доступ на запис до вашого сховища, ви можете додати їх до свого проекту. Це надасть їм можливість робити “push”, тобто вони матимуть доступ і на читання, і на запис до проекту та сховища Git.

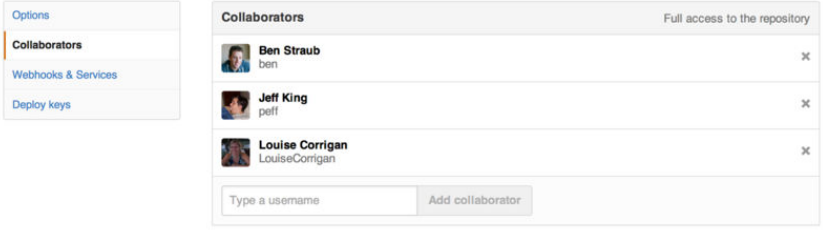
Натисніть на посилання “Settings” (налаштування) знизу бокової панелі праворуч.

ЗОБРАЖЕННЯ 6-32

Посилання на налаштування сховища.



Потім виберіть “Collaborators” (співпрацівники) з меню ліворуч. Потім просто наберіть ім’я в поле, та натисніть “Add collaborator.” (додати співпрацівника) Ви можете повторювати це скільки завгодно раз, щоб надати доступ усім, кому ви бажаєте. Якщо вам треба скасувати доступ, просто натисніть на “X” з правого боку рядка потрібного користувача.



ЗОБРАЖЕННЯ 6-33

Співпрацівники сховища.

Керування Запитами на Пул (Pull Requests)

Тепер у вас є проект з якимось кодом та можливо навіть декілька співпрацівників з доступом на запис, розгляньмо що робити, якщо хтось направив вам Запит на Пул.

Запити на пул можуть надходити або з гілки у форку вашого сховища, або просто з іншої гілки вашого сховища. Єдина різниця, що якщо він з форку, то зазвичай від людей, до гілки яких ви не маєте права викладати зміни та вони не мають права викладати зміни до вашої, а в разі внутрішнього Запиту на Злиття зазвичай обидві сторони мають на це право.

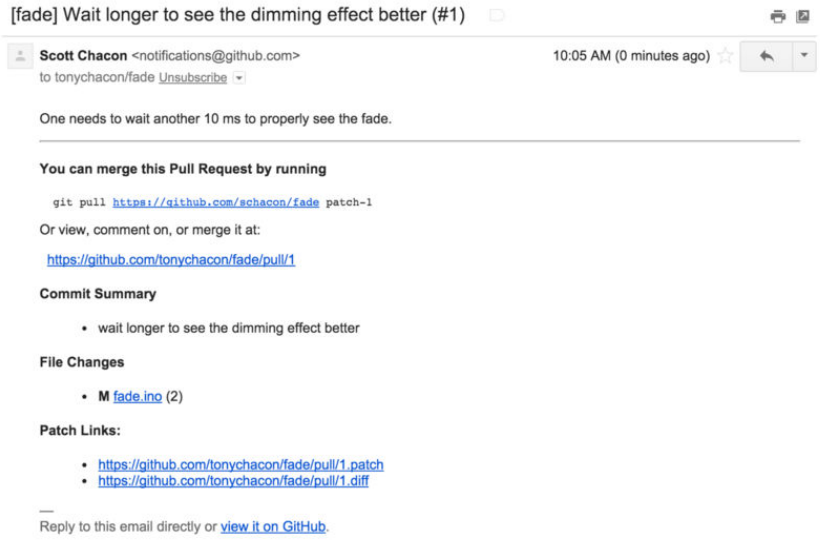
Для наступних прикладів, припустімо, що ви “tonychacon”, та ви створили новий проект Arduino під назвою “fade”.

ПОВІДОМЛЕННЯ ЕЛЕКТРОННОЮ ПОШТОЮ

Хтось приходить, змінює ваш код та відправляє вам Запит на Пул. Вам має надійти лист з повідомленням про новий Запит на Пул, що має виглядати як **Figure 6-34**.

ЗОБРАЖЕННЯ 6-34

Лист з повідомленням про новий Запит на Пул.


[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.

Варто звернути увагу на декілька речей у цьому листі. Він включає невелику статистику змін (`diffstat`) - список файлів, що були змінені в Запиті на Пул, та наскільки вони змінились. Також у ньому є посилання на сторінку GitHub Запиту на Пул. І ще декілька URL, які ви можете використовувати з командного рядка.

Якщо ви помітили рядок з текстом `git pull <url> patch-1`, то це простий метод злити зміни з віддаленої гілки без необхідності додавати віддалене сховище. Ми швидко це розглянули в **“Checking Out Remote Branches”**. Якщо ви бажаєте, то можете створити та перейти до тематичної гілки, а потім виконати цю команду, щоб злити зміни Запиту на Пул.

Інші цікаві посилання це `.diff` та `.patch`, які, як ви можете здогадатись, містять Запит на Пул у вигляді об’єднаних змін (unified diff) та патчу. Ви можете злити Запит на Пул навіть так:

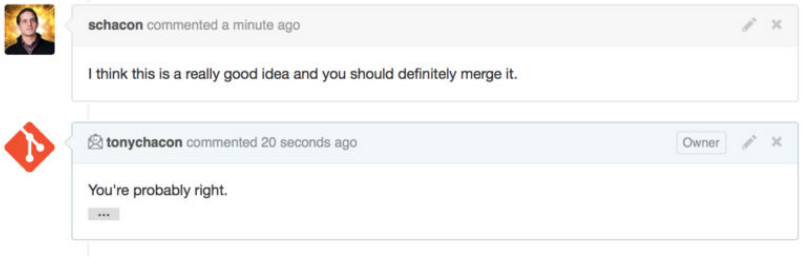
```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

СПІВПРАЦЯ НАД ЗАПИТОМ НА ПУЛ

Як ми вже бачили в **“Потік роботи GitHub”**, ви тепер можете спілкуватись з людиною, яка відкрила Запит на Пул. Ви можете коментувати окремі рядки коду, коментувати цілі коміти, чи

коментувати весь Запит на Пул, за допомогою GitHub різновиду Markdown.

Щоразу хтось інший коментує Запит на Пул, ви знову будете отримувати лист з повідомленням, отже ви будете знати що відбувається. Кожен з листів буде містити посилання на Запит на Пул саме туди, де щось відбулося, а також ви можете відповісти прямо на лист, і коментар у Запиті на Пул буде створено автоматично.



ЗОБРАЖЕННЯ 6-35

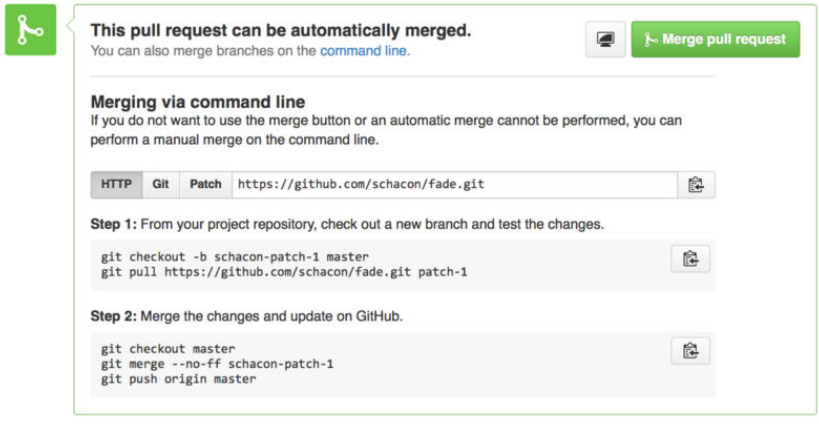
Відповідь на лист включається в переписку на GitHub.

Щойно код стає вам до вподоби та ви бажаєте його злити, ви можете або зробити пул коду та злити його локально, або використати `git pull <url> <branch>`, як ми бачили раніше, або можете додати форк як віддалене сховище, отримати з нього всі коміти, а потім вже злити зміни.

Якщо злиття тривіальне, ви також можете просто натиснути на кнопку “Merge” (злити) на сайті GitHub. Це зробить злиття “не-швидко-вперед” (`non-fast-forward`): створить коміт злиття навіть якщо злиття швидко-вперед можливе. Це означає що за будь-яких обставин, щоразу ви натискаєте на кнопку `merge`, буде створено коміт злиття. Як ви можете побачити на **Figure 6-36**, GitHub надасть вам усю цю інформацію, якщо ви натиснете на посилання вказівки (`hint`).

ЗОБРАЖЕННЯ 6-36

Кнопка злиття та інструкції щодо злиття Запиту на Пул вручну.



Якщо ви вирішите не зливати Запит, ви також можете просто закрити Запит на Злиття, про що автора запиту буде повідомлено.

ПОСИЛАННЯ (REFS) ЗАПИТІВ НА ПУЛ

Якщо вам доводиться працювати з **багатьма** Запитами на Злиття, та ви не бажаєте додавати купу віддалених сховищ чи робити по одному пулу на кожен, є один дотепний засіб, який GitHub вам дозволяє використати. Це дещо складний засіб та ми дещо докладніше розглянемо подробиці того, що відбувається в **“The Refspec”**, проте він може бути доволі корисним.

Насправді GitHub сприймає гілки Запитів на Злиття для сховища як щось на кшталт псевдо-гілок на сервері. Без додаткових дій ви не отримуєте їх при клонуванні, проте вони є в прихованому вигляді та ви можете доволі легко отримати до них доступ.

Щоб це продемонструвати, ми збираємося використати команду низького рівня (часто їх називають команди “plumbing”, про що ми прочитаємо більше в **“Сантехніка та порцеляна”**) під назвою `ls-remote`. Ця команда не потрібна при повсякденному використанні Git, проте вона корисна щоб показати нам, які посилання (`references`) присутні на сервері.

Якщо ми виконаємо цю команду на сховищі “blink”, яке ми вже раніше використовували, ми отримаємо список усіх гілок та теґів, а також інших посилань у сховищі.

```
$ 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
```

Авжеж, якщо ви у своєму сховищі виконаєте `git ls-remote origin` (замість `origin` може бути будь-яке віддалене сховище), ви отримаєте щось схоже на це.

Якщо сховище розміщене на GitHub та у вас є хоч один відкритий Запит на Пул, ви побачите посилання з префіксом `refs/pull/`. Це звичайні гілки, проте оскільки вони не мають префікса `refs/heads/`, ви не отримуєте їх при клонуванні або отриманні змін з серверу — процес отримання зазвичай повністю їх ігнорує.

Для кожного Запиту на Пул є по два посилання: те, що закінчується на `/head` вказує саме на останній коміт до гілки Запиту на Пул. Отже, якщо хтось відкриє Запит на Пул до вашого сховища, та їхня гілка називається `bug-fix` та вона вказує на коміт `a5a775`, то у **вашому** сховищі в нас не буде гілки `bug-fix` (адже це не у вашому форку), проте у нас *буде* `pull/<pr#>/head`, яке вказує на `a5a775`. Це означає, що ми легко можемо злити кожну гілку Запиту на Пул одразу, і не маємо для цього додавати купу віддалених сховищ.

Тепер ви можете отримати посилання явно наступним чином:

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

Ця команда каже Git: “Приєднайся до віддаленого сховища `origin`, завантаж звідти посилання під назвою `refs/pull/958/head`.” Git радісно це виконує, та завантажує все, що вам необхідно, щоб відновити це посилання, та записує вказівник до потрібного вам коміту в `.git/FETCH_HEAD`. Далі ви можете виконати `git merge FETCH_HEAD`, щоб злити зміни до гілки, в якій ви бажаєте перевірити зміни, проте повідомлення коміту зливання буде виглядати дещо дивно. Також, якщо ви переглядаєте **багато** запитів на пул, це стає марудним.

Існує також метод отримати *всі* запити на пул, та оновлювати їх кожен раз, коли ви з'єднуєтесь з віддаленим сховищем. Відкрийте `.git/config` у вашому улюбленому редакторі, та знайдіть там віддалене сховище `origin`. Воно має виглядати приблизно так:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Рядок, що починається з `fetch =` є "специфікацією посилань" (`refspec`). Це спосіб відображення імен у віддаленому сховищі в імена у вашій локальній директорії `.git`. Саме цей рядок з прикладу каже Git: "усе, що на віддаленому сховищі знаходиться під `refs/head` має опинитись у моєму локальному сховищі під `refs/remotes/origin`". Ви можете відредагувати цю секцію щоб додати іншу специфікацію посилань:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Цей останній рядок каже Git: "Усі посилання, що мають вигляд `refs/pull/123/head`, мають бути збережені локально як `refs/remotes/origin/pr/123`." Тепер, якщо ви збережете файл та виконаєте `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
# …
```

Тепер усі віддалені запити на пул представлені локально посиланнями так само, як і інші віддалені гілки. У них не можна вносити зміни, та вони оновлюються при отриманні змін. Це робить локальну перевірку коду із запиту на пул супер легкою:

```
$ 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'
```

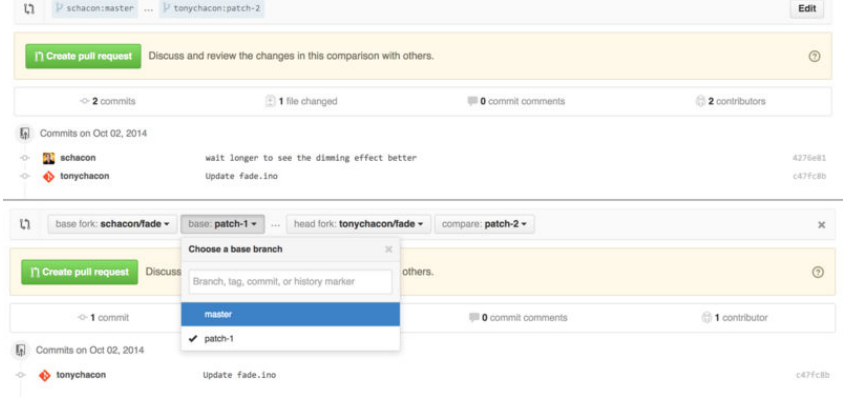
Найуважніші з вас помітили head наприкінці імені віддаленої частини специфікації посилання. У сховищі GitHub також є посилання `refs/pull/#/merge`, що вказують на коміт, який ви би отримали при натисканні на кнопку “merge” сайту. Це дозволяє вам протестувати зливання до власно натискання на кнопку.

ЗАПИТИ НА ПУЛ ДО ЗАПИТІВ НА ПУЛ

Ви можете відкривати Запити на Пул не тільки до головної (`master`) гілки, ви насправді можете відкривати Запит на Пул до будь-якої гілки в мережі. Ви дійсно можете навіть відкрити його навіть до іншого Запиту на Пул.

Якщо ви побачили Запит на Пул, що рухається у вірному напрямку, та у вас є ідеє зміни, що залежить від цього запиту, або ви невпевнені, що думка гарна, або у вас просто немає прав на запис до гілки запиту, ви можете відкрити Запит на Пул прямо до неї.

Коли ви відкриваєте Запит на Пул, нагорі сторінки є поле, що задає гілку, до якої ви створюєте запит на пул, та поле, що задає гілку, з якої ви просите взяти зміни. Якщо ви натиснете на кнопку “Edit” (редагувати) праворуч від поля, ви зможете вибрати не тільки гілки, а й форк.



ЗОБРАЖЕННЯ 6-37

Вручну змінюємо цільову гілку та форк Запиту на Пул.

Як бачите, доволі просто запросити зливання вашої нової гілки до іншого Запиту на Пул або до іншого форку проекту.

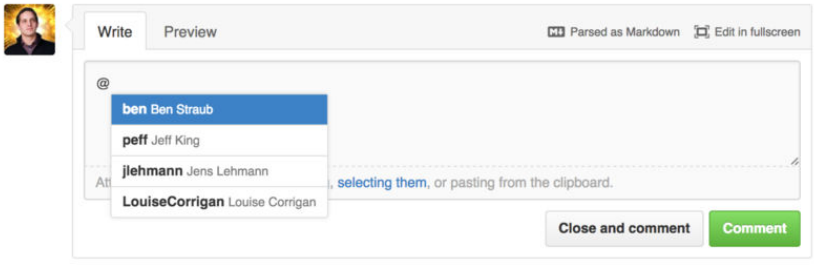
Згадки та повідомлення

У GitHub також є доволі гарна вбудована система повідомлень, яка може бути доречною, якщо у вас є питання чи вам потрібна допомога від конкретних людей чи команд.

У кожному коментарі ви можете набрати символ @ та він почне автодоповнювання імен та імен користувачів людей, що є співпрацівниками цього проекту, чи просто робили до нього внески.

ЗОБРАЖЕННЯ 6-38

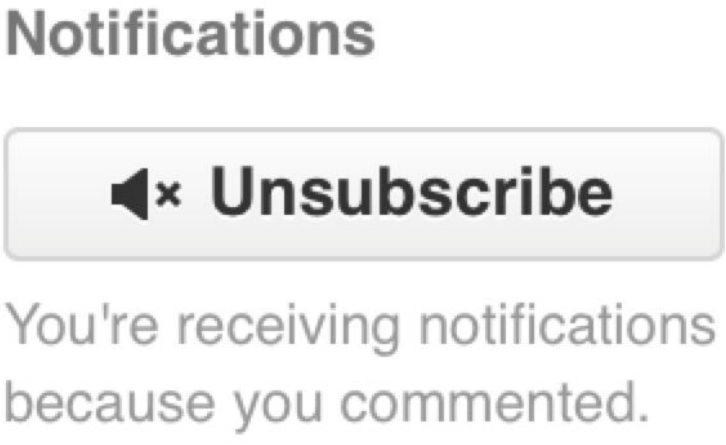
Починаємо набирати символ @ щоб когось згадати.



Ви також можете згадати користувача, якого нема в цьому випадному віконці, проте часто це виходить швидше за допомогою автодоповнювача.

Щойно ви зробите коментар зі згадкою користувача, він отримає повідомлення. Тобто це може бути дуже ефективним методом втягнути людей в спілкування, щоб їм не доводилось весь час продивлятись усі дискусії. Дуже часто люди на GitHub запрошують інших з їхньої команди чи компанії щоб переглянути Завдання чи Запит на Пул.

Якщо когось згадують у Запиті на Пул або Завданні, вони стають “підписаними” (subscribed) на них та продовжать отримувати повідомлення щоразу, коли з ними відбувається якась активність. Ви також можете бути підписані до чогось, що ви відкрили, якщо ви слідкуєте за сховищем або якщо ви коментували щось. Якщо ви більше не бажаєте отримувати повідомлення, є кнопка “Unsubscribe” (відписатись) на сторінці. Достатньо натиснути на неї, щоб GitHub припинив повідомляти вам про оновлення цієї сторінки.

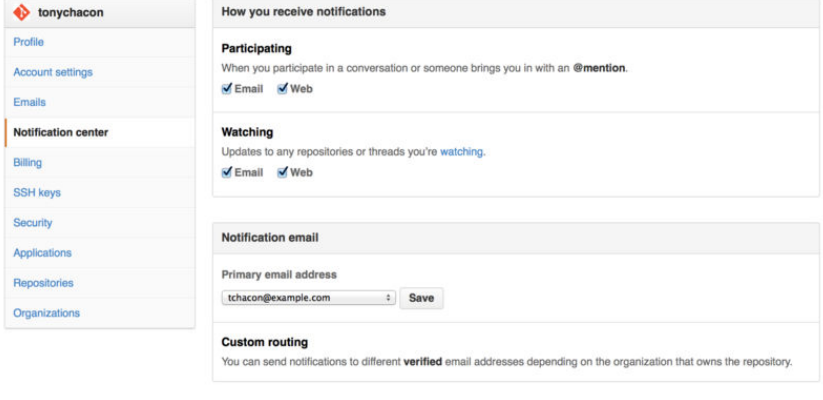


ЗОБРАЖЕННЯ 6-39

Відписуємось від Завдання чи Запиту на Пул.

СТОРІНКА ПОВІДОМЛЕНЬ

Коли ми кажемо “повідомлення” (notification) тут щодо GitHub, ми маємо на увазі окремий метод, яким GitHub намагається вам повідомити про якісь події. У вас є декілька різних методів їх налаштувати. Якщо ви перейдете до вкладки “Notification center” (центр повідомлень) зі сторінки налаштувань, ви побачите деякі доступні вам опції.



ЗОБРАЖЕННЯ 6-40

Опції центру повідомлень.

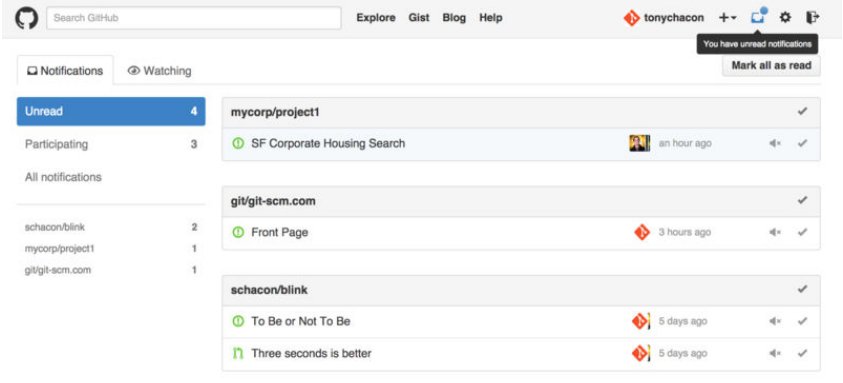
У вас є дві можливості отримувати повідомлення: через “Електронну пошту” або через “Веб” та ви можете вибрати один з них, жодного, або обидва для Запитів/Завдань та для активності в сховищах, за якими ви слідкуєте.

======= Веб повідомлення

Веб повідомлення існують тільки на GitHub і ви можете їх перевіряти виключно на GitHub. Якщо ця опція ввімкнута у ваших налаштуваннях та вам надійшло повідомлення, ви побачите маленьку синю точку над іконкою повідомлень нагорі вашого екрану, як видно на **Figure 6-41**.

ЗОБРАЖЕННЯ 6-41

Центр повідомлень



Якщо ви на неї натиснете, то побачите список усіх ваших повідомлень, згрупованих по проектам. Ви можете фільтрувати повідомлення за проектом, якщо натиснете на його назву в панелі ліворуч. Також ви можете підтвердити повідомлення, якщо натиснете на пташку (checkmark) біля повідомлення, або підтвердити *всі* повідомлення проекту, якщо натиснете на пташку зверху групи. Також є кнопка приглушення біля кожної пташки, якщо ви на неї натиснете, ви більше не будете отримувати повідомлень про цю тему.

Усі ці інструменти дуже корисні для роботи з великою кількістю повідомлень. Many GitHub power users will simply turn off email notifications entirely and manage all of their notifications through this screen. Багато досвідчених користувачів GitHub просто вимикають усі поштові повідомлення та працюють з ними виключно через цю сторінку.

Поштові повідомлення — це інший спосіб обробляти повідомлення GitHub. Якщо вони ввімкнені, ви будете отримувати листа щодо

кожного повідомлення. Ми вже бачили їх приклади в **Figure 6-13** та **Figure 6-34**. Листи також будуть вірно впорядковані у групи повідомлень, що дуже корисно, якщо ваш поштовий клієнт їх підтримує.

Також у листах від GitHub є чимало вбудованих до заголовків метаданих, що дуже допомагає при створенні особистих фільтрів та правил.

Наприклад, якщо ми подивимось на заголовки листа, що був відправлений до Тоні в **Figure 6-34**, ми побачимо наступне серед відправлених даних:

```
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
```

Тут є декілька цікавих рядків. Якщо ви бажаєте обрати або направити всі листи цього проекту, або тільки цього Запиту на Пул, для цього достатньо даних у `Message-ID`: цей заголовок має формат `<користувач>/<проект>/<тип>/<id>`. Якби б це було, наприклад, завдання, `<тип>` був би “issues” замість “pull”.

Поля `List-Post` та `List-Unsubscribe` означають, що, якщо ваш поштовий клієнт їх підтримує, ви легко можете написати (`post`) до списку, або відписатись (`unsubscribe`) від розсилки. Останнє нічим не відрізняється від використання кнопки “приглушити” (mute) у веб версії повідомлення та від кнопки “Unsubscribe” на сторінці Завдання чи Запиту на Пул.

Також варто сказати, що якщо у вас ввімкнені поштові та веб повідомлення, та ви прочитаєте поштову версію повідомлення, веб версія також буде позначена прочитаною, якщо ваш поштовий клієнт дозволяє зображення.

Особливі файли

Є декілька особливих файлів, що їх присутність у вашому сховищі помічає GitHub.

README (Прочитай мене)

Першим є файл `README`, який може бути майже будь-якого формату, який GitHub сприймає як текст. Наприклад, це може бути `README`, `README.md`, `README.asciidoc` тощо. Якщо GitHub побачить файл `README` у вашому коді, він відобразить його на головній сторінці вашого проекту.

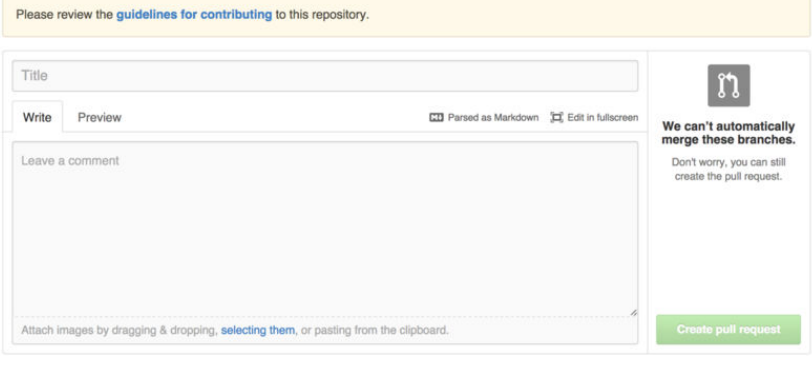
Багато команд використовують цей файл для зберігання всієї інформації, яка доречна для когось незнайомого зі сховищем або проектом. Зазвичай це такі речі як:

- Для чого цей проект
- Як його конфігурувати та інсталювати
- Приклад його використання або запуску
- Ліцензія проекту
- Як зробити внесок до нього

Оскільки GitHub буде відображати цей файл, ви можете додати до нього зображення або посилання щоб полегшати його читання.

CONTRIBUTING (Як зробити внесок)

Інший особливий файл, на який звертає увагу GitHub, називається `CONTRIBUTING`. Якщо у вас є файл `CONTRIBUTING` з будь-яким розширенням, GitHub покаже **Figure 6-42** коли хтось почне відкривати Запит на Пул.



ЗОБРАЖЕННЯ 6-42

Відкриття Запиту на Пул, якщо існує файл CONTRIBUTING.

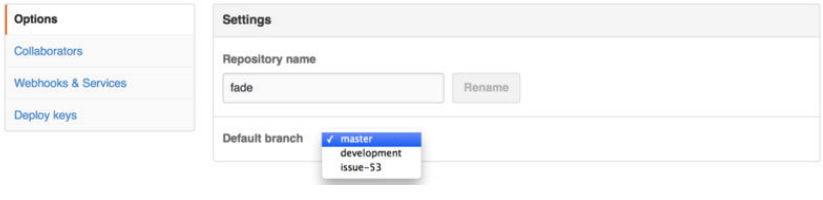
Це зроблено задля того, щоб ви могли вказати що саме ви хочете чи не хочете бачити в Запиті на Пул, який направляють до вашого проекту. Таким чином люди можуть прочитати ці інструкції до відкриття Запиту на Пул.

Адміністрування Проекту

Взагалі-то на GitHub небагато інструментів адміністрування проекту, проте деякі з них можуть бути корисними.

ЗМІНА ТИПОВОЇ ГІЛКИ

Якщо ви використовуєте не гілку “master” як головну, тобто гілку, до якої ви бажаєте щоб люди відкривали Запити на Пул, ви можете це змінити на сторінці налаштуваньсвого сховища на вкладці “Options” (опції).



ЗОБРАЖЕННЯ 6-43

Зміна типової гілки проекту.

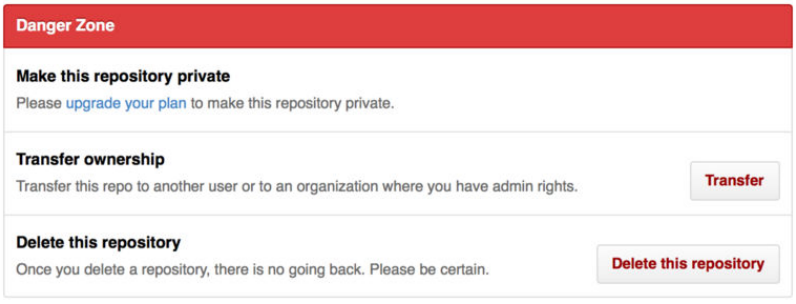
Просто змініть типову гілку в випадному віконці та без окремої вказівки всі головні операції будуть відбуватися над нею, зокрема яку гілку буде отримувати сховище при клонуванні.

ПЕРЕДАЧА ПРОЕКТУ

Якщо ви бажаєте передати проект іншому користувачу або організації на GitHub, для цього є опція “Transfer ownership” (передача власності) наприкінці тої самої вкладки “Options” на сторінці налаштувань вашого сховища.

ЗОБРАЖЕННЯ 6-44

Передача проекту іншому користувачу або організації GitHub.



Це корисно якщо ви покидаєте проект та хтось бажає його продовжити, або якщо ваш проект стає більшим і ви бажаєте перемістити його до організації.

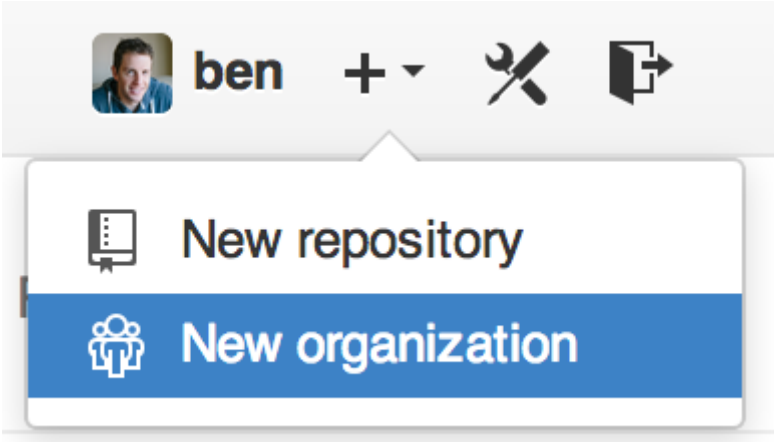
Це не тільки переміщує сховище разом з усіма глядачами (watcher) та зірками до іншого місця, а ще й налаштує перенаправлення з вашого URL до нового місця. Також будуть перенаправлені клонування та отримання змін з Git — не тільки веб запити.

Керування організацією

Крім облікових записів на одного користувача, на GitHub також є так звані Організації. Як і особисті облікові записи, облікові записи Організацій мають свій простір імен, в якому існують усі їхні проекти, проте багато іншого для них відрізняється. Ці облікові записи представляють групу людей з сумісним правом власності проектів, та є багато інструментів для керування підгрупами цих людей. Зазвичай організації використовують для груп з Відкритим Кодом (такі як “perl” чи “rails”) або компаній (такі як “google” чи “twitter”).

Основи організацій

Організацію доволі легко створити: просто натисніть на іконку “+” у нагорі праворуч на будь-якій сторінці GitHub та виберіть у меню “New organization” (нова організація).



ЗОБРАЖЕННЯ 6-45

Елемент меню “New organization” (нова організація).

Спочатку вам треба назвати вашу організацію та надати поштову адресу, що буде головним контактом групи. Потім ви можете запросити інших користувачів бути співвласниками облікового запису, якщо бажаєте.

Виконайте наступні кроки, і невдовзі ви будете власником новенької організації. Як і особисті облікові записи, організації безкоштовні, якщо все, що ви будете в них зберігати буде відкритим кодом.

Як власник організації, коли ви робите форк сховища, у вас буде вибір: робити форк у вашому власному просторі імен, чи у просторі імен організації. Коли ви створюєте нові сховища, ви можете створити їх або під особистим обліковим записом, або під будь-якою організацією, що її власником є ви. Також ви автоматично будете “слідкувати” (`watch`) за всіма сховищами, що ви створили для цих організацій.

Так само як у **“Ваш аватар”**, ви можете відвантажити аватар для вашої організації щоб трохи додати їй особливості. Також як і з особистими обліковими записами, у вас є головна сторінка організації, на якій є список усіх ваших сховищ — її можуть бачити й інші люди.

Тепер розгляньмо невеликі відмінності облікового запису організації.

Команди

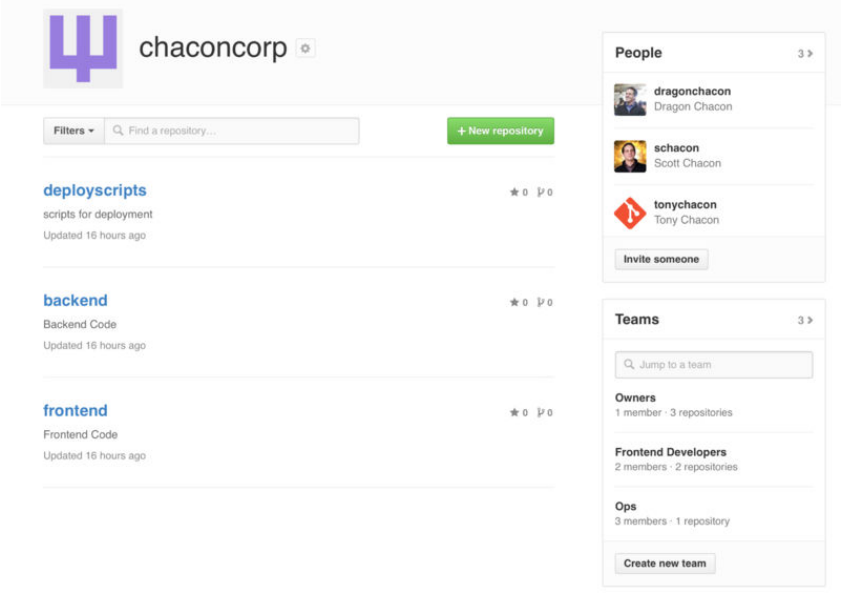
Організації пов'язані з окремими людьми через команди, що є простим групуванням окремих облікових записів і сховищ в організації, та ще який доступ ці люди мають у цих сховищах.

Наприклад, припустімо, що у вашій компанії три сховища: `frontend`, `backend` та `deployscripts`. Ви бажаєте, щоб ваші розробники HTML/CSS/Javascript мали доступ до `frontend` та можливо `backend`, а ваші люди з Операційного відділу мали доступ до `backend` та `deployscripts`. За допомогою команд цього легко досягти без необхідності керувати співпрацівниками для кожного окремого сховища.

Сторінка Організації має просту панель приладів зі всіма сховищами, користувачами та командами, що належать до цієї організації.

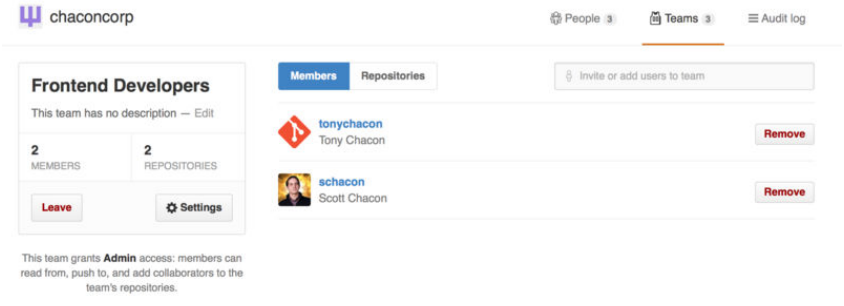
ЗОБРАЖЕННЯ 6-46

Сторінка Організації.



Щоб керувати вашими Командами, ви можете натиснути на бокову панель Teams праворуч на сторінці **Figure 6-46**. Тоді ви потрапите до сторінки, на якій ви можете додавати користувачів до команди, додавати сховища до команди та керувати налаштуваннями та рівнем доступу команди. Кожна команда може мати доступ тільки на читання,

доступ на читання та запис або доступ адміністрування до сховищ. Ви можете змінити рівень, якщо натиснете на кнопку “Settings” (налаштування) у **Figure 6-47**.



ЗОБРАЖЕННЯ 6-47

Сторінка команди.

Коли ви когось запрошуєте до команди, вони отримають листа, що повідомить їм про запрошення.

Крім того, `@згадки` команди (такі як `@acmecorp/frontend`) працюють так само, як і для окремих користувачів, крім того, що **всі** користувачі команди підписані на ці повідомлення. Це корисно, якщо ви бажаєте привернути увагу когось з команди, про те не знаєте, кого саме спитати.

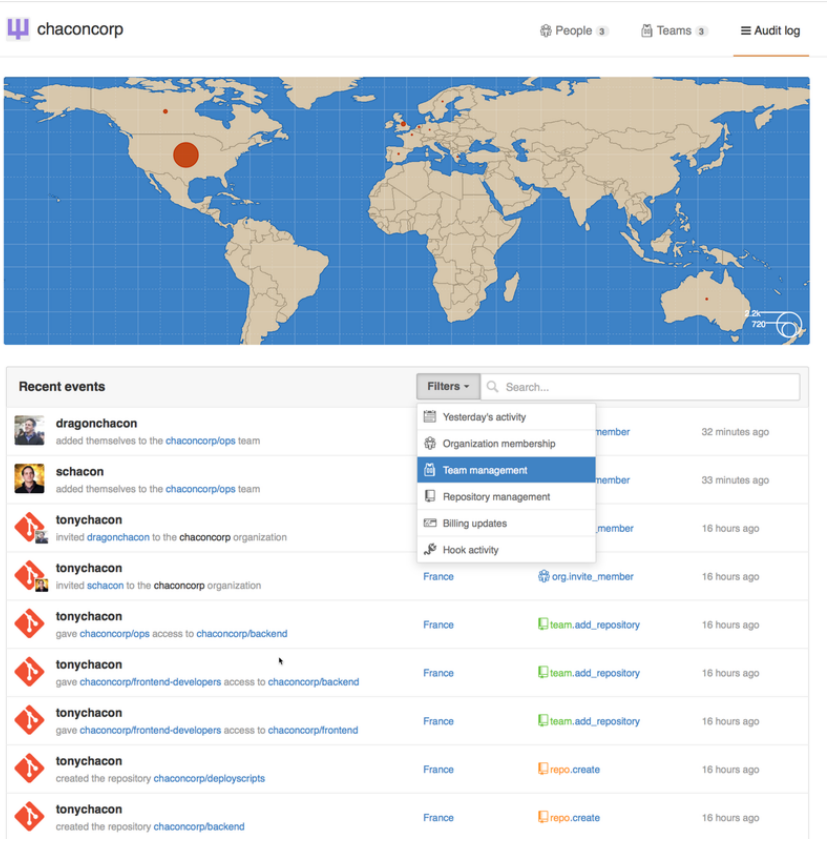
Користувач може бути в декількох командах, отже не обмежуйте себе тільки командами для контролю рівня доступу. Команди особливих інтересів, такі як `ux`, `css` чи `refactoring` корисні для деяких видів питань, та інші команди `legal` та `colorblind` для зовсім інших типів.

Журнал подій

Організації також надають власникам доступ до всієї інформації про те, що діялося в організації. Ви можете перейти до вкладки *Audit Log* (*Журнал Подій*) та побачити всі події, які відбувалися на рівні організації, хто та де в світі їх учинив.

ЗОБРАЖЕННЯ 6-48

Журнал подій.



Ви також можете фільтрувати за типами подій, за місцями або людьми.

Скриптування GitHub

Отже тепер ми розглянули весь основний функціонал та процеси роботи GitHub, проте будь-яка велика група чи проект матимуть необхідність в додаткових налаштуваннях або в інтеграції зовнішніх сервісів.

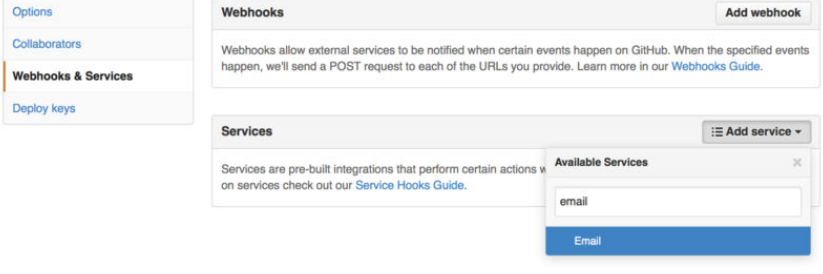
На щастя, GitHub дуже легко змінювати у багатьох напрямках. У цій секції ми розглянемо як користуватись системою хуків GitHub та його API щоб GitHub працював саме так, як ви хочете.

Хуки

Секції Хуки та Сервіси адміністративної сторінки сховища GitHub — це найпростіший метод налаштувати взаємодію між GitHub та зовнішніми системами.

СЕРВІСИ

Спочатку ми подивимось на Сервіси. І Хуки і Сервіси можна знайти в секції Settings (налаштування) вашого сховища, де ми раніше бачили як додавати Співпрацівників та змінювати типову гілку вашого проекту. У вкладці “Webhooks and Services” (веб хуки та сервіси) ви побачите щось таке **Figure 6-49**.



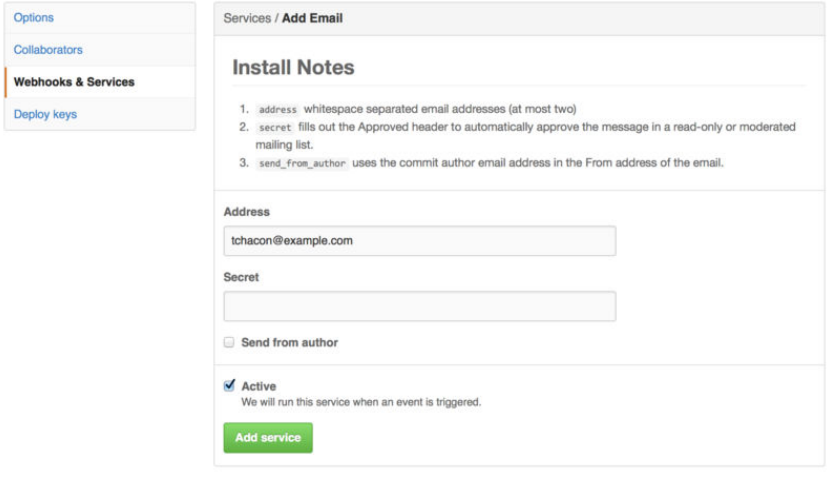
ЗОБРАЖЕННЯ 6-49

Секція конфігурації Сервіси та Хуки.

Є десятки сервісів, з яких ви можете вибирати, більшість з них є інтеграціями в інші комерційні та відкриті системи. Більшість з них є сервісами Безперервної Інтеграції, систем відстеження помилок, чатів та систем документації. Ми детально розглянемо налаштування простого прикладу: поштовий хук. Якщо ви виберете “email” з випадного віконця “Add Service” (додати сервіс), ви побачите екран налаштування **Figure 6-50**

ЗОБРАЖЕННЯ 6-50

Налаштування сервісу Email.



У цьому випадку, якщо ми натиснемо кнопку “Add service” (додати сервіс), на вказану поштову адресу буде надходити лист щоразу, коли хтось викладатиме зміни до сховища. Сервіси можуть слідкувати за різноманітними типами подій, проте більшість слідкує виключно за подіями запису до сховища, а тоді щось роблять з цією інформацією.

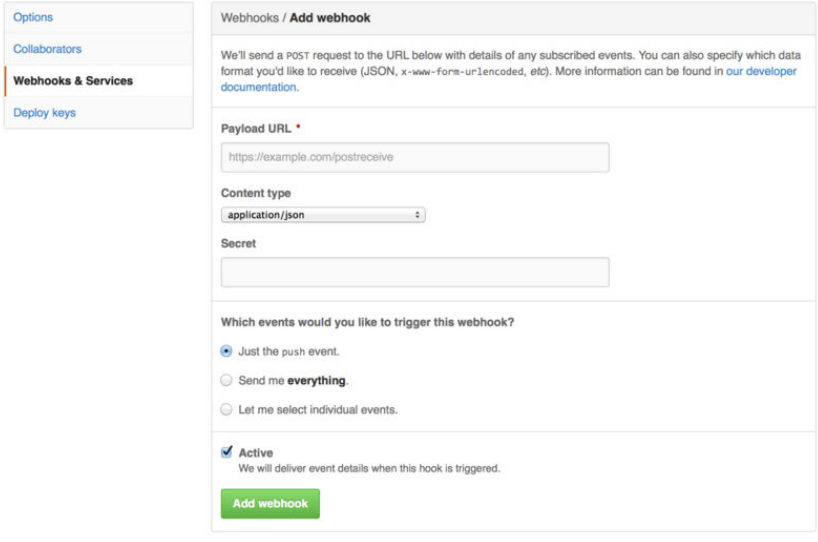
Якщо ви бажаєте інтегрувати з GitHub якусь систему, що ви зараз використовуєте, спочатку перевірте тут — можливо для неї вже існує сервіс інтеграції. Наприклад, якщо ви використовуєте Jenkins для виконання тестів вашого коду, ви можете ввімкнути вбудований сервіс інтеграції Jenkins щоб розпочинати тести щоразу, коли хтось заливає зміни до вашого сховища.

ХУКИ

Якщо ви бажаєте зробити щось більш специфічне або ви бажаєте інтегрувати сервіс чи сайт, якого нема в цьому списку, ви можете використати більш загальну систему хуків. Хуки сховища GitHub доволі прості. Ви задаєте URL та GitHub буде запит HTTP до цього URL на будь-які події, що вас цікавлять.

Зазвичай це працює так: ви запускаєте маленький веб сервіс, що слухає запити від хуку GitHub, а потім робить щось з отриманими даними.

Щоб увімкнути хук, треба натиснути на кнопку “Add webhook” (додати веб-хук) у **Figure 6-49**. Це переведе вас на сторінку, що виглядає як **Figure 6-51**.



ЗОБРАЖЕННЯ 6-51

Конфігурація веб хуку.

Налаштувати веб хук доволі просто. Найчастіше вам просто треба задати URL та секретний ключ, і натиснути “Add webhook”. Є декілька опцій щодо яких подій ви бажаєте щоб GitHub відправляв вам інформацію — без додаткових налаштувань GitHub відправляє запит тільки щодо події push, коли хтось викладає новий код до будь-якої гілки вашого сховища

Подивимося на маленький приклад веб сервісу, що ви можете налаштувати для обробки веб хуку. Ми використаємо веб фреймворк Sinatra (Ruby), адже він доволі лаконічний та вам має бути легко зрозуміти, що ми робимо.

Припустімо, що ми бажаємо отримувати листа якщо окрема людина викладає зміни до окремої гілки нашого проекту та змінює окремий файл. Цього доволі легко досягти за допомогою такого коду:

```
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"]
```

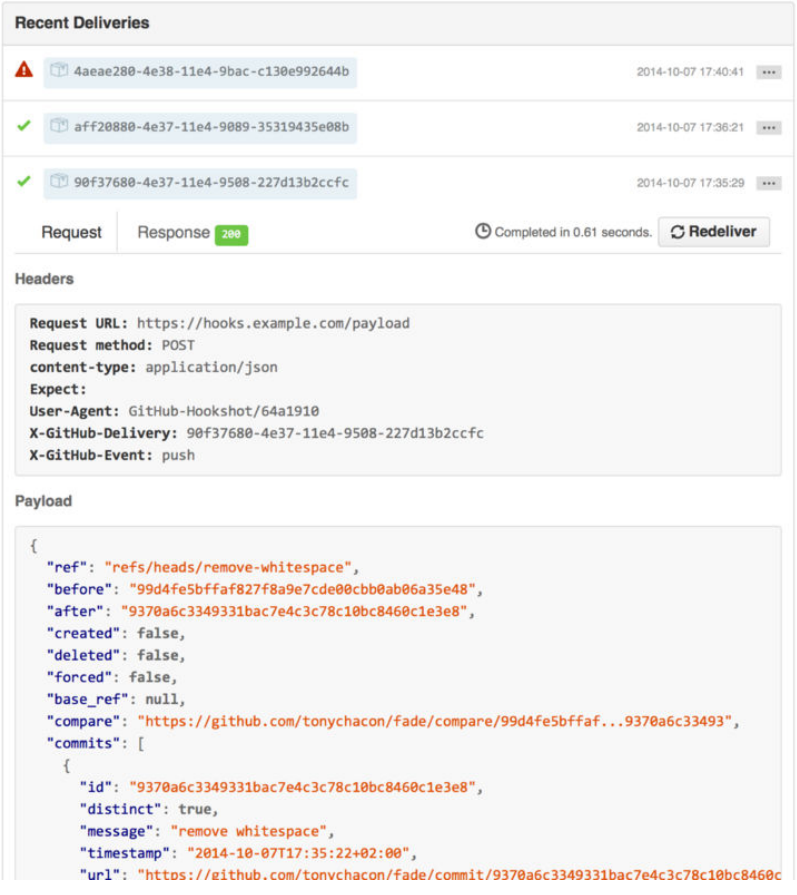
```
  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # 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
```

Тут ми отримуємо JSON з надісланого GitHub запиту, та знаходимо в ньому хто залив зміни, до якої гілки та які файли були змінені у всіх комітах. Потім ми звіряємо це з нашими умовами та надсилаємо листа, якщо все збігається.

Щоб розробляти та тестувати такі веб сервіси, у вас є гарна консоль розробника на тій сторінці, на який ви налаштовували хук. Ви можете дізнатись подробиці останніх запитів, що їх намагався зробити GitHub для цього вебхуку. Для кожного хуку ви можете дізнатись, коли він був відправлений, чи був він успішним, а також тіло та заголовки і запиту, і відповіді. Це все дуже допомагає при тестуванні та виправленні ваших хуків.



ЗОБРАЖЕННЯ 6-52

Інформація для виправлення веб хуків.

Ще одна чудова функція — це те, що ви можете ще раз надіслати будь-який з запитів щоб легко протестувати ваш сервіс.

Задля докладнішої інформацією щодо написання вебхуків та про усі різноманітні типи подій, на які ви можете реагувати, перейдіть до документації розробника GitHub за адресою: ***https://developer.github.com/webhooks/***

GitHub API

Сервіси та хуки надають вам можливість отримувати повідомлення про події, що стались з вашими сховищами, проте що як вам потрібно більше інформації про ці події? Що як вам треба автоматизувати щось

на кшталт додавання співпрацівників або додавання міток (label) до завдань (issues)?

Саме з цим нам допоможе GitHub API. GitHub має безліч кінцевих точок API, що дозволяють робити майже все, що можна робити на сайті в автоматичному режимі. У цій секції ми навчимось авторизуватися та встановлювати зв'язок з API, як коментувати завдання та як змінювати статус Запиту на Пул через API.

Базове використання

Найпростіше, що ви можете зробити — це простий GET запит до кінцевої точки, що не вимагає авторизації. Це можете бути інформація тільки для читання про користувача чи проект з відкритим кодом. Наприклад, якщо ми хочемо дізнатись більше про користувача з ім'ям "schacon", ми можемо виконати щось таке:

```
$ 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"
}
```

Є безліч кінцевих точок як ця, щоб отримувати інформацію про організації, проекти, завдання коміти — про все, що ви можете публічно бачити на GitHub. Ви навіть можете використовувати API щоб відобразити будь-який Markdown чи знайти шаблон .gitignore.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
```

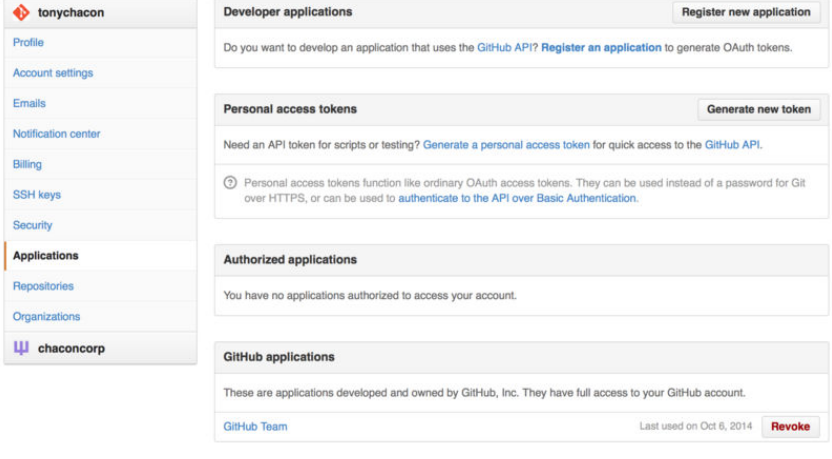
```
*.ear

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Коментування завдання

Втім, якщо ви бажаєте щось зробити на веб сайті, наприклад додати коментар до Завдання чи Запиту на Пул, або ви бажаєте побачити або взаємодіяти з приватними даними, вам доведеться авторизуватись.

Є декілька шляхів це зробити. Ви можете використати базову авторизацію, просто зі своїм ім'ям та паролем, проте зазвичай краще використовувати особисту помітку авторизації (access token). Ви можете згенерувати її з вкладки "Applications" вашої сторінки налаштувань.



ЗОБРАЖЕННЯ 6-53

Згенеруйте свою помітку авторизації з вкладки "Applications" вашої сторінки налаштувань.

Вас запитають опис та в яких контекстах ви бажаєте щоб ця помітка працювала. Переконайтесь, що ви використовуєте зрозумілий опис, щоб ви відчували впевненість при видаленні цієї помітки, коли ваш скрипт або програма більше не потрібні.

GitHub покаже вам помітку тільки одного разу, отже обов'язково скопіюйте її. Тепер ви можете використовувати її щоб авторизуватись у ваших скриптах замість використання імені та паролю. Це добре, бо

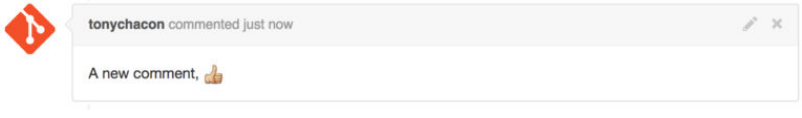
ви можете обмежити контекст використання цієї помітки, та її легко скасувати.

Також це підвищує обмеження частоти запитів. Без авторизації, ви можете робити не більше 60 запитів на годину. Якщо ви авротизуєтеся, ви можете робити до 5000 запитів на годину.

Отже використаймо її щоб прокоментувати одне з наших завдань. Припустімо, що ми бажаємо залишити коментар до окремого завдання, Завдання №6. Щоб це зробити, ми маємо надіслати HTTP POST запит до `repos/<ім'я користувача>/<сховище>/issues/<номер>/comments` з поміткою, що ми щойно згенерували в заголовку `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:"
}
```

Тепер, якщо ви відкриєте обговорення цього завдання, ви побачите коментар, що ви його щойно зробили в **Figure 6-54**.



ЗОБРАЖЕННЯ 6-54

Відправлений за допомогою GitHub API коментар.

Ви можете використовувати API майже для всього, що ви можете робити на сайті — створення та встановлення віх, призначення людей на Завдання та Запити на Пул, створення та зміна міток, отримувати

дані про коміт, створювати нові коміти та гілки, відкривання, закривання та зливання Запитів на Пул, створення та редагування команд, коментування рядків коду з Запиту на Пул, пошук на сайті тощо.

Зміна статусу Запиту на Пул

Ми розглянемо ще один останній приклад, адже це дійсно корисно, якщо ви працюєте з Запитами на Пул. Кожен коміт може мати один чи більше статусів, що з ним асоційовані, та існує API щоб додавати та отримувати ці статуси.

Більшість сервісів Безперервної Інтеграції та тестування використовують цей API щоб реагувати на нові зміни, спочатку перевіривши зміни, а потім повідомляти, чи пройшов коміт усі тести. Ви також можете використовувати цей API щоб перевірити, чи вірно є повідомлення коміту вірно оформленим, чи виконав автор змін усі ваші інструкції по тому, як робити внески, чи є коміт вірно підписаним — багато всього.

Припустімо, що ви налаштували вебхук для вашого сховища, який відправляє запит до маленького веб сервісу, що перевіряє, чи присутній в повідомленні коміту рядок `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']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

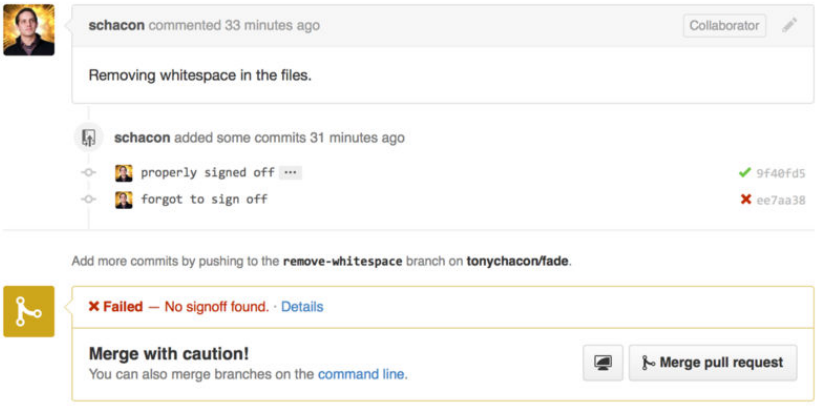
    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"
```

```
    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
```

Сподіваємось, що цей скрипт доволі легко зрозуміти. У цьому обробнику ми перевіряємо кожен викладений коміт, шукаємо рядок *Signed-off-by* у повідомленні коміту, та нарешті відправляємо HTTP POST запит до кінцевої точки `/repos/<користувач>/<сховище>/statuses/<хеш_коміту>` зі статусом.

У цьому випадку ви можете відправити статус (*success* - успіх, *failure* - невдача, *error* - помилка), опис того, що сталося, а також посилання, за яким користувач може перейти щоб отримати більше інформації та “контекст” (`context`) у разі декількох статусів для одного коміту. Наприклад, сервіс тестування може надати статус та такий як цей перевірочний сервіс також може надати статус — поле “context” дозволяє їх розрізнити.

Якщо хтось відкриє новий Запит на Пул на GitHub, та цей хук налаштований, ви можете побачити щось схоже на **Figure 6-55**.



ЗОБРАЖЕННЯ 6-55

Статус коміту через API.

Тепер ви можете бачити маленьку зелену пташку біля коміту поруч з рядком “Signed-off-by” у повідомленні. А також червоний хрестик біля того коміту, який автор забув підписати. Ви також можете бачити, що Запит на Пул отримує статус останнього коміту гілки, та попереджає вас, якщо він невдалий. Це дійсно корисно, якщо ви використовуєте це API для результатів тестів, щоб ви випадково не злили якийсь Запит на Пул, в якому останній коміт не проходить тести.

Octokit

Хоч ми майже все в прикладах робили за допомогою `curl` та простих HTTP запитів, існує декілька бібліотек з відкритим кодом, що пропонують більш зручний інтерфейс до API. У момент написання, існують бібліотеки для підтримки мов Go, Objective-C, Ruby та .NET. Перевірте ***http://github.com/octokit*** задля докладнішої інформації, адже вони обробляють більшість HTTP замість вас.

Сподіваємось, що ці інструменти можуть вам допомогти та змінити GitHub, щоб він краще працював у ваших специфічних процесах роботи. Для повної документації всього API, а також інструкцій для поширених завдань, дивіться ***https://developer.github.com***.

Підсумок

Тепер ви користувач GitHub. Ви знаєте, як створити обліковий запис, керувати організацією, створювати та викладати до сховищ, робити внески до проектів інших людей, та приймати внески від інших. В

наступному розділі ви дізнаєтесь про більш потужні утиліти та поради щодо складних ситуацій, завдяки яким ви дійсно опануєте Git.

7 Інструменти Git

Тепер ви вже вивчили більшість повсякденних команд та процесів роботи, що вам потрібні для керування та підтримки сховища Git з вашим програмним кодом. Ви виконали базові завдання супроводжування та зберігання файлів, та ви приборкали силу області індексування, легкого тематичного галуження та зливання.

Тепер ви довідаєтесь про декілька дуже потужних речей, що може робити Git, які не є необхідними в повсякденності, проте вони можуть колись знадобитись.

Вибір ревізій

Git дозволяє вам задавати окремі коміти або низку комітів декількома шляхами. Вони не обов'язково очевидні, проте їх корисно знати.

Окремі ревізії

Ви безсумнівно можете послатись на коміт за допомогою хеша SHA-1, який йому надано, проте існують більш зручні для людей методи послатись на коміт. Ця секція перелічує різноманітні способи, за допомогою яких ви можете звернутися до одного коміту.

Короткий SHA-1

Git достатньо розумний, щоб зрозуміти, який коміт ви маєте на увазі, якщо ви надасте йому тільки перші декілька символів, у разі якщо цей частковий SHA-1 має хоча б чотири символи та однозначним — тобто тільки один об'єкт у поточному сховищі починається з цього часткового SHA-1. Наприклад, щоб побачити окремий коміт, припустімо, що ви виконали команду `git log` та знайшли коміт, до якого ви бажаєте додати деякий функціонал:

```
$ 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
```

У даному випадку, візьмемо `1c002dd....`. Якщо ви хочете зробити `git show` над цим комітом, наступні команди рівнозначні (якщо коротші версії однозначні):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git може визначити унікальне скорочення для ваших значень SHA-1. Якщо додати `--abbrev-commit` до команди `git log`, вивід буде використовувати коротші значення, проте збереже їх унікальними. З цією опцією Git видає по сім символів, проте може використати й більше, якщо це необхідно для того, щоб SHA-1 був однозначним:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Зазвичай, від восьми до десяти символів є більш ніж достатньо, щоб значення було унікальним в межах проекту.

Наприклад, ядро Linux, яке є доволі великим проектом, що містить більш ніж 450000 комітів та 3.6 мільйонів об'єктів, не має двох об'єктів, чиї SHA-1 суми співпадають більш ніж першими 11 символами.

КОРОТКА ПРИМІТКА ЩОДО **SHA-1**

Багато-хто іноді переймається, що за випадковим збігом, у них з'являться два об'єкти у сховищі з однаковим значенням SHA-1. Що тоді?

Якщо ви збережете об'єкт з хешем, що вже є у іншого об'єкта у вашому сховищі, Git побачить попередній об'єкт у вашій базі даних Git та вирішить, що об'єкт вже збережено. Якщо ви колись спробуєте отримати цей об'єкт знову, ви отримаєте дані першого об'єкту.

Втім, ви маєте знати наскільки химерним є цей сценарій. SHA-1 сума має 20 байт, тобто 160 біт. Кількість випадкових об'єктів для хешування, щоб досягти 50% імовірності однієї колізії — приблизно 2^{80} (формула для визначення ймовірності колізії: `p = (n(n-1)/2) * (1/2^160))`. 2^{80} тобто 1.2×10^{24} або 1 мільйон мільярдів мільярдів. Це в 1200 разів більше, ніж піщинок на землі.

Ось приклад, щоб ви мали уявлення, що треба зробити, щоб отримати SHA-1 колізію. Якби б усі 6.5 мільярдів людей на Землі програмували, та кожну секунду кожен з них писав кількість коду, еквівалентну всій історії ядра Linux (3.6 мільйонів об'єктів Git) та заливали їх до одного величезного сховища Git, їм знадобилося би приблизно 2 роки доки в цьому сховищі опинилось достатньо об'єктів для досягнення 50% ймовірності однієї колізії SHA-1. Імовірність того, що кожен з програмістів вашої команди буде вбитий вовками в непов'язаних випадках в одну ніч, і то вища.

Гілкові посилання

Найбільш безпосередній засіб задати коміт потребує, щоб було гілкове посилання, що вказує на нього. Тоді ви можете використовувати ім*я гілки в будь-якій команді Git, що очікує об*єкта типу коміт або SHA-1 значення. Наприклад, якщо ви бажаєте показати останній об'єкт-коміт на гілці, наступні команди рівнозначні, якщо гілка `topic1` вказує на `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Якщо ви бажаєте побачити, на який саме SHA-1 вказує гілка, або якщо ви хочете побачити, з які саме SHA-1 працює будь-який з цих прикладів, ви можете використати внутрішню команду Git під назвою `rev-parse`. Ви можете переглянути **Chapter 10** для докладнішого опису

внутрішніх інструментів. Взагалі-то `rev-parse` існує для операцій низького рівня, та не створений для використання в повсякденних операціях. Втім, він може бути іноді корисним, якщо вам треба побачити, що відбувається насправді. У даному випадку ви можете використати `rev-parse` на своїй гілці.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

Скорочення reflog (журнал посилань)

Одна з речей, які Git робить у фоні доки ви собі працюєте — він зберігає “reflog” — журнал того, де був ваш HEAD та гілкових посилань за останні декілька місяців.

Ви можете продивитись свій журнал посилань за допомогою `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
```

Щоразу, коли вершина вашої гілки оновлюються за будь-якої причини, Git зберігає цю інформацію для вас у цій тимчасовій історії. Та ви можете задавати старші коміти за допомогою цих даних. Якщо ви бажаєте побачити п’яте попереднє значення HEAD вашого сховища, ви можете використати посилання `@{n}`, яке ви бачите у виводі reflog:

```
$ git show HEAD@{5}
```

Ви також можете використовувати цей синтаксис, щоб побачити де була гілка якийсь заданий час тому. Наприклад, щоб дізнатись, де була ваша гілка `master` вчора, ви можете набрати

```
$ git show master@{yesterday}
```

Це покаже вам, де вершина гілки була вчора. Ця техніка працює тільки для даних, які й досі є у вашому reflog, отже ви не можете використовувати її щоб дізнатись щось про коміти старші за декілька місяців.

Щоб побачити інформацію журналу посилань у форматі, який видає `git log`, ви можете просто виконати `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'
```

Важливо зазначити, що вся інформація журналу посилань виключно локальна — це журнал того, що ви робили у своєму сховищі. Посилання будуть іншими на іншій копії сховища. Та відразу після клонування сховища, у вас буде пустий журнал посилань, адже ніякої діяльності у вашому сховищі ще не було. Виконання `git show HEAD@{2.months.ago}` спрацює тільки якщо ви зробили клон проекту щонайменше два місяці тому — якщо ви зробили клон п'ять хвилин тому, ви не отримаєте жодних результатів.

Батьківські Посилання

Іншим розповсюдженим способом задавати коміт — за допомогою його пращурів. Якщо ви поставите `^` наприкінці посилання, Git вирішить, що воно означає батька цього коміту. Припустімо, що ви дивитесь на історію свого проекту:

```
$ 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
```

Тоді ви можете побачити попередній коміт за допомогою HEAD^, що означає “батько 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'
```

Ви також можете задати число після ^ — наприклад, d921970^2 означає “другий батько коміту d921970.” Цей синтаксис корисний тільки для комітів злиття, які мають більш ніж одного батька. Перший батько — це гілка, на якій ви були при злитті, а другий — це коміт гілки, яку ви зливали:

```
$ 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
```

Другою головною операцією для вибору пращурів є ~. Вона також вказує на першого батька, отже HEAD~ та HEAD^ рівнозначні. Різниця з’являється, коли ви додаєте число. HEAD~2 означає “перший батько першого батька,” або “дідусь” — вона переходить вздовж перших

батьків задану кількість разів. Наприклад, у наведеній вище історії, `HEAD~3` буде

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Це також може бути записано як `HEAD^^^`, що також є першим батьком першого батька першого батька:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Ви також можете поєднувати ці синтаксиси - ви можете отримати другого батька попереднього посилання (якщо це був коміт злиття) за допомогою `HEAD~3^2` тощо.

Інтервали комітів

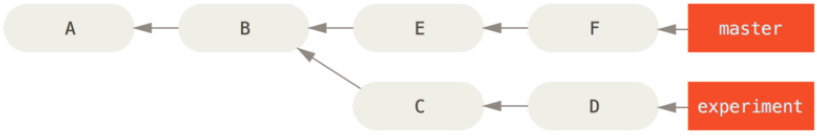
Тепер, коли ви вже знаєте, як задавати індивідуальні коміти, розгляньмо як задавати інтервали комітів. Це особливо корисно для керування гілками - якщо у вас багато гілок, ви можете використовувати специфікації інтервалів для відповіді на запитання на кшталт “Що зроблено у цій гілці, проте досі не злито до головної гілки?”

ПОДВІЙНА КРАПКА

Найпоширеніший метод задати інтервал — це подвійна крапка. Це просить Git знайти інтервал комітів, який є досяжним з одного коміта, проте не є досяжним з іншого. Наприклад, припустімо, що ваша історія комітів виглядає як **Figure 7-1**.

ЗОБРАЖЕННЯ 7-1

Приклад історії для вибору інтервалу.



Ви хочете дізнатись, що є у вашій експериментальній гілці такого, що досі не злито до вашої головної гілки `master`. Ви можете попросити Git показати журнал саме таких комітів за допомогою `master..experiment` - це означає "усі коміти, що є досяжними з experiment, проте не є досяжними з master." Задля стислості та зрозумілості в наступних прикладах, я буду використовувати літери комітів з діаграми замість дійсного виводу команди log, проте у вірному порядку:

```
$ git log master..experiment
D
C
```

Якщо ви, навпаки, бажаєте дізнатись протилежне - усі коміти в `master`, яких немає в `experiment` - треба просто поміняти місцями імена гілок. `experiment..master` показує все в `master`, що не є досяжним з `experiment`:

```
$ git log experiment..master
F
E
```

Це корисно, якщо ви бажаєте синхронізувати гілку `experiment` та переглянути, що ви збираєтесь зливати. Ще доволі часто цей синтаксис використовується щоб побачити, що ви збираєтесь викладати до віддаленого сховища:

```
$ git log origin/master..HEAD
```

Ця команда показує всі коміти у вашій поточній гілці, яких немає в гілці `master` віддаленого сховища `origin`. Якщо виконати `git push`, а ваша поточна гілка слідкує за `origin/master`, саме коміти, яки показує `git log origin/master..HEAD`, і будуть передані серверу. Ви також

можете пропустити один бік синтаксису, тоді Git використає HEAD. Наприклад, ви можете отримати такий саме результат, як і в попередньому прикладі, якщо наберете `git log origin/master..` - Git поставить HEAD, якщо один бік відсутній.

ДЕКІЛЬКА ТОЧОК

Подвійна крапка — це корисне скорочення. Проте можливо ви бажаєте задати більш ніж дві гілки для позначення ревізій, наприклад щоб побачити які коміти є в декількох гілках, проте їх немає в поточній гілці. Git дозволяє це за допомогою символу `^` або `--not` до будь-якого посилання, з якого ви не хочете бачити досяжні коміти. Отже наступні три команди еквівалентні:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Це мило, адже за допомогою цього синтаксису ви можете задати більш ніж два посилання у вашому запиті, чого ви не можете досягти за допомогою подвійної крапки. Наприклад, якщо ви бажаєте побачити всі коміти, які є досяжними з `refA` або `refB`, проте не з `refC`, ви можете набрати одне з:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Це створює доволі потужну систему запитів ревізій, яка має вам допомогти зрозуміти, що є у ваших гілках.

ПОТРІЙНА КРАПКА

Останній головний синтаксис вибору інтервалів — це потрійна крапка, яка задає всі коміти, які досяжні з будь-якої з двох посилань, проте недосяжні з них обох. Знову подивіться на приклад історії комітів у **Figure 7-1**. Якщо ви бажаєте побачити, що є в `master` або `experiment`, проте не є для них спільним, ви можете виконати

```
$ git log master...experiment
F
E
```

```
D
C
```

Нагадуємо, це видає вам нормальний вивід команди `log`, проте показує вам інформацію тільки про ці чотири коміти, традиційно упорядковані за датою коміту.

Часто з такою командою `log` використовують опцію `--left-right`, яка показує з якого боку інтервалу кожен коміт. Це робить дані більш корисними:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

За допомогою цих інструментів, ви легко можете дати Git знати, який коміт або коміти ви бажаєте оглянути.

Інтерактивне індексування

Git пропонує декілька скриптів, які полегшують деякі завдання командного рядка. Тут ми розглянемо декілька інтерактивних команд, що допомагають легко майструвати коміти, щоб вони включали тільки певні комбінації та частини файлів. Ці інструменти дуже допомагають, коли ви редагуєте купу файлів, а потім вирішуєте, що краще записати ці зміни в декількох цілеспрямованих комітах, ніж в одному великому безладному коміті. Таким чином ви можете бути впевнені, що ваші коміти є логічно розділеними змінами та можуть бути легко перевірені програмістами, що з вами працюють. Якщо ви виконаєте `git add` з опцією `-i` або `--interactive`, Git перейде до інтерактивного режиму та покаже щось таке:

```
$ 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>
```

Як ви бачите, ця команда показує вам зовсім інший вигляд області індексації – майже ту саму інформацію, що ви отримуєте за допомогою `git status`, проте трохи стисліше та докладніше. У списку проіндексовані зміни вказані ліворуч, а непроіндексовані - праворуч.

Після цього йде секція Команд. За їх допомогою ви можете багато всього: індексувати файли, деіндексувати файли, індексувати частини файлів, додавати не супроводжувані файли та продивлятись проіндексовані зміни.

Індексування та деіндексування файлів

Якщо ви наберете `2` або `u` після запиту `What now>`, скрипт запитає вас, які файли ви бажаєте проіндексувати:

```
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>>
```

Щоб проіндексувати файли `TODO` та `index.html`, ви можете набрати числа:

```
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>>
```

`*` біля кожного файла означає, що файл є вибраним для індексування. Якщо ви нічого не наберете, та натиснете Ентер після порожнього запиту `Update>>`, Git візьме все вибране та проіндексує його:

```
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
```

Як бачите, файли TODO та index.html проіндексовані, а файл simplegit.rb досі не проіндексований. Якщо ви тепер бажаєте деіндексувати файл TODO, ви можете використати опцію 3 або r (від revert, скасувати):

```
*** 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
```

Якщо ви знову подивитесь на статус Git, то побачите, що файл TODO деіндексований:

```
*** 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
```

Щоб побачити зміни, які ви проіндексували, ви можете використати команди 6 або d (від diff, різниця). Вона показує список індексованих файлів, та ви можете вибрати ті, для яких бажаєте побачити індексовані зміни. Це дуже схоже на виконання `git diff --cached` з командного рядка:

```
*** 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">
```

За допомогою цих команд, ви можете використовувати інтерактивний режим додавання щоб поратися в області індексування з легкістю.

Індексування латок (patch)

Git також може індексувати окремі частини файлів без індексування решти частин. Наприклад, якщо ви зробили дві зміни у файлі simplegit.rg, проте бажаєте проіндексувати тільки одну з них, це дуже легко зробити в Git. Наберіть 5 або p (від patch - латка) в інтерактивному запиті. Git спитає, які файли ви бажаєте частково індексувати. Потім, для кожної секції вибраних файлів, відобразить шмати різниці файлів та запитає, чи бажаєте ви проіндексувати кожен:

```
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,?]?
```

Наразі у вас багато варіантів. Набравши `?` ви побачите список своїх можливостей:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ? (Проіндексувати цей шмат [y,n,a,d,/,j,J,g
y - stage this hunk (проіндексувати цей шмат)
n - do not stage this hunk (не індексувати цей шмат)
a - stage this and all the remaining hunks in the file (індексувати цей і решту шм
d - do not stage this hunk nor any of the remaining hunks in the file (не індексув
g - select a hunk to go to (вибрати шмат до якого перейти)
/ - search for a hunk matching the given regex (шукати шмат за регулярним виразом)
j - leave this hunk undecided, see next undecided hunk (залишити цей шмат нерозв'я
J - leave this hunk undecided, see next hunk (залишити цей шмат нерозв'язаним, пер
k - leave this hunk undecided, see previous undecided hunk (залишити цей шмат нер
K - leave this hunk undecided, see previous hunk (залишити цей шмат нерозв'язаним,
s - split the current hunk into smaller hunks (подрібнити цей шмат)
e - manually edit the current hunk (редагувати поточний шмат вручну)
? - print help (надрукувати допомогу)
```

Зазвичай вам буде достатньо `y` та `n`, якщо ви бажаєте індексувати кожен, проте індексування всіх шматів у деяких файлах, або відтермінування рішення щодо шмату також можуть бути корисними. Якщо ви проіндексуєте одну частину файлу та залишите іншу неіндексованою, ваш статус буде виглядати так:

```
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
```

Статус файлу `simplegit.rb` цікавий. Він показує, що декілька рядків індексовані, а декілька ні. Ви частково проіндексували файл. Тепер ви

можете покинути скрипт інтерактивного додавання та виконати `git commit` щоб зберегти частково проіндексовані файли.

Вам не обов'язково бути в інтерактивному режимі додавання щоб частково проіндексувати файл - ви можете запустити той самий скрипт за допомогою `git add -p` або `git add --patch` з командного рядка.

Більш того, ви можете використовувати цей режим для часткового скидання (resetting) файлів за допомогою команди `reset --patch`, отримувати частини файлів командою `checkout --patch`) та заховувати частини файлів командою `stash save --patch`. Ми докладніше розглянемо кожну з них, коли дійдемо до складніших використань цих команд.

Хования та чищення

Часто, коли ви працюєте над частиною свого проекту, усе перебуває в неохайному стані, а ви бажаєте переключити гілки щоб трохи попрацювати над чимось іншим. Складність у тому, що ви не бажаєте робити коміт напівготового завдання тільки щоб повернутися до цього стану пізніше. З цим нам допомагає команда `git stash`.

Хования (stashing) бере чорновий стан вашої робочої директорії – тобто ваші змінені супроводжувані файли та індексовані зміни – та зберігає їх у стеку незавершених змін, які ви можете знову використати будь-коли.

Хования ваших змін

Задля демонстрації, треба перейти до проекту та почати працювати над декількома файлами та можливо проіндексувати один з них. Якщо виконати `git status`, можна побачити чорновий стан:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   lib/simplegit.rb
```

Тепер ви бажаєте переключити гілки, проте наразі не бажаєте робити коміт ваших змін. Отже ви сховаєте ваші зміни. Щоб додати нове ховання (stash) до вашого стеку, виконайте `git stash` або `git stash save`:

```
$ 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")
```

Ваша робоча директорія стала чистою:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Тепер ви легко можете переключати гілки та працювати деінде. Ваші зміни збережені в стеку. Щоб побачити збереженні ховання, використовуйте `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
```

У даному випадку, два ховання були зроблені раніше, отже у вас є доступ до трьох різних схованих наборів змін. Ви можете використати щойно сховані зміни за допомогою команди, яку показано у виводу допомоги вищенаведеної команди ховання: `git stash apply`. Якщо ви бажаєте використати одне з попередніх ховань, то вам доведеться задати його назву, наприклад: `git stash apply stash@{2}`. Якщо ви не задаєте ховання, Git використовує найновіше та намагається його використати:

```
$ git stash apply
# On branch master
```

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

Як ви бачите, Git знову змінює файли, які ви відновили коли зберігали хования. У даному випадку, ви мали чисту робочу директорію, коли намагались застосувати хования, та намагались це зробити на тій самій гілці, з якої ви зберігали його. Проте мати чисту робочу директорію та використовувати хования та тій самій гілці не обов'язково для його успішного застосування. Ви можете зберегти хования на одній гілці, потім переключитися на іншу, та спробувати використати зміни там. Також ви можете мати змінені та не збережені файли в робочій директорії, коли ви застосовуєте хования – Git надасть вам конфлікти зливання, якщо використати хования чисто неможливо.

Зміни до ваших файлів повернулись, проте раніше індексовані файли не були знову проіндексовані. Щоб це зробити, треба виконати команду `git stash apply` з опцією `--index`, тоді команда використає та проіндексує зміни. Якби б ви виконали останню команду, ви б повернулися до попереднього стану:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
```

Опція `apply` лише намагається використати сховані зміни – вона все одно залишається в стеку. Щоб видалити його, ви можете виконати `git stash drop` з назвою хования, яке треба видалити:

```
$ 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)
```

Також ви можете виконати `git stash pop` щоб застосувати ховання та відразу видалити його зі стека.

Винахідливе ховання

Є декілька варіантів ховання, що можуть бути корисними. Перша доволі популярна опція — це `--keep-index` команди `stash save`. Вона каже Git не ховати нічого вже проіндексованого за допомогою команди `git add`.

Це може дійсно допомогти, якщо ви зробили багато змін, проте бажаєте зробити коміт тільки з деяких з них, а до інших повернутись пізніше.

```
$ 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
```

Ще одна поширена річ, яку ви можете зробити за допомогою ховання — це сховати не супроводжувані файли разом із супроводжуваними. Без додаткових опцій, `git stash` зберігає тільки файли, які вже знаходяться в індексі. Якщо додати опцію `--include-untracked` або `-u`, Git також сховає будь-які створені не супроводжувані файли.

```
$ 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
$
```

Нарешті, якщо ви поставите `--patch`, Git не буде ховати всі зміни, а замість цього в інтерактивному режимі запитає, які зміни ви бажаєте сховати, а які залишити в робочій директорії.

```
$ 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
```

Створення гілки з хования

Якщо ви сховаєте дещо, залишите його на деякий час, продовжите працю над гілкою, з якої ви робили хования, у вас можуть виникнути проблеми при відновленні схованих змін. Якщо застосування хования спробує змінити файл, який ви змінили, то з'явиться конфлікт зливання та вам доведеться його вирішувати. Якщо ви шукаєте легшого шляху знову подивитись на сховані зміни, то ви можете виконати `git stash branch`, що створить нову гілку, отримає коміт, з якого ви зробили хования, відновить ваші зміни, та видалить хования, якщо воно успішно застосується:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Це зручне скорочення щоб легко відновити сховані зміни та продовжити над ними працювати в новій гілці.

Очищення робочої директорії

Нарешті, можливо ви бажаєте не ховати якісь зміни до файлів у робочій директорії, а просто позбутися їх. Команда `git clean` зробить це.

Серед іншого це може бути корисним для видалення сміття, що було згенеровано зливаннями або зовнішніми утилітами, або для видалення результатів компіляції щоб зробити чисту компіляцію.

Треба бути винятково обережним з цією командою, адже вона створена для видалення файлів з вашої робочої директорії, які не супроводжується. Якщо ви зміните думку, зазвичай не існує методу отримати зміст цих файлів. Безпечніше виконати `git stash --all` щоб видалити все, проте зберегти його у хованні.

Якщо ж ви хочете видалити непотрібні файли, або очистити свою робочу директорію, то це може зробити команда `git clean`. Щоб видалити всі не супроводжувані файли в робочій директорії, виконайте `git clean -f -d`, що видаляє будь-які файли та піддиректорії, які в результаті стають порожніми. Опція `-f` означає *force* (змусити) або “дійсно зроби це”.

Якщо вам більш до вподоби бачити, що ви робите, ви можете виконати цю команду з опцією `-n`, що означає “нічого не видаляй та скажи мені що *б* ти видалив”.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Без додаткових опцій, команда `git clean` видаляє тільки не супроводжувані файли, які не ігноруються. Будь-які файли, що відповідають шаблонам `.gitignore` або інших файлів ігнорування не будуть видалені. Якщо ви бажаєте видалити й ці файли, наприклад видалити всі файли `.o`, що були згенеровані при останній компіляції, щоб зробити повністю чисту компіляцію, ви можете додати опцію `-x` до команди `clean`.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

Якщо ви не впевнені, що буде робити команда `git clean`, завжди спочатку виконайте її з опцією `-n` та двічі все перевірте перш ніж змінювати `-n` на `-f` та дійсно зробити очищення. Інший спосіб бути обережним — зробити процес інтерактивним за допомогою опції `-i`.

Це виконає команду очистки в інтерактивному режимі.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each
    6: help
What now>
```

Таким чином ви можете пройти по кожному файлу окремо, або задати шаблон для видалення в інтерактивному режимі.

Підписання праці

Git криптографічно захищений, проте, не захищений від неправильного користування. Якщо ви отримуєте роботу інших з мережі та бажаєте перевірити, що коміти дійсно з довіреного джерела, Git пропонує декілька шляхів підписання та перевірки праці за допомогою GPG.

Знайомство з GPG

Спершу, якщо ви бажаєте щось підписувати, вам необхідно налаштувати GPG та встановити ваш персональний ключ.

```
$ 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
```

Якщо у вас немає ключа, то вам треба згенерувати його за допомогою `gpg --gen-key`.

```
gpg --gen-key
```

Після здобуття приватного ключа для підпису, ви можете задати значення налаштуванню `user.signingkey`, щоб Git підписував об'єкти за допомогою цього ключа.

```
git config --global user.signingkey 0A46826A
```

Тепер Git буде використовувати цей ключ, щоб підписувати які забажаєте теґи та коміти.

Підписання теґів

Якщо ви налаштували приватний ключ GPG, ви тепер можете його використовувати для підписання нових теґів. Треба тільки замість `-a` писати `-s`.

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

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
```

Якщо ви виконаєте `git show` для цього теґу, то зможете побачити свій GPG підпис:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----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
```

Перевірка теґів

Щоб перевірити підписаний теґ, треба використати `git tag -v [назва теґу]`. Ця команда використовує GPG щоб перевірити підпис. Вам потрібен публічний ключ автора підпису у вашому кільці ключів (keyring), щоб це спрацювало:

```
$ 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
```

Якщо у вас немає публічного ключа автора підпису, то ви отримаєте щось таке:

```
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'
```

Підписання комітів

У новіших версіях Git (v1.7.9 та вище), тепер є можливість також підписувати окремі коміти. Якщо вас цікавить підписання саме комітів, а не просто теґів, усе що вам потрібно зробити — це додати `-S` до команди `git commit`.

```
$ git commit -a -S -m 'signed commit'

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] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Щоб побачити та перевірити підписи, у команди `git log` також є опція `--show-signature`.

```
$ 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

    signed commit
```

Крім того, за допомогою опції формату `%G?`, команда `git log` може перевірити всі підписи, які знайде.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Тут ми бачимо, що лише останній коміт підписаний та справжній, а всі попередні — ні.

Починаючи з Git 1.8.3, з опцією `--verify-signatures` команди “git merge” та “git pull” можуть перевіряти та відмовляти при зливанні коміту, який не несе довіреного підпису GPG.

Якщо ви використаєте цю опцію при зливанні гілки та вона містить непідписані або недійсні коміти, зливання не спрацює.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Якщо зливання містить виключно справжні підписані коміти, зливання покаже вам усі перевірені підписи та перейде до власно зливання.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Ви також можете використовувати опцію `-S` із самою командою `git merge`, щоб підписати створюваний коміт злиття. Наступний приклад перевіряє підпис кожного коміту гілки, з якої ми зливаємо, та більш того підписує створений коміт зливання.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon

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(+)
```

Усі мусять підписувати

Підписання теґів та комітів це чудово, проте, якщо ви вирішите це використовувати у своєму нормальному процесі роботи, ви маєте переконатися, що кожен з вашої команди розуміє як підписувати. Якщо ви цього не зробите, то вам доведеться витратити купу часу на пояснення, як переписати їх коміти на підписану версію. Переконайтесь, що ви розумієте GPG та переваги підписання речей до того, як додасте це до вашого прийнятого процесу роботи.

Пошук

У базі коду, майже будь-якого розміру, часто потрібно з'ясувати, де функцію викликають, або де вона визначена, або знайти історію методу. Git пропонує декілька корисних інструментів для швидкого та легкого пошуку в коді та комітах, що були збережені в базі даних Git. Ми розглянемо деякі з них.

Git Grep

Git має команду під назвою `grep`, що дозволяє легко шукати в будь-якому дереві коміту або робочій теці заданий рядок або за регулярним виразом. У наступних прикладах ми будемо шукати в коді самого Git.

Без додаткових опцій, `grep` шукає тільки у файлах вашої робочої директорії. Якщо додати опцію `-n`, буде виведено номери рядків, в яких Git знайшов збіг.

```
$ 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
```

У команди `grep` є багато цікавих опцій.

Наприклад, замість попереднього виклику, можна отримати від Git підсумок, що показує в яких файлах є збіги та скільки їх у кожному файлі, за допомогою опції `--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
```

Якщо бажаєте дізнатись, в якому методі чи функції ймовірно знайдено збіг, то скористайтесь `-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)) {
```

Отже тут ми можемо бачити, що `gmtime_r` викликається з функцій `match_multi_number` та `match_digit` у файлі date.c.

Також можна шукати складні комбінації рядків за допомогою опції `--and`, яка надає можливість шукати декілька збігів разом в одному рядку. Наприклад, пошукаймо рядки, що визначають константу з рядками “LINK” або “BUF_MAX” в них у коді Git у версіях старіших за 1.8.0.

Ми також використаємо опції `--bread` та `--heading`, які допомагають розділити вивід для легшого сприйняття.

```
$ 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 */
```

Команда `git grep` має декілька переваг над звичайними пошуковими командами `grep`, `ack` тощо. По-перше, вона дійсно швидка, по-друге, за її допомогою можна шукати в будь-якому дереві Git, а не тільки в робочій директорії. Як ми бачили в останньому прикладі, ми шукали щось у старіших версіях коду Git, а не в поточній вибраній версії.

Пошук у журналі Git

Напевно вас цікавить не тільки **де** щось існує, а ще й **коли** воно існувало або з'явилося. Команда `git log` пропонує декілька потужних інструментів для пошуку окремих комітів за змістом їх повідомлень або навіть змістом різниці, яку вони додали.

Якщо ви бажаєте дізнатись, наприклад, коли константа `ZLIB_BUF_MAX` з'явилася, ви можете попросити Git показати виключно коміти, які або додали, або вилучили рядок, за допомогою опції `-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
```

Якщо ви подивитесь на зміни цих комітів, то побачите що в `ef49a7a` константа була додана, а в `e01503b` вона була змінена.

Якщо вам треба бути точнішим, то ви можете використати регулярний вираз для пошуку за допомогою опції `-G`.

РЯДКОВИЙ ПОШУК У ЖУРНАЛІ

Ще однин доволі складний пошук журналу, що може бути дивовижно корисним — це рядковий пошук історії. Це доволі новий функціонал і ще не дуже широковживаний, проте він може бути дійсно корисним. Його можна викликати за допомогою опції `-L` команди `git log`, і тоді вам буде показана історія функції або рядка коду вашої бази коду.

Наприклад, якщо ми бажаємо побачити кожну зміну функції `git_deflate_bount` з файлу `zlib.c`, то ми можемо виконати `git log -L :git_deflate_bound:zlib.c`. Тоді Git спробує зрозуміти, де межі цієї функції та буде проглядати історію, ті покаже нам кожну зміну, що була зроблена в цій функції у вигляді послідовності патчів аж до моменту створення цієї функції.

```
$ 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);
+}
+
```

Якщо Git не може знайти функцію чи метод вашої мови програмування, ви також можете надати регулярний вираз. Наприклад, ця команда має зробити те ж саме: `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. Ви також можете дати інтервал рядків або один номер рядка, щоб побачити подібний вивід.

Переписування історії

Часто, працюючи з Git, хочеться переробити історію комітів з якоїсь причини. Одна з чудових рис Git — він дозволяє робити рішення якнайпізніше. Ви можете вирішити, які файли до якого коміту потраплять, доки не зробите коміт з області індексування, можете вирішити, що ви не збираєтесь поки що працювати над чимось за допомогою хования (stash), та можете переписувати вже збережені коміти, щоб вони виглядали так, нібито вони були збережені іншим чи ном. Це може включати зміну порядку комітів, зміну повідомлень чи редагування файлів в коміті, зварювання (squash) комітів разом або розбивання комітів на частини, або повне вилучення комітів – все доки ви надасте доступ до вашої роботи іншим.

У цій секції ми розглянемо як виконати всі ці корисні задачі, щоб ви мали можливість зробити історію комітів саме такою, як ви бажаєте, до того, як поділитися нею з іншими.

Зміна останнього коміту

Зміна останнього коміту напевно є найбільш розповсюдженою операцією переписування історії. Часто вам хочеться зробити дві прості речі з вашим останнім комітом: змінити повідомлення коміту, або змінити щойно записаний знімок: додати, змінити або вилучити файли.

Якщо ви бажаєте лише виправити повідомлення останнього коміту, то це зробити дуже просто:

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

Це команда запустить текстовий редактор, в якому вже буде готове для редагування повідомлення останнього коміту. Коли ви збережете файл та закриєте редактор, Git створить новий коміт з цим повідомленням та зробить його вашим новим останнім комітом.

Якщо ви зробили коміт, а потім захотіли змінити зроблений знімок додаванням або редагуванням файлів, можливо через те що ви забули додати щойно створений файл, коли робили коміт, це можна зробити майже так само. Треба проіндексувати зміни, які ви бажаєте: відредагуйте файл та виконайте `git add` або виконайте `git rm` на супроводжуваному файлі, і наступний `git commit --amend` візьме вашу поточну область індексування, та зробить з неї знімок для нового коміту.

Треба бути обережним з цим засобом, адже ці покращення змінюють SHA-1 коміту. Це як дуже маленьке перебазування (rebase) – не змінюйте свій останній коміт, якщо ви його вже кудись виклали.

Зміна декількох повідомлень комітів

Щоб змінити коміт, що записаний раніше в історії, необхідно використати складніші інструменти. Git не має окремого інструменту зміни-історію, проте можна використати утиліту перебазування (rebase) щоб перебазувати послідовність комітів на HEAD, на якому вони і так базувались, замість переміщення на іншу базу. За допомогою інтерактивного перебазування можна зупинитись після коміту, який ви бажаєте змінити, та змінити повідомлення, додати файли - що забажаєте. Інтерактивне перебазування можна запустити за допомогою опції `-i` команди `git rebase`. Необхідно зазначити як далеко назад ви бажаєте переписувати коміти, для чого треба сказати команді, на який коміт перебазуватися.

Наприклад, якщо ви бажаєте змінити останні три коміти, або будь-які повідомлення комітів у цій групі, то треба задати як аргумент `git rebase -i` батька останнього коміту, який ви бажаєте редагувати, тобто `HEAD~2^` або `HEAD~3`. Можливо легше запам*ятати ~3, адже це дозволяє редагувати три останні коміти. Проте пам*ятайте, що насправді цей запис означає чотири коміти тому, батька останнього коміту, якого ви бажаєте редагувати:

```
$ git rebase -i HEAD~3
```

Не забувайте також, що це команда перебазування – кожен коміт в інтервалі HEAD~3..HEAD буде переписаний, зміните ви його повідомлення чи ні. Не включайте жодного коміту, який ви вже відправили до центрального серверу – інакше ви заплутаєте інших розробників тим, що надасте альтернативну версію тих самих змін.

Виконання цієї команди відкриває ваш текстовий редактор зі списком комітів, що виглядає приблизно так:

```
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 (використати коміт)
#  r, reword = use commit, but edit the commit message (використати коміт, проте р
#  e, edit = use commit, but stop for amending (використати коміт, але зупинитись
#  s, squash = use commit, but meld into previous commita (використати коміт, прот
#  f, fixup = like "squash", but discard this commit's log message (як "squash", п
#  x, exec = run command (the rest of the line) using shell (виконати команду (реш
#
# These lines can be re-ordered; they are executed from top to bottom. (Ці рядки м
#
# If you remove a line here THAT COMMIT WILL BE LOST. (Якщо вилучити рядок, ЦЕЙ КО
#
# However, if you remove everything, the rebase will be aborted. (Втім, якщо видал
#
# Note that empty commits are commented out (Зауважте, що пусті коміти закоментова
```

Важливо зазначити, що коміти йдуть у зворотному порядку, а не так, як ви звикли їх бачити при використанні команди log. Якщо виконаймо log, то побачимо щось таке:

```
$ 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
```

Зауважте зворотний порядок. Інтерактивне перебазування дає вам скрипт, який збирається виконувати. Воно почне з коміту, який ви

зазначите в командному рядку (HEAD~3) та відтворить зміни внесені кожним з комітів зміни згори донизу. Воно видає найстаріший нагорі, а не найновіший, адже він є першим, що буде відтворений.

Вам треба відредагувати скрипт щоб він зупинився на коміті, який ви бажаєте редагувати. Для цього змініть слово ‘pick’ (підібрати) на ‘edit’ (редагувати) для кожного з комітів, після яких треба зупинитися. Наприклад, щоб змінити тільки повідомлення третього коміту, можна змінити файл наступним чином:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Коли ви збережете файл та вийдете з редактора, Git поверне вас до останнього коміту в цьому списку та надасть вам командний рядок з наступним повідомленням:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit (Зупинився після f7f3f6d...)
You can amend the commit now, with (Тепер ви можете змінити коміт за допомогою)

       git commit --amend

Once you’re satisfied with your changes, run (Щойно ви задоволені своїми змінами, виконайте)

       git rebase --continue
```

Ця інструкція докладно розповідає, що робити. Наберіть

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

Змініть повідомлення коміту та вийдіть з редактору. Потім виконайте

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

Ця команда застосує решту два коміти автоматично — і все готово. Якщо ви змініте pick на edit ще в якомусь рядку, то можете повторити ці кроки для кожного коміту, для якого набрали edit. Щоразу Git

зупинятиметься, дозволить вам змінити коміт, та продовжить, доки ви не завершите перебазування.

Зміна послідовності комітів

Інтерактивне перебазування комітів можна також використовувати для зміни порядку комітів або їх повного вилучення. Якщо ви бажаєте вилучити коміт “added cat-file” та змінити послідовність решти двох комітів, то можете змінити скрипт перебазування з такого

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

на такий:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Коли ви збережете файл та вийдете з редактора, Git поверне гілку до батька цих комітів, застосує `310154e`, потім `f7f3f6d` та зупиниться. Таким чином ви змінили послідовність цих комітів та повністю вилучили коміт “added cat-file”.

Зварювання комітів

Також можливо взяти декілька комітів та зварити їх в один коміт за допомогою інтерактивного перебазування. Скрипт має корисні інструкції в повідомленні перебазування:

```
#
# Commands:
#  p, pick = use commit (використати комiт)
#  r, reword = use commit, but edit the commit message (використати комiт, проте р
#  e, edit = use commit, but stop for amending (використати комiт, але зупинитись
#  s, squash = use commit, but meld into previous commita (використати комiт, прот
#  f, fixup = like "squash", but discard this commit's log message (як "squash", г
#  x, exec = run command (the rest of the line) using shell (виконати команду (реш
#
# These lines can be re-ordered; they are executed from top to bottom. (Ці рядки м
#
# If you remove a line here THAT COMMIT WILL BE LOST. (Якщо вилучити рядок, ЦЕЙ КО
```

```
#
# However, if you remove everything, the rebase will be aborted. (Втім, якщо видалити все, п
#
# Note that empty commits are commented out (Зауважте, що пусті коміти закоментовані)
```

Якщо замість “pick” або “edit” ви задасте “squash”, Git застосує і цю зміну, і зміну безпосередньо до неї, до того як дозволити вам злити повідомлення комітів разом. Отже, якщо ви бажаєте зробити один коміт з цих трьох комітів, то треба зробити скрипт таким:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Коли ви збережете файл та вийдете з редактору, Git застосує всі три зміни та поверне вас у редактор для злиття трьох повідомлень комітів:

```
# This is a combination of 3 commits. (Це комбінація 3 комітів)
# The first commit's message is: (Повідомлення першого коміту)
changed my name a bit

# This is the 2nd commit message: (Повідомлення другого коміту)

updated README formatting and added blame

# This is the 3rd commit message: (Повідомлення третього коміту)

added cat-file
```

Коли ви збережете цей файл, отримаєте єдиний коміт, що вносить зміни всіх трьох попередніх комітів.

Розщеплення коміту

Розщеплення коміту скасовує коміт, а потім частково індексує та зберігає коміти стільки разів, скільки ви бажаєте створити комітів. Наприклад, припустімо, що ви бажаєте розщепити середній з трьох комітів. Замість “updated README formatting and added blame” треба зробити два коміти: перший — “updated README formatting”, та другий — “added blame”. Це можна зробити в скрипті `rebase -i`, якщо змінити інструкцію біля коміту, котрий треба розщепити, на “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Потім, коли скрипт повернеться до командного рядку, треба скинути (reset) цей коміт, взяти скинуті зміни, та створити з них декілька комітів. Коли ви збережете файл та вийдете з редактору, Git повернеться до батька першого коміту зі списку, застосує перший коміт (`f7f3f6d`), застосує другий (`310154e`) та поверне вас до консолі. Там ви можете зробити скидання коміту за допомогою команди `git reset HEAD^`, яка фактично скасовує коміт та залишає змінені файли не індексованими. Тепер ви можете індексувати файли та робити коміти, доки не отримаєте декілька комітів, а потім виконати `git rebase --continue` щойно все готово:

```
$ 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 застосує останній коміт (`a5f4a0d`) зі скрипту, і історія виглядатиме так:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Нагадуємо, що це змінить SHA-1 кожного з комітів вашого списку, отже треба бути впевненим, що жоден з комітів списку ще не був відправлений до спільного сховища.

Ядерний варіант: filter-branch

Є ще одна опція переписування історії, яку можна використовувати, якщо треба переписати більшу кількість комітів у якийсь скриптований спосіб – наприклад, змінити вашу поштову адресу глобально, або

вилучити файл з усіх комітів. Команда називається `filter-branch`, і вона може переписати величезні частини історії, отже напевно не варто його використовувати, якщо ваш проект вже є в загальному доступі чи хтось інший працює з комітами, які ви збираєтесь переписати. Втім, вона може бути дуже корисно. Ви дізнаєтесь про декілька найпоширеніших застосувань, щоб отримати загальну ідею про те, що може робити цей інструмент.

ВИЛУЧЕННЯ ФАЙЛА З КОЖНОГО КОМІТУ

Таке трапляється скрізь. Хтось випадково зберігає величезний двійковий файл бездумним `git add .`, та ви бажаєте видалити його скрізь. Чи може ви випадково зберегли файл, що містить пароль, та бажаєте зробити проект відкритим. `filter-branch` саме потрібний вам інструмент, якщо ви бажаєте відшкребти щось з усієї історії. Щоб вилучити файл passwords.txt з усієї історії, можна використати опцію `--tree-filter` команди `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Опція `--tree-filter` виконує задану команду після кожного коміту проекту, після чого знову робить коміт результатів. У даному випадку, ви вилучаєте файл під назвою passwords.txt з кожного знімку, існував він чи ні. Якщо ви бажаєте вилучити всі випадково збережені файли резервних копій, то можете виконати щось на кшталт `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Ви зможете спостерігати, як Git переписує дерева та коміти, після чого пересуває вказівник гілки. Зазвичай розумно робити це в тестовій гілці, після чого робити жорстке скидання (hard-reset) вашої гілки master, коли ви переконаєтесь, що результат відповідає запланованому. Щоб виконати `filter-branch` на всіх гілках, передайте команді `--all`.

РОБИМО ПІДДИРЕКТОРІЮ НОВИМ КОРЕНЕМ

Припустімо, що ви зробити імпорт з іншої системи контролю версій та маєте піддиректорії, що не мають сенсу (trunk, tags тощо). Якщо ви бажаєте зробити піддиректорію `trunk` новим коренем проекту для кожного коміту, то вам теж потрібен `filter-branch`:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Тепер коренем вашого проекту стало те, що було в піддиректорії `trunk` для кожного коміту. Git автоматично вилучить коміти, що не впливали на піддиректорію.

ГЛОБАЛЬНА ЗМІНА ПОШТОВОЇ АДРЕСИ

Ще один поширений випадок: ви могли забути виконати `git config` щоб задати своє ім'я та поштову адресу до початку роботи, або можливо бажаєте відкрити проект з роботи та змінити вашу робочу поштову адресу на особисту адресу. У цьому випадку, можна змінити поштову адресу декількох комітів автоматично за допомогою `filter-branch`. Треба бути обережним, щоб змінити тільки вашу адресу, отже використовуймо `--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
```

Це проходить та змінює кожен коміт, який має вашу адресу. Оскільки коміти містять значення SHA-1 своїх батьків, ця команда змінює SHA-1 кожного коміту вашої історії, а не лише тих, що мають збіг поштової адреси.

Усвідомлення скидання (reset)

До того, як переходити до більш спеціалізованих інструментів, поговорімо про `reset` (скинути) та `checkout` (отримати). Ці дві команди найбільше збивають з пантелику, особливо, коли ви вперше ними користуєтесь. Вони роблять так багато всього, що спроба дійсно зрозуміти їх та використовувати правильно здається безнадійною. Щоб усе ж таки це зробити, ми пропонуємо просту метафору.

Три дерева

Набагато легше зрозуміти `reset` та `checkout`, якщо уявити, що Git керує вмістом трьох різних дерев. “Деревом” ми тут будемо називати “колекцію файлів”, а не саме структуру даних. (Є декілька випадків, в яких індекс насправді не поводиться як дерево, проте легше поки що уявляти його так.)

Git як система керує та маніпулює трьома деревами під час нормальної роботи:

Дерево	Роль
HEAD	Знімок останнього коміту, наступний батько
Індекс	Пропонований знімок наступного коміту
Робоча Директорія	Пісочниця

HEAD

HEAD є вказівником на посилання поточної гілки, яка у свою чергу вказує на останній коміт, що був зроблений у цій гілці. Це означає, що HEAD буде батьком наступного створеного коміту. Зазвичай найпростіше думати про HEAD як про знімок **останнього коміту**.

Насправді, доволі легко побачити, як цей знімок виглядає. Ось приклад отримання списку файлів директорії та SHA-1 суми кожного файла в знімку 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
```

Команди `cat-file` та `ls-tree` є командами низького рівня, якими не дуже користуються в повсякденній роботі, проте вони корисні при з’ясуванні того, що насправді коїться.

ІНДЕКС

Індекс — це **пропозиція наступного коміту**. Ця концепція Git також має назву “Область Додавання”, адже саме сюди дивиться Git при виконанні `git commit`.

Git заповнює цей індекс списком усього вмісту файлів, що був отриманий зі сховища до вашої робочої теки востаннє та яким він тоді був. Потім ви замінюєте деякі з цих файлів новими версіями, та `git commit` перетворює це на дерево для нового коміту.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0       README
100644 8f94139338f9404f26296befa88755fc2598c289 0       Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0       lib/simplegit.rb
```

Ми знову скористалися `ls-files`, яка зазвичай виконується за лаштунками та показує як наразі виглядає індекс.

Технічно, індекс не є деревом – насправді його реалізовано як сплощений маніфест, проте для нас це достатньо близько.

РОБОЧА ДИРЕКТОРІЯ

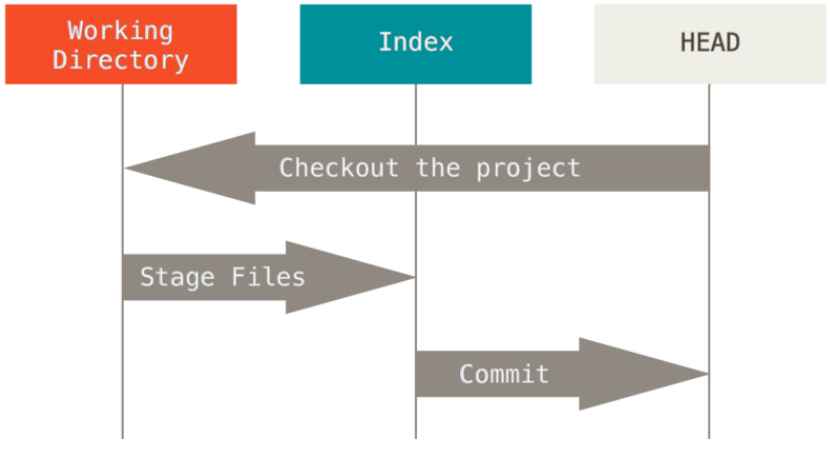
Нарешті, є ваша робоча директорія. Інші два дерева зберігають свій вміст у ефективний проте незручний спосіб: усередині теки `.git`. Робоча Директорія розпаковує їх до реальних файлів, що дозволяє їх редагувати набагато легше. Вважайте Робочу Директорію **пісочницею**, де ви можете спробувати зміни до того, як додати їх до індексу, а потім до історії.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

1 directory, 3 files
```

Робочий процес

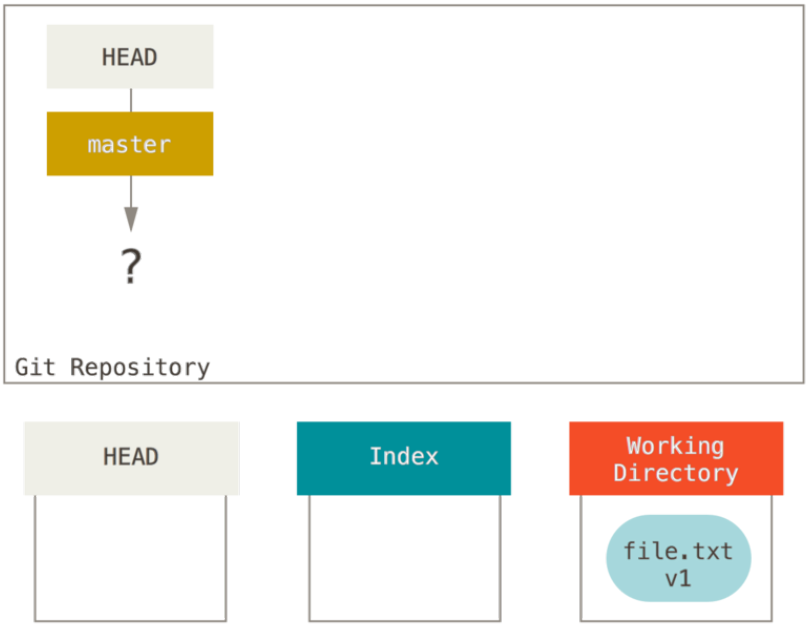
Головне призначення Git - записувати знімки вашого проекту в послідовно кращих станах за допомогою маніпулювання цими трьома деревами.



ЗОБРАЖЕННЯ 7-2

Уявімо собі цей процес: припустімо, ви переходите до нової директорії з єдиним файлом у ній. Ми назвемо цю версію файла **v1**, та будемо позначати її синім. Тепер виконаємо `git init`, що створить сховище Git з посиланням HEAD, що вказує на ненароджену гілку (`master` ще не існує).

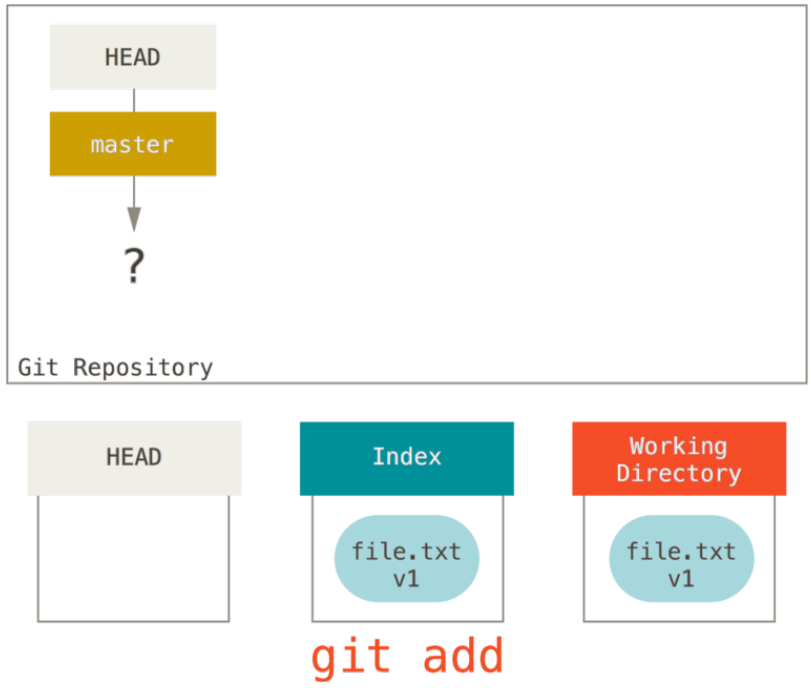
ЗОБРАЖЕННЯ 7-3



Наразі тільки в дереві Робочої Директорії є якийсь вміст.

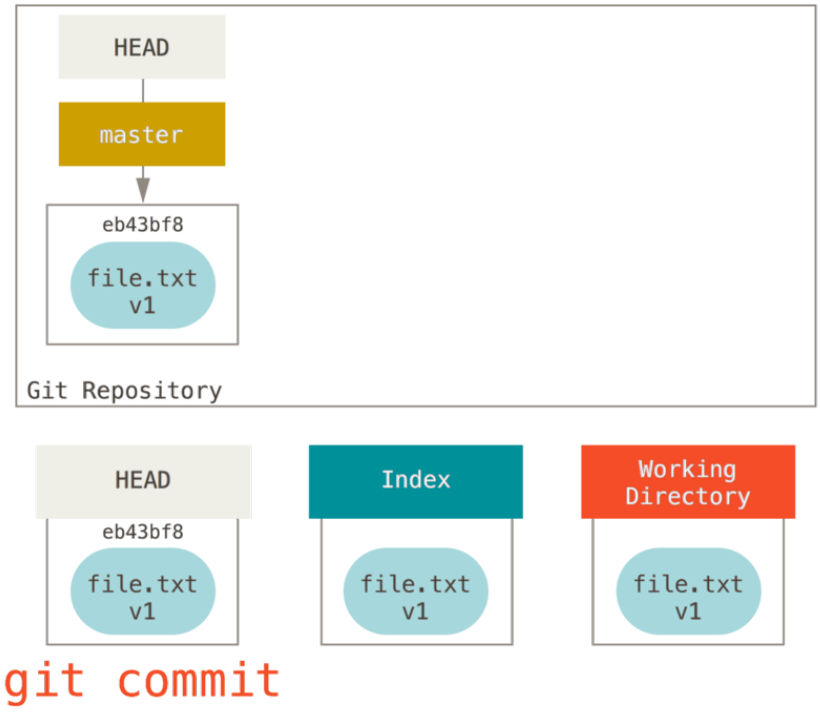
Тепер ми бажаємо зробити коміт з цим файлом, отже ми використовуємо `git add` щоб взяти вміст з Робочої Директорії та скопіювати його до Індексу.

ЗОБРАЖЕННЯ 7-4



Потім виконуємо `git commit`, що бере вміст Індексу та зберігає його в незмінному знімку, створює об'єкт коміту, що вказує на цей знімок, та оновлює `master`, щоб той вказував на цей коміт.

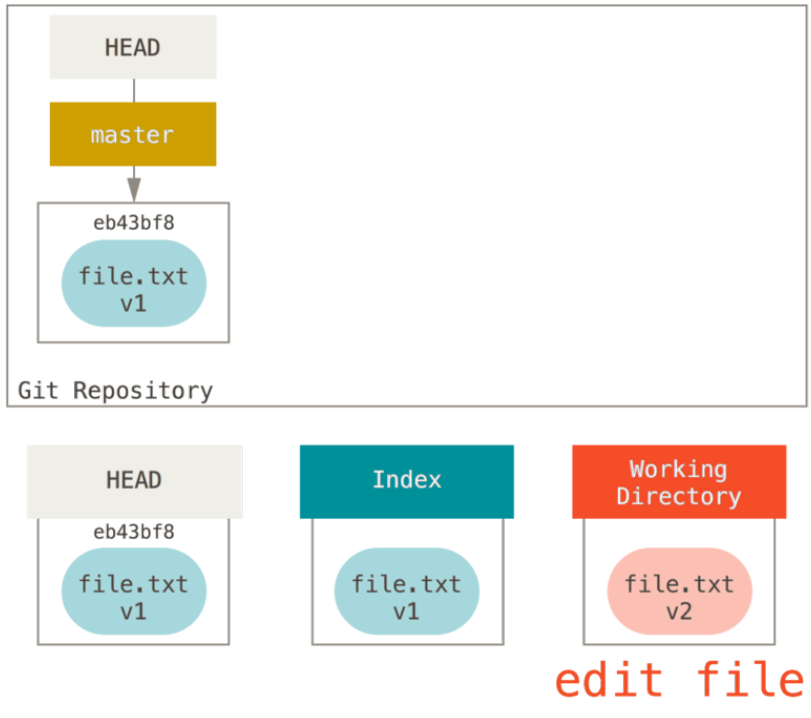
ЗОБРАЖЕННЯ 7-5



Якщо ми виконаємо `git status`, то не побачимо ніяких змін, адже всі три дерева однакові.

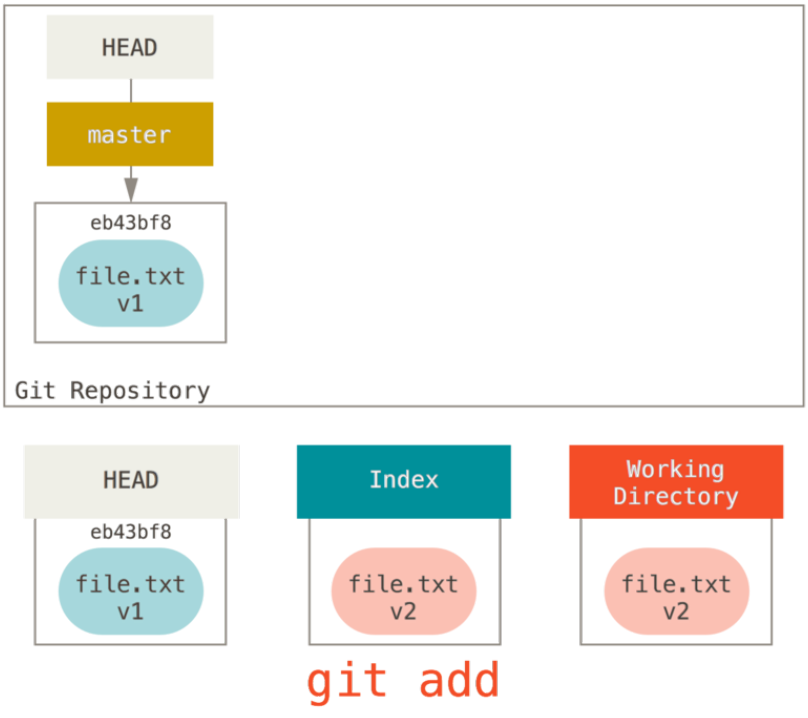
Тепер ми хочемо зробити зміну в цьому файлі та зберегти їх у коміті. Ми пройдемо той самий процес. Спочатку змінимо файл у робочій директорії. Назвемо цю версію файла **v2**, та позначимо її червоним.

ЗОБРАЖЕННЯ 7-6



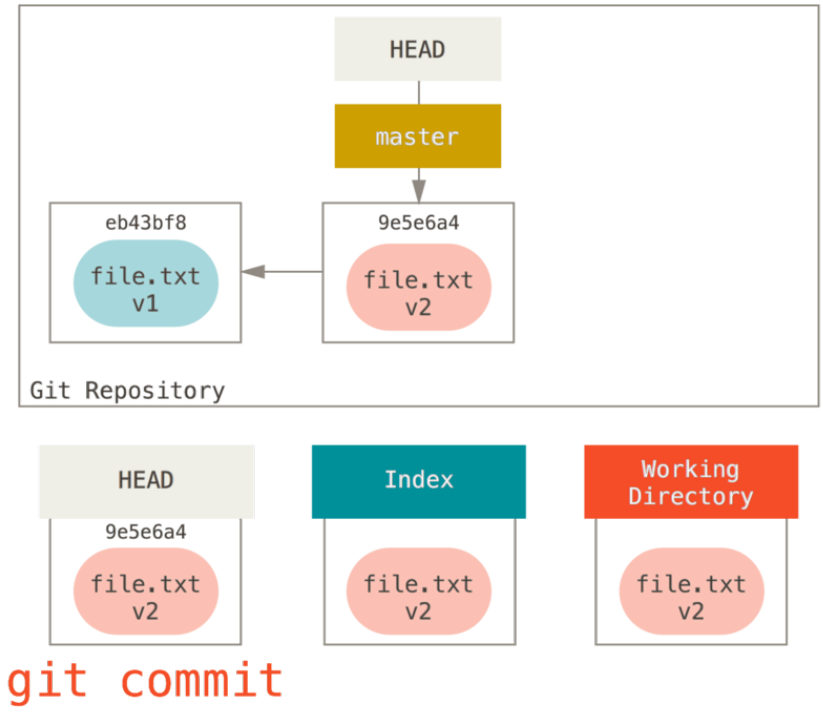
Якщо ми зараз виконаємо `git status`, то побачимо файл червоним у “Changes not staged for commit”, адже в цього елемента є різниця між Індексом та Робочою Директорією. Далі виконуємо `git add` на ньому, щоб додати його до Індекса.

ЗОБРАЖЕННЯ 7-7



Тепер якщо ми виконаємо `git status`, то побачимо файл зеленим. під “Changes to be commited”, адже Індекс та HEAD різняться - тобто, наш пропонований наступний коміт зараз відрізняється від останнього коміту. Нарешті, виконуємо `git commit` щоб завершити коміт.

ЗОБРАЖЕННЯ 7-8



Тепер `git status` нічого не виведе, адже всі три дерева знову однакові.

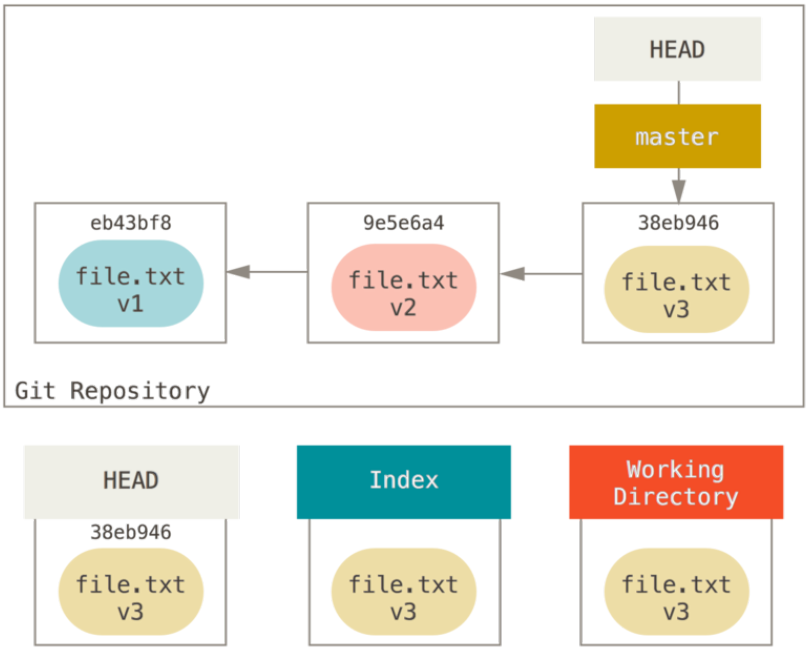
Переключення гілок та клонування проходить за схожим процесом. Коли ви отримуєте (checkout) гілку, Git перенаправляє **HEAD** до нового посилання гілки, заповнює **Індекс** знімком того коміту, далі копіює вміст **Індексу** до **Робочої Директорії**.

Роль скидання (reset)

Команду `reset` легше зрозуміти в цьому контексті.

Задля наступних прикладів, скажімо ми змінили файл `file.txt` знову та зробили третій коміт. Отже тепер наша історія виглядає так:

ЗОБРАЖЕННЯ 7-9

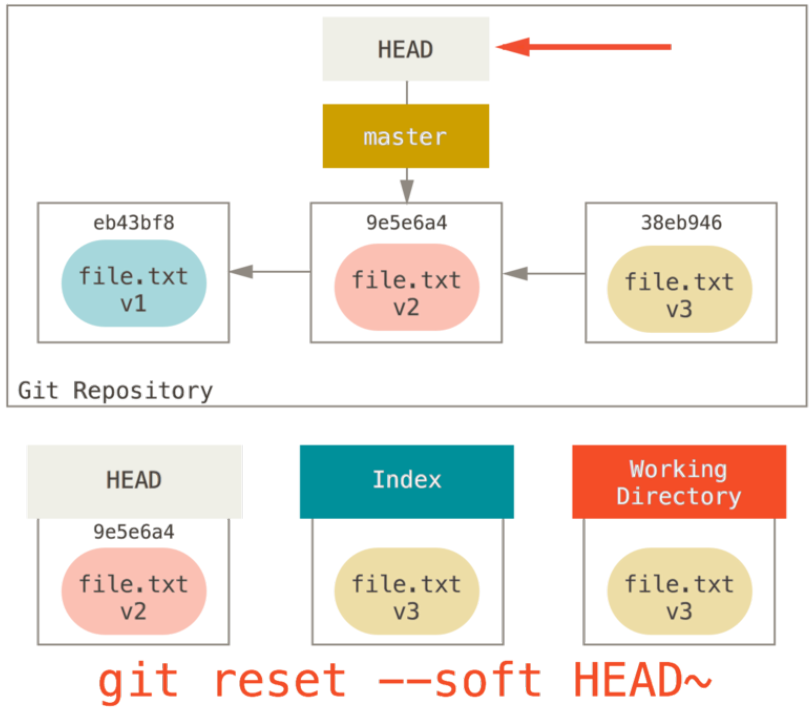


Розгляньмо ґрунтовно що саме робить команда `reset` при виклику. Вона безпосередньо змінює ці три дерева в простий та передбачуваний спосіб. Вона здійснює три базові операції.

КРОК 1: ПЕРЕМІСТИТИ **HEAD**

Спершу `reset` перемістить те, на що вказує HEAD. Це не те саме, що змінити сам HEAD (це робить `checkout`). `reset` пересуває гілку, на яку вказує HEAD. Це означає, що якщо HEAD вказує на гілку `master` (тобто гілка `master` є поточною), то виконання `git reset 9e5e6a4` почнеться зі зміни вказівника `master` так, що він буде вказувати на `9e5e6a4`.

ЗОБРАЖЕННЯ 7-10



Байдуже яку форму `reset` викликано, все одно це перше, що команда спробує зробити. Виклик `reset --soft` просто зупиниться на цьому.

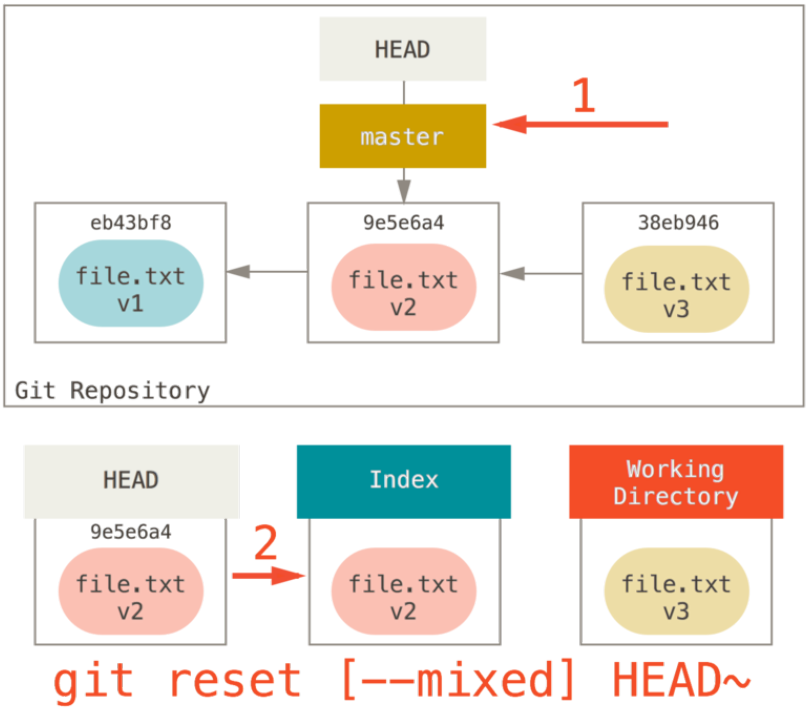
Тепер подивіться хвильку на зображення та усвідомте, що сталося: суттєво остання команда `git commit` була скасована. При виконанні `git commit`, Git створює новий коміт та пересуває до нього гілку, на яку вказує HEAD. Коли ви робите `reset` назад до HEAD~ (батько HEAD), ви повертаєтесь туди, де були, без зміни Індексу чи Робочої Директорії. Тепер можна оновити Індекс та знову виконати `git commit`, щоб здійснити те, що можна зробити за допомогою `git commit --amend` (дивіться “Зміна останнього коміту”).

КРОК 2: ОНОВЛЕННЯ ІНДЕКСУ (--MIXED)

Зауважте, що при виконанні `git status` ви побачити зеленим різницю між Індексом та новим HEAD.

Наступне, що зробить `reset` — оновить Index вмістом знімку, на який тепер вказує HEAD.

ЗОБРАЖЕННЯ 7-11



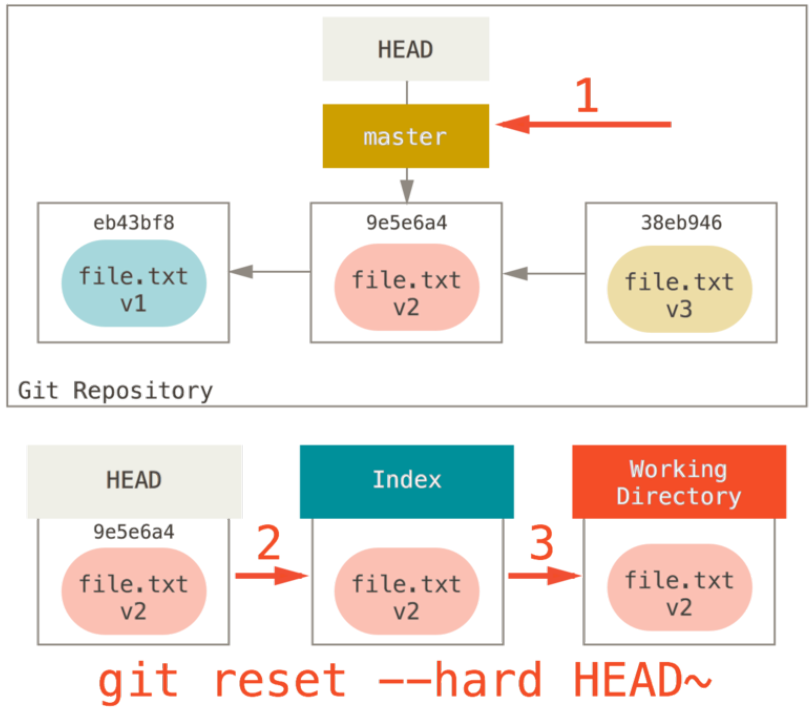
Якщо ви використаєте опцію `--mixed`, то `reset` на цьому зупиниться. Те саме буде і без опції, отже якщо ви не будете надавати ніяких опцій (просто `git reset HEAD~` у даному випадку), тут команда і зупиниться.

Тепер подивіться ще хвильку на з це зображення, щоб зрозуміти що сталося: ви все одно скасували останню `commit`, проте також *деіндексували* все. Ви відкотили до стану до виконання `git add` та `git commit`.

КРОК 3: ОНОВЛЕННЯ РОБОЧОЇ ТЕКИ (--HARD)

Третє, що робить `reset` — змушує Робочу Директорію виглядати як Індекс. Якщо ви використаєте опцію `--hard`, Git виконає і цей крок.

ЗОБРАЖЕННЯ 7-12



Поміркуймо що щойно сталося. Ви скасували останній коміт, команди `git add` та `git commit`, **та** всю працю, яку ви робили у вашій робочій директорії.

Важливо зазначити, що ця опція (`--hard`) єдиний шлях зробити `reset` небезпечним, і один з дуже нечисленних випадків, в яких Git може дійсно знищити дані. Будь-який інший виклик `reset` можна доволі легко скасувати, проте опцію `--hard` не можна, адже вона насилу переписує файли в Робочій Директорії. У даному окремому випадку, ми досі маємо версію **v3** нашого файлу в коміті в нашій базі даних Git, і могли б відновити її за допомогою `reflog`, проте якби б ми не зробили були коміт, Git все одно переписав би файли та ми ніяк не змогли б відновити нашу працю.

КОРОТКИЙ ПІДСУМОК

Команда `reset` переписує ці три дерева у заданому порядку, та зупиняється де ви їй скажете:

1. Пересунути гілку, на яку вказує HEAD *(зупинитися тут, якщо `--soft`)*
2. Зробити так, щоб Індекс виглядав як HEAD *(зупинитися тут, якщо не `--hard`)*
3. Зробити так, щоб Робоча Директорія виглядала як Індекс

Скидання зі шляхом

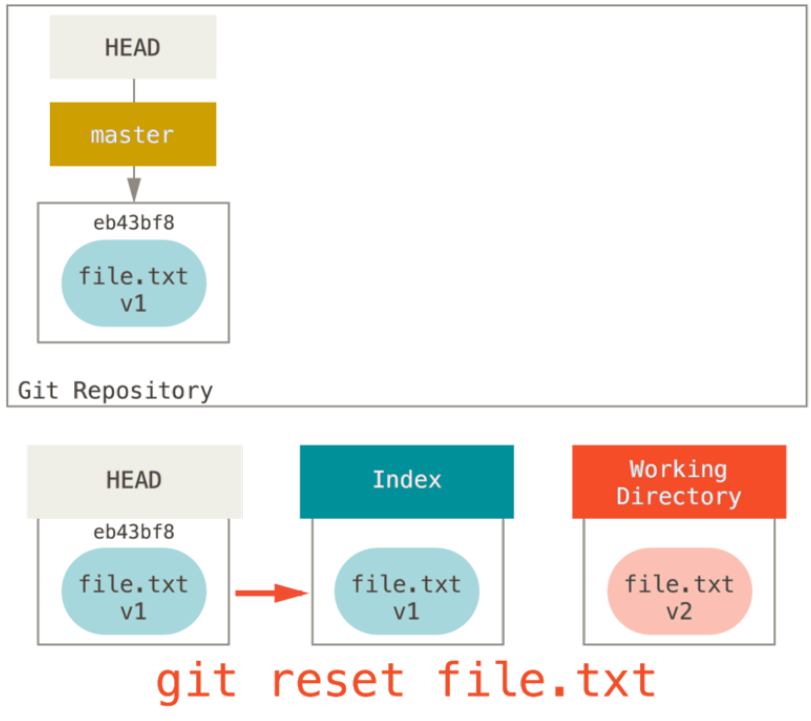
Ми розглянули поведінку `reset` у базовому випадку, проте їй також можна передати шлях, що його треба обробити. Якщо задати шлях, `reset` пропустить крок 1, та обмежить решту своїх дій файлом або декількома файлами. Це доволі розумно – HEAD є просто вказівником та не може вказувати на частину одного коміту та частину іншого. Проте Індекс та Робоча директорія *можуть* бути частково оновленими, отже скидання йде далі до кроків 2 та 3.

Отже, припустіть, що ми виконали `git reset file.txt`. Цей формат (адже ви не вказали SHA-1 коміту або гілку, та не задали `--soft` або `--hard`) є скороченням `git reset --mixed HEAD file.txt`, що:

1. Пересунути гілку, на яку вказує HEAD *(пропущено)*
2. Зробити так, щоб Індекс виглядав як HEAD *(зупинитися тут)*

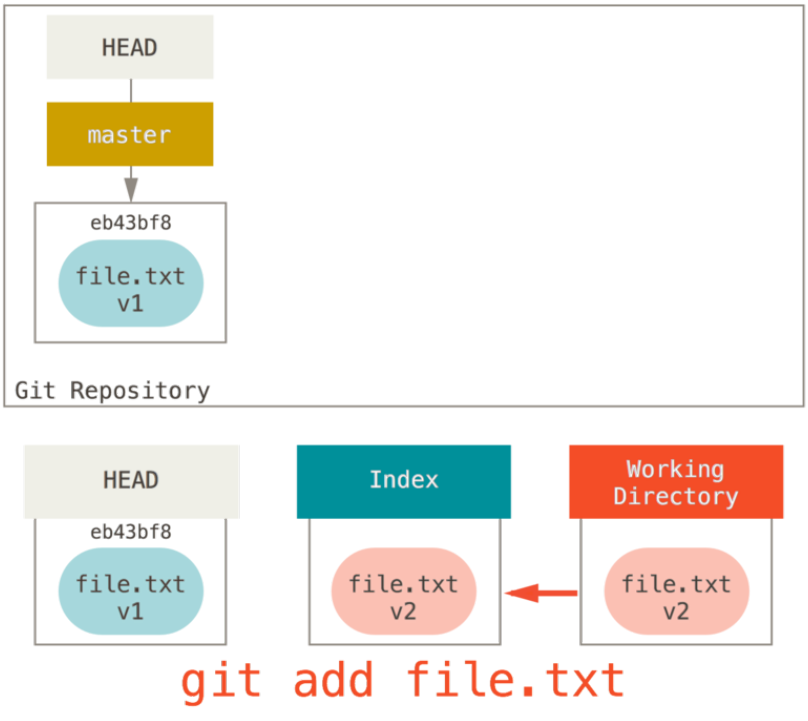
Отже суттєво просто копіює `file.txt` з HEAD до Індексу.

ЗОБРАЖЕННЯ 7-13



Це має ефект *деіндексації* файла. Якщо подивитися на зображення, та згадати, що робить `git add`, то зрозуміло що вони роблять цілковито протилежні речі.

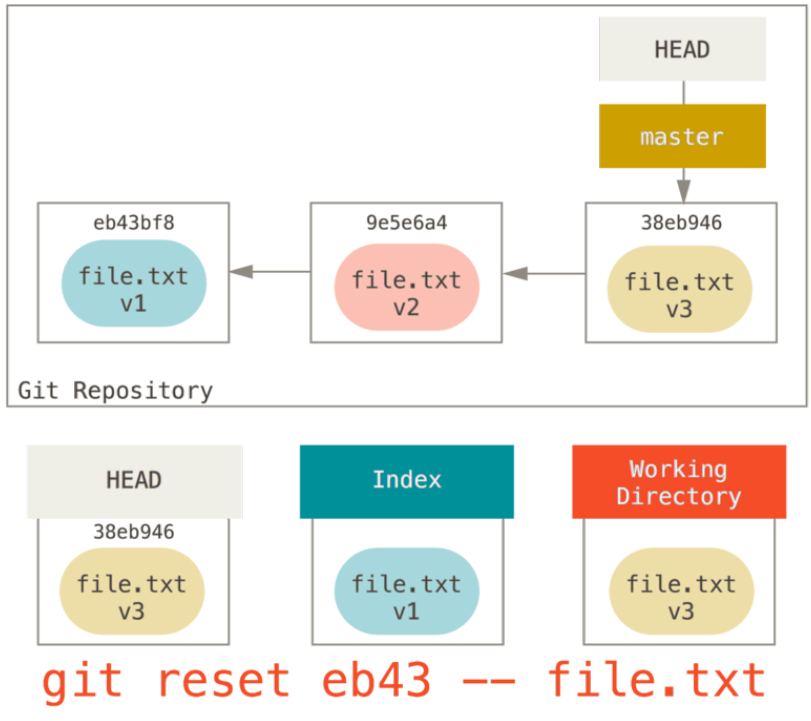
ЗОБРАЖЕННЯ 7-14



Ось чому вивід команди `git status` пропонує виконувати цю команду для деіндексації файла. (Докладніше в **“Вилучання файла з індексу”**.)

Ми могли б так же легко не давати Git припускати, що ми маємо на увазі “візьми дані з HEAD”, якби б задали окремий коміт, з якого треба брати версію. Ми могли просто виконати щось на кшталт `git reset eb43bf file.txt`.

ЗОБРАЖЕННЯ 7-15



Ця досягає такого ж ефекту, ніби ми повернули вміст файлу до **v1** у Робочій Директорії, виконали на ньому `git add`, а потім повернули його вміст до **v3** знов (тільки без проходження всіх цих кроків). Якщо тепер виконати `git commit`, буде записано зміну, що повертає файл до стану **v1**, хоча ми ніколи не мали його в такому стані в Робочій Дирикторії.

Також цікаво знати, що як і команда `git add`, `reset` приймає опцію `--patch` щоб деіндексувати вміст файлу по шматкам. Отже ви можете вибірково деіндексувати або скасовувати вміст.

Зварювання (squashing)

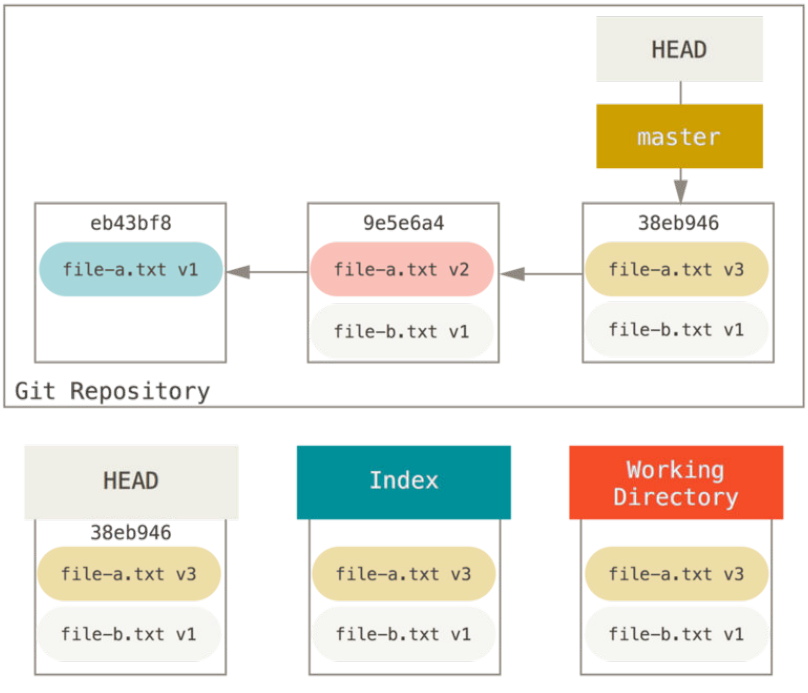
Подивімося на те, як можна зробити щось цікаве цією винайденою можливістю – зварювання комітів.

Припустімо у вас є послідовність комітів з повідомленнями “oops.”, “WIP” та “forgot this file” (забув цей файл). Ви можете використати `reset` щоб швидко та легко зварити їх в один коміт, що дозволяє вам

виглядати дійсно витонченим. (У **“Зварювання комітів”** йдеться про інший спосіб це зробити, проте в даному випадку легше скористатися `reset`.)

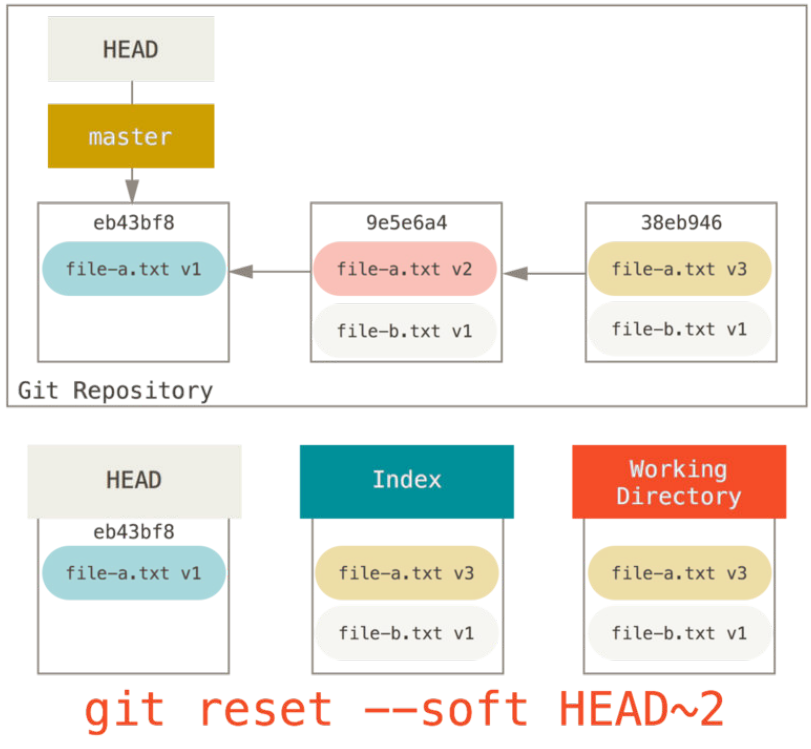
Скажімо у вас є проект, в якому перший коміт містить один файл, другий додав новий файл та змінив перший, а третій знову змінив перший файл. Другий коміт був незавершеною працею та ви бажаєте його зварити з останнім.

ЗОБРАЖЕННЯ 7-16



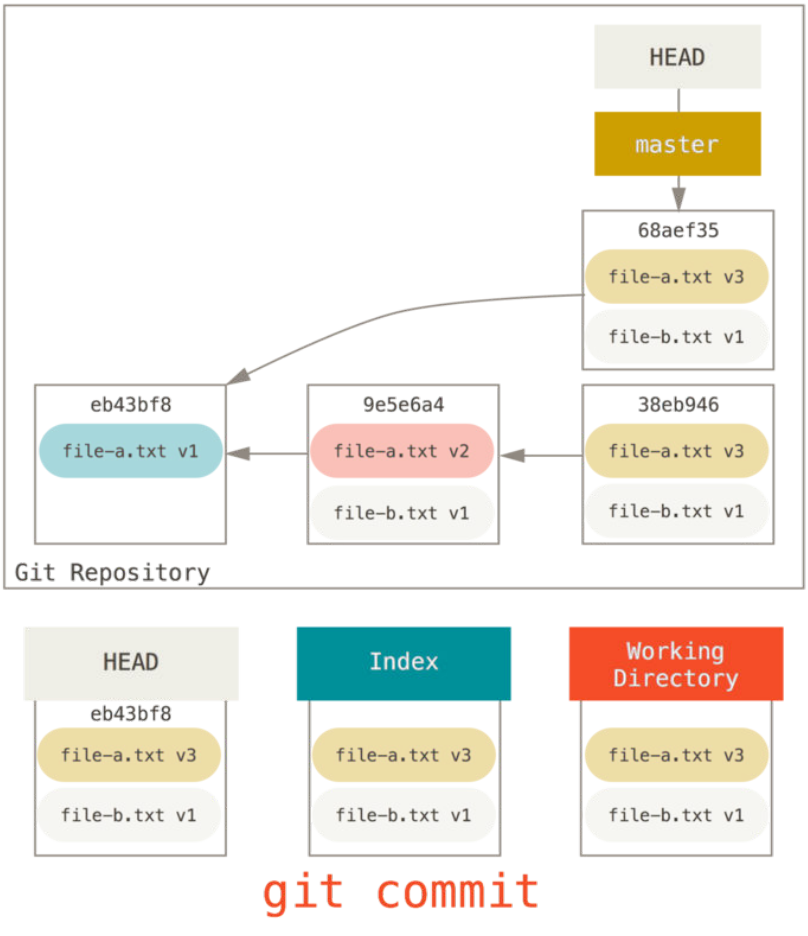
Ви можете виконати `git reset --soft HEAD~2` щоб перемістити гілку HEAD назад до старшого коміту (першого коміту, який ви бажаєте залишити):

ЗОБРАЖЕННЯ
7-17



Та просто знову виконати `git commit`:

ЗОБРАЖЕННЯ 7-18



Тепер, як бачите, ваша досяжна історія, історія яку ви будете викладати, тепер виглядає ніби ви зробили були лише коміт з `file-a.txt` першої версії, потім другий коміт та коміт, що змінив `file-a.txt` до третьої версії та додав `file-b.txt`. Коміт з другою версією файла більше не існує в історії.

Отримання (checkout)

Нарешті, вам може бути цікаво в чому різниця між `checkout` та `reset`. Як і `reset`, `checkout` маніпулює трьома деревами, проте трохи по-іншому в залежності від того, даєте ви їй окремі файли чи ні.

БЕЗ ОКРЕМИХ ФАЙЛІВ

Виконання `git checkout [гілка]` дуже схоже на виконання `git reset --hard [гілка]` в тому, що оновлює всі три дерева до `[гілки]`, проте є дві важливих відмінності.

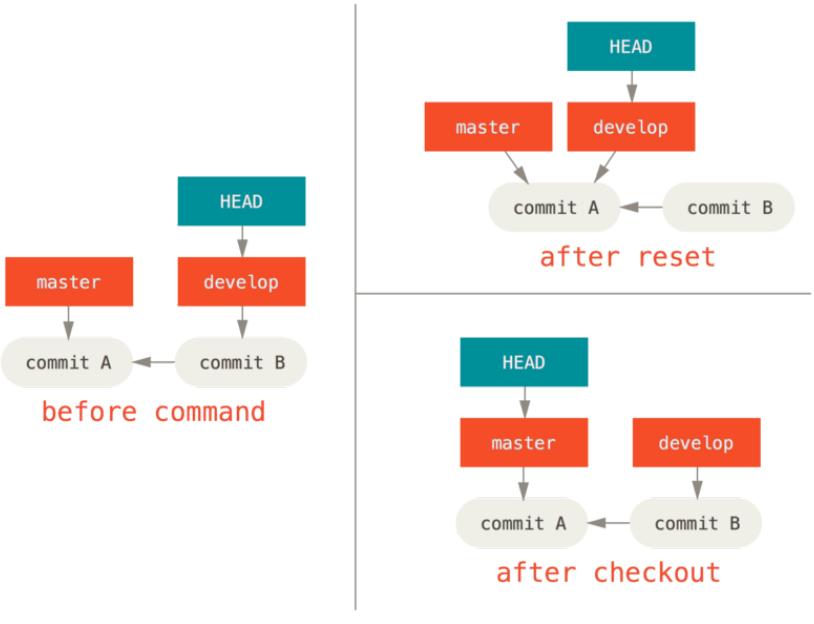
По-перше, на відміну від `reset --hard`, `checkout` безпечна команда щодо робочої директорії. Вона спершу перевіряє, що не зіпсує ніяких файлів, в яких є зміни. Насправді, вона навіть трохи кмітливіша – вона намагається зробити просте злиття в Робочій Директорії, отже всі файли, які ви *не* змінювали будуть оновлені. `reset --hard`, з іншого боку, просто замінить все не розбираючи та не перевіряючи.

Друга важлива відмінність у тому, як виконується оновлення HEAD. `reset` переміщує гілку, на яку вказує HEAD, а `checkout` натомість переміщує сам HEAD, щоб той вказував на іншу гілку.

Наприклад, скажімо в нас є гілки `master` та `develop`, які вказують на різні коміти, та поточною гілкою є `develop` (отже HEAD на неї вказує). Якщо виконати `git reset master`, то сам `develop` почне вказувати на той самий коміт, що й `master`. Якщо ж виконати `git checkout master`, то пересувається не `develop`, а HEAD. HEAD почне вказувати на `master`.

Отже, в обох випадках ми переміщуємо HEAD до коміту A, проте *як* ми це робимо дуже різниться. `reset` переміщує гілку, на яку вказує HEAD, а `checkout` переміщує сам HEAD.

ЗОБРАЖЕННЯ 7-19



З ФАЙЛАМИ

Інший спосіб виконати команду `checkout` — це надати йому шляхи файлів, що, як і з `reset`, призведе до збереження `HEAD`. Вона поводиться так само як `git reset [гілка] файл` в тому, що оновлює індекс файлом з того коміту, проте також переписує файл у робочій директорії. Вона була б повністю рівнозначна `git reset --hard [гілка] файл` (якби б `reset` це дозволяв) – вона небезпечна для робочою директорії, та не переміщує `HEAD`.

Також, як і `git reset` та `git add`, `checkout` розуміє опцію `--patch`, що дозволяє вибірково повертати вміст файла по шматкам.

Підсумок

Сподіваємось, що тепер ви розумієте і почуваєтесь комфортніше з командою `reset`, проте ймовірно досі трохи спантеличені, чим саме вона відрізняється від `checkout` та навряд чи запам'ятали всі правила різноманітних форм викликів.

Ось шпаргалка щодо того, як команди впливають на які дерева. Колонка "`HEAD`" має значення "ПОС" (посилання), якщо переміщує

посилання (гілку), на яке вказує HEAD, або "HEAD", якщо переміщує сам HEAD. Приділить особливу увагу колонці *РД у Безпеці?* (РД - робоча директорія) – якщо в ній **НІ**, двічі поміркуйте перш ніж виконати команду.

	HEAD	**Індекс**	**Робоча Директорія**	**РД у Безпеці?**
Рівень Комітів				
`reset --soft [коміт]`	ПОС	НІ	НІ	ТАК
`reset [коміт]`	ПОС	ТАК	НІ	ТАК
`reset --hard [коміт]`	ПОС	ТАК	ТАК	**НІ**
`checkout [коміт]`	HEAD	ТАК	ТАК	ТАК
Рівень Файлів				
`reset (коміт) [файл]`	НІ	ТАК	НІ	ТАК
`checkout (коміт) [файл]`	НІ	ТАК	ТАК	**НІ**

Складне злиття

Зливання в Git зазвичай проходить доволі легко. Оскільки Git дозволяє легко зливати іншу гілку декілька разів, ви можете працювати з дуже довготривалою гілкою, і в той же час оновлювати її в процесі роботи, та часто розв'язувати маленькі конфлікти замість того, щоб бути враженим величезним конфліктом наприкінці роботи з нею.

Утім, іноді трапляються й хитромудрі конфлікти. На відміну від інших систем контролю версій, Git не намагається бути надто розумним щодо розв'язання конфліктів. Філософія Git — бути розумним, коли злиття можна зробити однозначно, проте, якщо є конфлікт, Git не намагається бути розумним та автоматично його вирішити. Отже, якщо ви надто зволікаєте зі злиттям двох гілок, що розходяться швидко, у вас можуть виникнути проблеми.

У цій секції, ми розглянемо деякі з тих проблем, що можуть виникнути, та які утиліти Git допомагають впоратись з багатьма складними ситуаціями. Ми також розглянемо деякі інші, нестандартні типи зливань, які ви можете робити, а також як відмовитися від вже зробленого зливання.

Конфлікти злиття

Хоч ми й розглянули деякі основи розв'язання конфліктів зливання в "Основи конфліктів зливання", для складніших конфліктів Git пропонує декілька утиліт, що допоможуть вам збагнути що коїться та як краще мати справу з конфліктом.

Спершу, якщо це взагалі можливо, спробуйте зробити вашу робочу директорію чистою до того, як робити зливання, що призводять до конфліктів. Якщо у вас є незавершені зміни, або зробіть коміт до тимчасової гілки, або сховайте їх (stash). Таким чином ви зможете скасувати **будь-яку** з ваших спроб. Якщо у вас є незбережені зміни у робочій директорій, коли ви намагаєтесь зробити зливання, деякі з наступних інструкцій можуть сприяти втраті вашої праці.

Розгляньмо дуже простий приклад. У нас є надпростий файл Ruby, що друкує *hello world*.

```ruby
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

У нашому репозиторії, ми створюємо нову гілку `whitespace` та замінюємо всі Unix символи нового рядка на варіант DOS, тобто змінюємо кожен рядок файлу, проте виключно пробільні символи. Потім ми змінюємо рядок "hello world" на "hello mundo".

```
$ git checkout -b whitespace
Switched to a new branch '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 -b
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(-)
```

Тепер перейдемо назад до гілки та додамо якийсь опис цієї функції.

```
$ git checkout master
Switched to branch '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(+)
```

Тепер ми спробуємо злити до гілки `whitespace` та отримаємо конфлікти через зміни пробільних символів.

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

ПРИПИНЕННЯ ЗЛИТТЯ

Тепер у нас є декілька варіантів. По-перше, розповімо, як вийти з цього становища. Можливо, ви не очікували конфліктів, та не дуже бажаєте вирішувати ситуацію зараз, ви можете просто припинити злиття за допомогою `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

Опція `git merge --abort` намагається повернутися до стану, в якому ви були до початку зливання. Єдиний випадок, коли це може не вийти бездоганно — якщо у вас були несховані незбережені зміни в робочій директорії, коли ви почали злиття, інакше все пройде без проблем.

Якщо з будь-якої причини ви опинилися в жахливому стані, та просто бажаєте почати все з початку, ви можете виконати `git reset --hard HEAD` чи куди ви хочете повернутись. Не забувайте, що це знесе вашу робочу директорію, отже переконайтесь, що всі локальні зміни вам не потрібні.

ІГНОРУВАННЯ ПРОБІЛЬНИХ СИМВОЛІВ

У цьому окремому випадку, конфлікти пов'язані з пробільними символами. Ми знаємо про це, адже випадок простий, проте це доволі легко побачити в реальних ситуаціях, подивившись на конфлікт: кожен рядок видалено з одного боку та знову додано з іншого. Без додаткових опцій, Git розглядає всі рядки як змінені, отже не може злити файли.

Втім, типова стратегія зливання може приймати опції, а декілька з них про правильне ігнорування зміни пробільних символів. Якщо ви бачите, що у вас купа проблем з пробільними символами при злитті, ви можете просто припинити його та спробувати ще раз, цього разу з `-Xignore-all-space` або `-Xignore-space-change`. Перша опція **цілковито** ігнорує пробільні символи при порівнянні рядків, а друга

сприймає послідовності одного чи більше пробільних символів як рівнозначні.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Оскільки в даному випадку, справжні зміни файла не конфліктують, щойно ми проігноруємо зміни пробільних символів, злиття пройде чудово.

Це просто рятує життя, якщо хтось з вашої команди подеколи полюбляє замінити пробіли на таби чи навпаки.

ПОВТОРНЕ ЗЛИТТЯ ФАЙЛА ВРУЧНУ

Хоч Git і може впоратися з обробкою пробільних символів доволі якісно, існують інші типи змін, які Git певно не можне обробити автоматично, проте самі зміни зроблені автоматично. Наприклад, вдамо, що Git не зміг впоратися зі змінами пробілів, та ми маємо зробити це самі.

Що нам насправді треба зробити — це прогнати файл, який ми намагаємося злити, через програму `dos2unix` до того, як починати власне злиття файла. То як нам це зробити?

Спочатку нам треба потрапити до стану конфлікту злиття. Потім нам треба отримати копії нашої версії файла, їхньої версії (з гілки, яку ми зливаємо до нашої) та спільну версії (звідки обидві сторони розгалузились). Далі ми бажаємо виправити або їхній варіант або наш та спробувати злити знову лише цей один файл.

Отримати три версії файла насправді доволі просто. Git зберігає кожну з цих версій в індексі під “станами” (stages), які мають пов’язані з ними номери. Стан 1 — це спільний предок, стан 2 — це ваша версія, а стан 3 — з `MERGE_HEAD, версія, яку ви зливаєте до себе (`їхня'', theirs)

Ви можете витягнути копію кожної з цих версій конфліктного файлу за допомогою команди `git show` зі спеціальним синтактом.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Якщо ви бажаєте чогось суворішого, скористайтесь кухонною командою `ls-files -u`, щоб отримати власне SHA-1 суми Git блобів кожного з цих файлів

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1       hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2       hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3       hello.rb
```

`:1:hello.rb` — це просто скорочення для пошуку SHA-1 цього блобу.

Тепер у нас є зміст всіх трьох станів у нашій робочій директорії, ми можемо вручну виправити їхню версію — усунути проблему з пробільними символами, та ще раз злити файл за допомогою маловідомої команди `git merge-file`, яка робить саме це.

```
$ 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 -b
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()
```

Тепер ми гарно злили файл. Насправді, це працює навіть краще, ніж опція `ignore-space-change`, адже зміни пробільних символів виправлено до злиття, а не просто проігноровано. У злитті з `ignore-space-change`, ми в результаті отримали декілька рядків з DOS символами нового рядка, що призвело до змішання.

До того, як завершувати цей коміт, ви можете поглянути, що саме змінилось між однією чи іншою стороною. Ви можете попросити `git diff` порівняти те, що ви збираєтесь зберегти з вашої робочої директорії як результат злиття, з будь-яким з цих станів. Пройдімось по ним усім.

Щоб порівняти результат з тим, що було до зливання, іншими словами, щоб побачити, що саме з'явилось від злиття, можна виконати `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()
```

Отже тут ми легко можемо побачити, що саме сталося в нашій гілці, що ми насправді змінюємо у файлі цим злиттям: змінюємо один рядок.

Якщо ми бажаємо побачити, чим результат зливання відрізняється від того, що було на їхній стороні, можемо виконати `git diff --theirs`. У цьому й наступному прикладі, ми маємо використати `-b`, щоб прибрати пробільні символи, бо порівняння проходить зі збереженим у Git, а не з нашим очищеним файлом `hello.theirs.rb`.

```
$ git diff --theirs -b
* 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
```

Нарешті, ви можете побачити, як файл змінився з обох сторін за допомогою `git diff --base`.

```
$ git diff --base -b
* 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()
```

Тепер ми можемо використати команду `git clean`, щоб прибрати зайві файли, що ми їх створили для зливання вручну, бо вони більше не потрібні.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

ОТРИМАННЯ ПРИ КОНФЛІКТАХ

Можливо, ми наразі не задоволені розв'язанням конфлікту з якихось причин, або можливо редагування вручну однієї чи обох сторін досі не вийшло та нам треба більше контексту.

Змінімо трохи наш приклад. У цьому прикладі, у нас є дві більш довготривалі гілки, у кожній по декілька комітів, проте при зливанні створюють конфлікт саме за змістом.

```
$ 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
```

Тепер у нас три унікальних коміти, які є лише в гілці `master`, та три інших коміти, які є в гілці `mundo`. Якщо ми спробуємо злити `mundo`, то отримаємо конфлікт.

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

Ми хотіли б побачити, у чому власне конфлікт. Якщо відкрити файл, то побачимо щось таке:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Обидві сторони зливання додали зміст до цього файлу, проте деякі з комітів змінили файл в одному місці, що й спричинило цей конфлікт.

Дослідімо декілька утиліт у вашому розпорядженні, щоб визначити, як виник цей конфлікт. Можливо, як саме варто розв'язати конфлікт не є очевидним. Вам потрібно більше контексту.

Одна з корисних команд — `git checkout` з опцією `--conflict`. Ця команда ще раз отримає файл, та замінить позначки конфлікту (conflict markers). Це може бути корисним, якщо ви бажаєте відновити позначки та спробувати розв'язати їх знову.

Ви можете встановити `--conflict` значення `diff3`, або `merge` (що є типовим значенням). Якщо передати `diff3`, то Git використає трохи іншу версію позначок конфлікту: не тільки надасть вам "вашу" та

“їхню” версії, а ще й “базову” версію до файлу, щоб надати вам більше контексту.

```
$ git checkout --conflict=diff3 hello.rb
```

Після виконання, файл стане виглядати так:

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

Якщо вам подобається такий формат, ви можете зробити його типовим для майбутніх конфліктів злиття, якщо встановите налаштування `merge.conflictstyle` у значення `diff3`.

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

Команда `git checkout` також приймає опції `--ours` та `--theirs`, які є дійсно швидким засобом вибору лише однієї сторони, взагалі без зливання іншої.

Це особливо корисно для конфліктів двійкових файлів, адже ви можете просто вибрати одну сторону, або коли ви хочете злити окремі файли з іншої гілки - ви можете злити, а потім отримати (checkout) окремі файли з однієї сторони до створення коміту.

ЖУРНАЛ ЗЛИВАННЯ

Ще одна корисна утиліта при розв’язанні конфліктів злиття — `git log`. Вона може допомогти вам отримати інформацію про те, що могло сприяти конфлікту. Переглянути трохи історії, щоб запам’ятати, чому два рядки розробки зачепили одну область коду, може бути дуже корисним подеколи.

Щоб отримати повний список усіх унікальних комітів, що є у будь-якій з конфліктуючих гілок, можна використати синтаксис “потрійної крапки”, про який ми дізналися в “Потрійна крапка”.

```
$ 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
```

Це гарний список усіх шести комітів, а також з якої лінії розробки надійшов кожен коміт.

Ми можемо ще спростити список так, щоб це надало нам більш точний контекст. Якщо додати опцію `--merge` до `git log`, вона покаже лише ті коміти з обох сторін злиття, які зачепили файл, в якому наразі є конфлікти.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

Якщо виконати це з опцією `-p`, отримаємо лише зміни у файлах, які опинились у конфлікті. Це може бути **дійсно** корисним у швидкому надаванні контексту, що допоможе зрозуміти причини конфлікту та як його розумно розв’язати.

ПОЄДНАНИЙ ФОРМАТ РІЗНИЦІ (DIFF)

Оскільки Git індексує будь-які успішні результати злиття, коли ви виконуєте `git diff` у стані конфлікту злиття, ви отримуєте лише конфліктуючі зміни. Це може бути корисним, щоб побачити, що вам ще треба розв’язати.

Якщо виконати `git diff` одразу після конфлікту злиття, то вона видасть інформацію в доволі унікальному форматі різниці.

```
$ 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()
```

Формат називається “Поєднана різниця” (combined diff) та видає дві колонки даних навпроти кожного рядка. Перша колонка показує вам, якщо рядок змінився (доданий чи вилучений) між “вашою” гілкою та файлом у робочій директорії, а друга колонка робить те саме між “їхньою” гілкою та робочою директорією.

Отже в цьому прикладі можна побачити, що рядки `<<<<<<<` та `>>>>>>>` є в робочій копії, проте їх нема в жодній зі сторін зливання. Це має сенс, адже утиліта злиття додала їх для нас, але від нас очікується їх вилучення.

Якщо розв’язати конфлікт та виконати `git diff` знову, ми побачимо схожий результат, проте трохи корисніший.

```
$ 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()
```

Це показує нам, що “hola world” був на нашій стороні, проте не в робочій копії; “hello mundo” був на їхній стороні, проте не в робочій копії; та нарешті, “hola mundo” не був у жодній зі сторін, проте є в

робочій копії. Це може бути корисним, щоб переглянути розв'язання конфлікту до створення коміту.

Ви також можете отримати таке від `git log` для будь-якого вже зробленого зливання , щоб побачити як щось було розв'язано. Git зробить видрук у цьому форматі, якщо виконати `git show` для коміту злиття, або якщо додати опцію `--cc` до `git log -p` (яка типово показує лише латки для комітів не злиття).

```
$ 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()
```

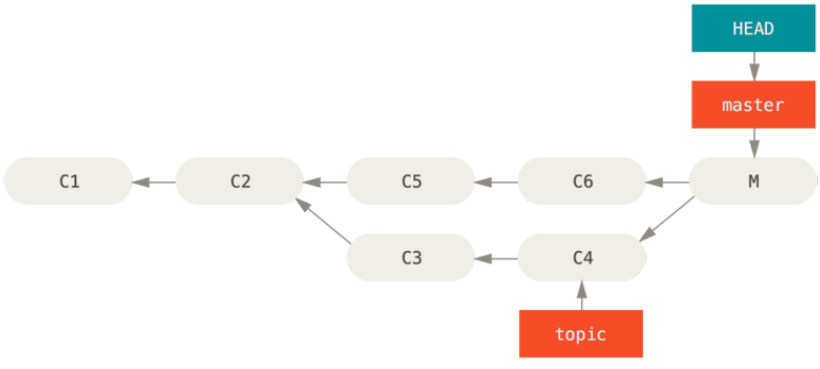
Скасування зливань

Тепер, коли ви знаєте, як створити коміт злиття, ви напевно зробите якусь помилку. Одна з чудових речей роботи з Git — що це нормально робити помилки, адже їх можливо (та зазвичай легко) виправити.

Коміти зливання тут не відрізняються. Скажімо, ви почали працювати над тематичною гілкою, та випадково злили її до `master`, і тепер історія комітів виглядає так:

ЗОБРАЖЕННЯ 7-20

Випадковий коміт злиття



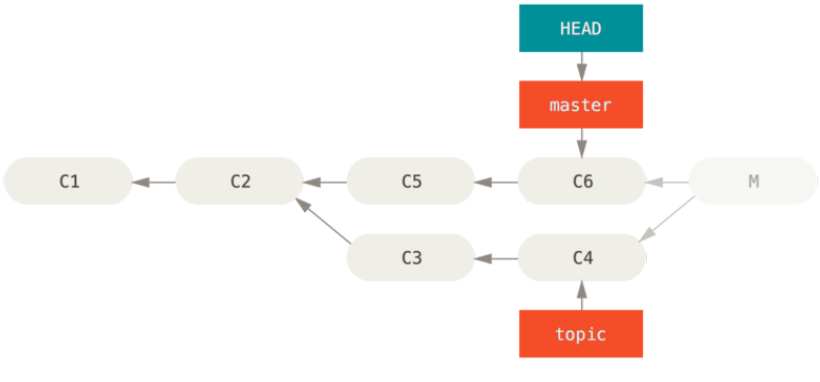
Є два підходи до цієї проблеми, у залежності від бажаного результату.

ВИПРАВИТИ ПОСИЛАННЯ

Якщо небажаний коміт зливання існує лише у вашому локальному репозиторії, найлегше та найкраще рішення — пересунути гілки так, щоб вони вказували на що ви забажаєте. У більшості випадків, якщо одразу після помилки `git merge`, виконати `git reset --hard HEAD~`, то вказівники гілок будуть скинуті і будуть виглядати так:

ЗОБРАЖЕННЯ 7-21

Історія після `git reset --hard HEAD~`



Ми розглядали `reset` у **“Усвідомлення скидання (reset)”**, отже, вам має бути неважко зрозуміти, що тут сталося. Ось швидке повторення: `reset --hard` зазвичай робить наступні кроки:

1. Пересунути гілку, на яку вказує HEAD. У даному випадку, ми бажаємо пересунути `master` до того, де вона була до коміту злиття (C6).
2. Скопіювати HEAD до індекс.
3. Скопіювати індекс до робочої директорії.

Недолік цього підходу є те, що це переписування історії, що може стати проблемою в спільному репозиторії. Прогляньте **“Небезпеки перебазовування”** для докладного опису того, що може статися; коротка версія така: якщо інші люди мають коміти, які ви переписуєте, вам варто уникати `reset`. Цей підхід також не спрацює, якщо інші коміти були створені після злиття; переміщення посилань у результаті втратить ці зміни.

ВИВЕРТАННЯ КОМІТУ

Якщо переміщення вказівників гілок не спрацює для вас, Git дає ще один варіант: створити новий коміт, що скасовує всі зміни з існуючого коміту. Git називає цю операцію “вивертання” (revert), і в цьому конкретному випадку, треба викликати його так:

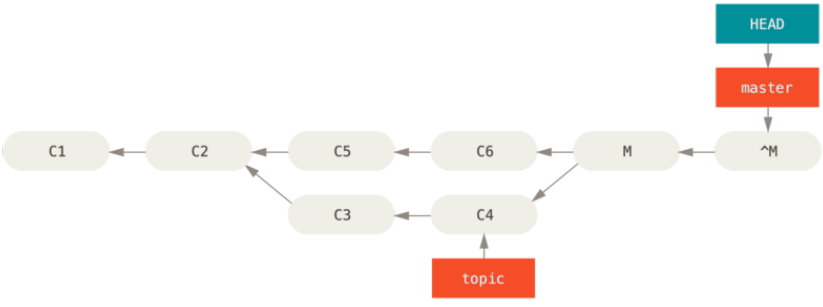
```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

Опція `-m 1` вказує, який батько є “стрижневим” (mainline) та має бути збереженим. Якщо виконати злиття до HEAD (`git merge topic`), новий коміт матиме двох батьків: перший це HEAD (C6), а другий — останній коміт гілки, яку ви зливаєте (C4). У даному випадку, ми бажаємо скасувати всі зміни, спричинені у батьку #2 (C4), проте зберігти весь зміст з батька #1 (C6).

Історія з виверненим комітом виглядає так:

ЗОБРАЖЕННЯ 7-22

Історія після `git revert -m 1`



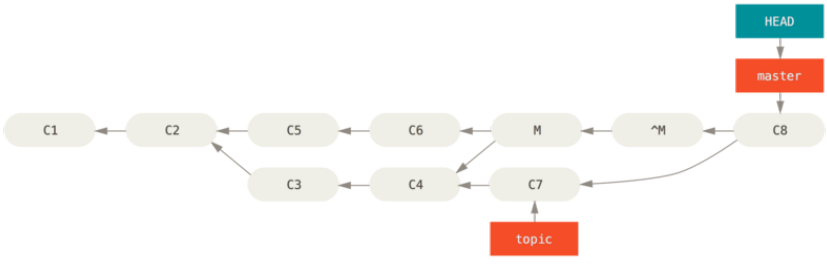
Новий коміт `^M` має точно такий зміст, як `C6`, отже починаючи звідти, усе нібито злиття ніколи не було, окрім того, що тепер не злиті коміти досі присутні в історії `HEAD`. Git заплутається, якщо ви спробуєте злити `topic` до `master` знову:

```
$ git merge topic
Already up-to-date.
```

У гілці `topic` немає нічого недосяжного з гілки `master`. Гірше того, якщо ви додасте щось до `topic`, та зіллєте знову, Git візьме лише зміни *після* виверту злиття:

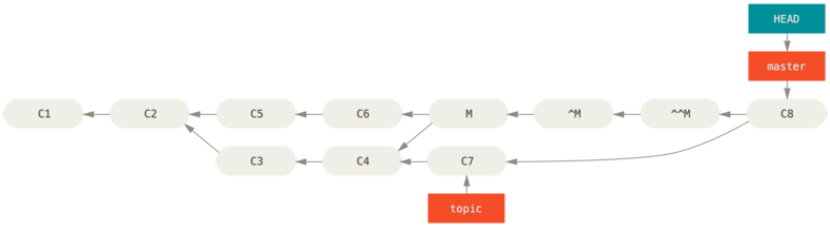
ЗОБРАЖЕННЯ 7-23

Історія з поганим злиттям



Найкращій вихід — це вивернути виворіт першого злиття, адже тепер ви бажаєте повернути зміни, які ви були вивернули, **потім** створити новий коміт злиття:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```



ЗОБРАЖЕННЯ 7-24

Історія після повторного злиття вивернутого злиття

У цьому прикладі, `M` та `^M` взаємно знищились. `^^M` в результаті зливає зміни `C3` та `C4`, а `C8` зливає зміну `C7`, отже тепер `topic` повністю злито.

Інші типи зливань

Досі ми розглядали нормальні злиття двох гілок, що зазвичай обробляються тим, що називається стратегією злиття “recursive”. Одначе, існують й інші методи зливати гілки разом. Швидко розгляньмо декілька з них.

ПЕРЕВАГА НАШОГО ЧИ ЇХНЬОГО

Почнімо з додаткових можливостей звичайного режиму зливання “recursive”. Ми вже зустрічали опції `ignore-all-space` та `ignore-space-change`, які передаються до `-X`, проте ми також можемо сказати Git надавати перевагу одній чи іншій стороні, коли він бачить конфлікт.

Без додаткових опцій, коли Git бачить конфлікт між двома гілками при зливанні, він додасть позначки конфлікту до вашого коду та позначить файл конфліктним, та дасть вам його розв’язати. Якщо ви бажаєте, щоб Git просто вибрав якусь сторону та проігнорував іншу замість того, щоб залишати вам вручну розв’язувати конфлікт, ви можете передати команді `merge` опцію `-Xours` чи `-Xtheirs`.

Якщо це зробити, Git не буде додавати позначки конфлікту. Будь-які зміни, що можна злити, будуть злиті. Для будь-яких конфліктуючих

змін, Git просто вибере сторону, яку ви задали, включно з двійковими файлами.

Якщо ми повернемося до попереднього прикладу “hello world”, ми можемо бачити, що зливання нашої гілки призводить до конфліктів.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Проте якщо виконати з `-Xours` або `-Xtheirs`, конфлікту не буде.

```
$ git merge -Xours mundo
Auto-merging 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
```

У даному випадку, замість отримання позначок конфлікту у файлі з “hello mundo” з одного боку та “hola world” з іншого, Git просто обере “hola world”. Однак, усі інші неконфліктуючі зміни на цій гілці успішно злиті.

Цю опцію також можна передати команді `git merge-file`, яку ми бачили раніше, наприклад виконавши `git merge-file --ours` для окремих зливань файлів.

Якщо вам потрібно щось схоже, проте щоб Git навіть не намагався зливати зміни з іншої сторони, є більш драконівська опція, а саме *стратегія* злиття “ours”. Це не те саме, що *опція* рекурсивного (recursive) злиття “ours”.

Це фактично зробить підроблене злиття. Буде записано новий коміт злиття з обома гілками в якості батьків, проте на гілку, яку ви зливаєте, навіть не глянуть. У якості результату злиття буде записано саме код вашої поточної гілки.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Як бачите, між гілкою, на якій ви були, та результатом зливання немає ніякої різниці.

Це може бути корисним для введення Git в оману, бо він буде вважати, що гілка вже злита при подальших зливаннях. Наприклад, якщо ви відгалузили гілку “release”, та зробили якусь роботу в ній, яку ви потім забажаєте злити до гілки “master”. У той же час, якесь виправлення з “master” має бути додано до вашої гілки `release`. Ви можете злити виправлення до гілки `release` і також `merge -s ours` ту ж гілку до гілки `master` (хоча виправлення вже там), тоді при подальшому зливанні гілки `release`, ніяких конфліктів від цього виправлення не виникне.

ЗЛИВАННЯ ПІДДЕРЕВ

Ідея зливання піддерева в тому, що у вас два проекти, один з яких відповідає піддиректорії іншого та навпаки. Коли ви використовуєте зливання піддерев, Git зазвичай достатньо розумний, щоб визначити, що одне є піддеревом іншого та злити відповідно.

Ми розглянемо приклад додавання окремого проекту до вже існуючого, та зливання коду з другого до піддиректорії першого.

Спочатку, ми додамо застосунок Rack до вашого проекту. Ми додамо проект Rack у якості віддаленого посилання до нашого проекту, а потім отримаємо його в окрему гілку:

```
$ 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"
```

Тепер у нас є корінь проекту Rack у гілці `rack_branch` та наш власний проект у гілці `master`. Якщо переключитись на одну, а потім на іншу, можна побачити, що в них різні корені:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Це дещо дивна концепція. Не всі гілки вашого репозиторію насправді мають бути гілками одного проекту. Це не поширено, адже зрідка корисно, проте мати гілки, що містять цілковито різні історії, доволі легко.

У даному випадку, ми хочемо втягнути проект Rack до нашого проекту `master` в якості піддиректорії. Ми можемо зробити це в Git за допомогою `git read-tree`. Ви дізнаєтесь більше про `read-tree`, та йому подібних, у **Chapter 10**, проте, покищо знайте, що вона зчитує корінь дерева однієї гілки до вашого поточного індексу та робочої директорії. Ми щойно переключились назад до вашої гілки `master`, та втягуємо гілку `rack_branch` до піддиректорії нашої гілки `master` нашого головного проекту:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Коли ми збережемо коміт, здається, ніби всі файли Rack є в тій піддиректорії – нібито ми скопіювали їх з архіву. Цікаво те, що ми легко можемо зливати зміни з однієї гілки в іншу. Отже, якщо проект Rack буде оновлено, ми зможемо отримати останні зміни, коли переключимося на ту гілку та виконаємо `pull`:

```
$ git checkout rack_branch
$ git pull
```

Потім, ми можемо злити ці зміни назад до нашої гілки `master`. Щоб взяти зміни та отримати заповненим повідомлення коміту, використайте опцію `--squash`, а також опцію `-Xsubstree` стратегії

зливання `recursive`. (Рекурсивна стратегія зливання є типовою, проте ми включили її для ясності.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Усі зміни з проекту Rack злиті та готові до збереження в локальному коміті. Ви можете робити й навпаки – зберігати зміни у піддиректорії `rack` вашої гілки `master` та потім зливати їх до гілки `rack_branch`, щоб пізніше відправити їх до супроводжувачів або надсилання до джерела проекту.

Це надає спосіб мати процес роботи схожий на процес роботи з підмодулями без використання підмодулів (які ми розглянемо в **“Підмодулі”**). Ми можемо зберігати гілки інших пов’язаних проектів у нашому репозиторії та іноді робити зливання піддерев з ними. Це мило з якогось боку, наприклад, весь код збережено в одному місці. Проте, є інші недоліки: це трохи складніше, та легше наробити помилок при реінтеграції змін або випадково надіслати гілку до геть не того репозиторію.

Ще одна трохи дивна річ: щоб отримати різницю між вашою піддиректорією `rack` та кодом у вашій гілці `rack_branch` – щоб побачити, чи треба їх зливати – ви не можете використовувати звичайну команду `diff`. Замість цього, ви маєте виконати `git diff-tree` з гілкою, яку ви бажаєте порівняти зі своєю:

```
$ git diff-tree -p rack_branch
```

Або, щоб порівняти вашу піддиректорію `rack` зі змістом гілки `master`, що був на сервері, коли ви востаннє отримували звідти зміни, ви можете виконати:

```
$ git diff-tree -p rack_remote/master
```

Rerere

Можливості `git rerere` є трохи прихованими. Назва походить від “використовуй записані розв’язання” (reuse recorded resolution) та, як

зрозуміло з назви, дозволяє вам попросити Git запам'ятати, як ви розв'язали конфлікт шматків (hunk), щоб наступного разу такий самий конфлікт Git міг автоматично розв'язати для вас.

Є декілька ситуацій, в яких цей функціонал може бути дуже доречним. Один з прикладів, що згадується в документації: якщо ви бажаєте переконатись, що довготривала тематична гілка зіллється чисто, проте не бажаєте мати купу проміжних комітів злиття. Якщо `rerere` увімкнено, ви можете іноді зливати, розв'язувати конфлікти, а потім припиняти злиття. Якщо робити це постійно, то останнє злиття має бути простим, адже `rerere` може просто зробити за вас все автоматично.

Таку саму тактику можна використовувати, якщо ви бажаєте тримати гілку перебазованою, щоб не доводилося мати справу з однаковими конфліктами перебазування щоразу, як ви його робите. Або якщо ви бажаєте взяти гілку, яку ви зливали та виправили купу конфліктів, а потім вирішили замість цього зробити перебазування - імовірно вам не доведеться знову виправляти всі ці конфлікти.

Ще одна ситуація: коли ви іноді зливаєте купу незавершених тематичних гілок разом, щоб перевірити результат, як сам проект Git часто робить. Якщо тести провалились, ви можете скасувати зливання та зробити їх знову без тематичної гілки, що спричинила проблему, без необхідності знову розв'язувати конфлікти.

Щоб увімкнути функціональність `rerere`, треба просто виконати це налаштування конфігурації:

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

Також його можна ввімкнути, якщо створити директорію `.git/rr-cache` в окремому репозиторії, проте налаштування конфігурації ясніше, та може бути зроблено глобально.

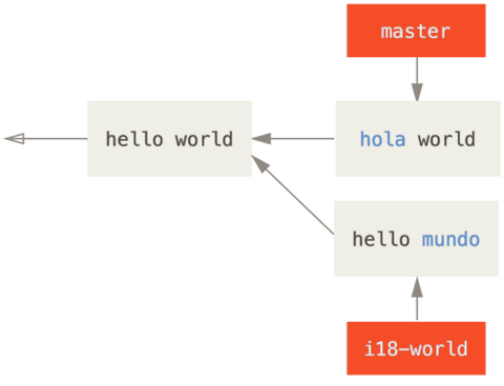
Тепер погляньмо на простий приклад, схожий на попередній. Скажімо, у нас є файл, що виглядає так:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

В одній гілці ми змінюємо слово "hello" на "holf", потім в іншій гілці змінюємо "world" на "mundo", як і раніше.

ЗОБРАЖЕННЯ 7-25



Коли ми зіллємо ці дві гілки разом, ми отримаємо конфлікт злиття:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

Зверніть тут увагу на новий рядок `Recorded preimage for FILE` (записано предвідбиток для ФАЙЛУ). Решта виглядає так само, як і звичайний конфлікт зливання. Наразі, `rerere` знає декілька речей. Зазвичай, ви зараз виконали б `git status`, щоб побачити всі конфлікти:

```
$ 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
#
```

Однак, `git rerere` також скаже вам, що він записав стан до злиття, якщо виконати `git rerere status`:

```
$ git rerere status
hello.rb
```

А `git rerere diff` покаже поточний стан розв'язання - з чого почалось розв'язання та як ви його розв'язали.

```
$ 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
```

Також (і це насправді не пов'язано з `rerere`), ви можете використати `ls-files -u`, щоб побачити файли конфлікту та попередню, ліву та праву версії:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1       hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2       hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3       hello.rb
```

Тепер ви можете розв'язати конфлікт щоб було `puts 'hola mundo'` та знову виконати команду `rerere diff`, щоб побачити, що запам'ятає rerere:

```
$ 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
```

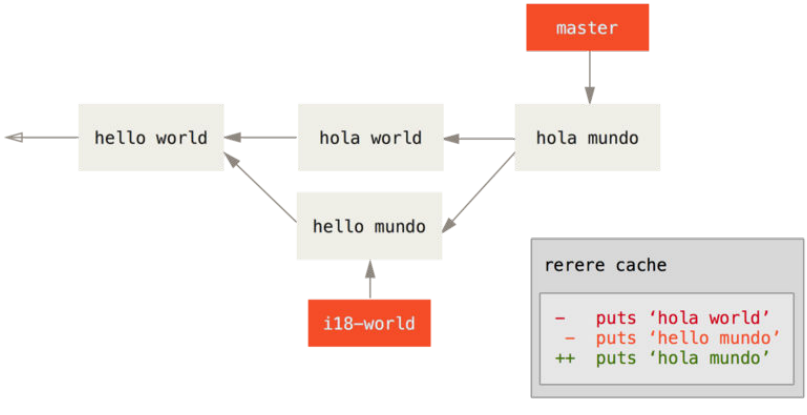
Це нам каже, що коли Git побачить конфлікт шмату (hunk) у файлі `hello.rb`, де з одного боку “hello mundo”, а з іншого “hola world”, він розв’яже конфлікт рядком “hola mundo”.

Тепер ми можемо позначити його розв’язаним та зберегти у коміті:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Зверніть увагу на “Recorded resolution for FILE” (Записано розв’язок для ФАЙЛУ).

ЗОБРАЖЕННЯ 7-26



Тепер, скасуймо це злиття та замість нього перебазуймо нашу гілку поверху master. Ми можемо пересунути нашу гілку назад за допомогою `reset`, як ми бачили в **“Усвідомлення скидання (reset)”**.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Наше злиття скасовано. Тепер перебазуймо тематичну гілку.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Тепер, ми отримали саме такий конфлікт зливання, як очікували, проте погляньте на рядок `Resolved FILE using previous resolution` (Розв'язали ФАЙЛ за допомогою попереднього розв'язку). Якщо ми відкриємо файл, то побачимо, що він вже розв'язаний, там більше немає позначок конфлікту.

```
$ cat hello.rb
#! /usr/bin/env ruby

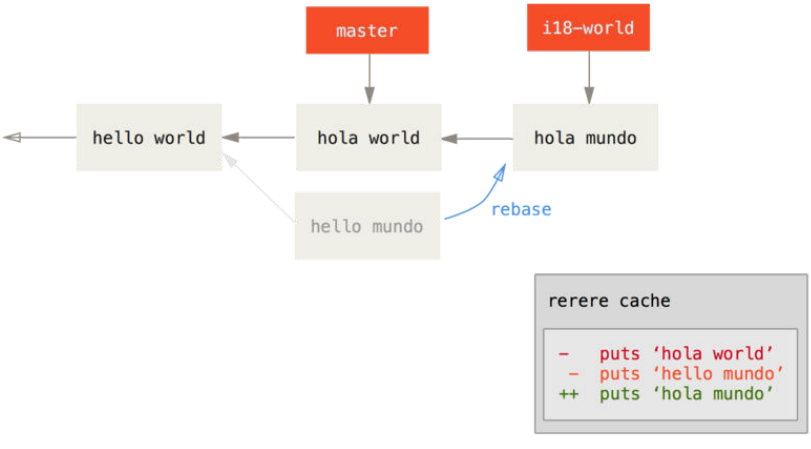
def hello
  puts 'hola mundo'
end
```

Також, `git diff` покаже вам, як конфлікт був знову розв'язаний автоматично:

```
$ 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
```

ЗОБРАЖЕННЯ 7-27



Ви також можете повернути файл до стану конфлікту за допомогою команди 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
```

Ми бачили приклад цього в “Складне злиття”. Проте зараз, розв’яжімо його знову: для цього треба просто виконати `rerere` знов:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Ми знову розв'язали файл автоматично за допомогою збереженого `rerere` розв'язок. Тепер ви можете проіндексувати файл та продовжити перебазування, щоб завершити його.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Отже, якщо ви робите багато повторних зливань, або бажаєте зберігати тематичну гілку у відповідності з гілкою master без численних зливань, або часто перебазовуєте, то можете увімкнути `rerere`, щоб трохи полегшити собі життя.

Зневадження з Git

Git також пропонує декілька утиліт, що допомагають зневаджувати проблеми у вашому проекті. Через те, що Git розроблено для роботи майже з будь-яким видом проектів, ці інструменти доволі загальні, проте часто можуть допомогти вам відстежити ваду (bug) чи винуватця, коли щось пішло не так.

Анотація файла

Якщо ви шукаєте ваду у своєму коді та бажаєте знати, коли вона з'явилася та чому, анотація файлів часто є найкращим інструментом. Він показує вам, який коміт востаннє редагував кожен рядок будь-якого файла. Отже, якщо ви бачите, що метод у вашому коді має помилку, ви можете анотувати файл за допомогою `git blame`, щоб побачити, коли кожен рядок методу востаннє редагувався та ким. Цей приклад використовує опцію `-L` щоб обмежити вивід до рядків з 12 до 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
```

Зауважте, що перше поле це часткова SHA-1 коміту, який востаннє змінював цей рядок. Наступні два поля є отриманими з цього коміту значеннями: ім'я автора та дата створення цього коміту — отже ви легко можете побачити, хто та коли редагував рядки. Після цього йде номер рядка та зміст файла. Також зверніть увагу на рядки комітів `^4832fe2`: вони означають, що ці рядки були в першому коміті файла. Цей коміт був першим, коли файл було додано до проекту, та ці рядки відтоді не змінювались. Це крапельку дивно, адже тепер ви бачили принаймні три різних значення, в яких Git використовує ^ щоб змінити SHA-1 коміту, проте саме це в даному випадку цей символ і означає.

Інша чудова річ у Git: він не слідкує за перейменуваннями файлів явно. Він записує знімки та потім намагається збагнути, що було перейменовано сам, вже після перейменування. Один з цікавих наслідків цього полягає в тому, що ви також можете попросити Git знайти всілякі переміщення коду. Якщо передати `-C` до `git blame`, Git проаналізує файл, що ви його анотуєте, та спробує зрозуміти, звідки з'явилися частини коду, якщо вони були скопійовані з іншого місця. Наприклад, припустімо ви переробляєте файл `GITServerHandler.m` на декілька файлів, один з яких має назву `GITPackUpload.m`. Якщо виконати `blame` на файлі `GITPackUpload.m` з опцією `-C`, ви можете побачити, звідки з'явились секції коду:

```
$ 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)
```

Це дійсно корисно. Зазвичай, ви отримуєте першим комітом той коміт, з якого ви скопіювали код, адже це перший раз, коли ви змінювали ці рядки у цьому файлі. Git повідомляє вам перший коміт, в якому ви написали ці рядки, навіть якщо це сталося в іншому файлі.

Двійковий пошук

Щоб анотування файла допомогло, треба щоб ви спочатку знали, де проблема. Якщо ви не знаєте, що зламалося, і було зроблено десятки або сотні комітів відтоді як ви точно знали, що код працював, ви напевно звернетесь до `bisect` по допомогу. Команда `bisect` виконує двійковий пошук у вашій історії комітів, щоб допомогти вам визначити якомога швидше, який коміт спричинив проблему.

Скажімо, ви щойно виклали ваш готовий код до виробничого середовища, та отримуєте звіти вад про щось, чого не було у вашому середовищі розробки, і ви гадки не маєте, чому код так себе поводить. Ви повертаєтесь до свого коду, і виявляється, що ви можете відтворити проблему, проте не можете зрозуміти, що працює не так. Ви розділяєте свій код навпіл (bisect), щоб з'ясувати це. Спочатку виконуємо `git bisect start`, щоб розпочати процес, потім скористаємося `git bisect bad`, щоб повідомити системі, що поточний коміт зламаний. Потім, ви маєте повідомити `bisect`, коли востаннє був відомий гарний стан за допомогою `git bisect good [good_commit]`:

```
$ 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 з'ясував, що було зроблено приблизно 12 комітів між позначиним як останній гарний коміт (v1.0) та поточною поганою версією, та отримав (checked out) середній для вас. Наразі, ви можете виконати ваші тести, щоб дізнатись, чи існує проблема в цьому коміті. Якщо існує, то вона виникла десь до цього середнього коміту; якщо ж

ні, то проблема виникла після середнього коміту. Виявляється, що тут проблеми немає, і ми кажемо про це Git за допомогою `git bisect good`, та продовжуємо свою подорож:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Тепер ви на іншому коміті, посередині між щойно протестованим та вашим поганим комітом. Ви виконуєте тести знов, та з'ясовуєте, що цього разу коміт зламаний, отже ви кажете про це Git за допомогою `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Цей коміт у порядку, і тепер Git має всю необхідну інформацію, щоб визначити, де виникла проблема. Він повідомляє вам SHA-1 першого поганого коміту, та показує деяку інформацію про коміт та які файли було змінено в цьому коміті, щоб ви могли дізнатися, що такого сталося, що могло спричинити цю ваду:

```
$ 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
```

Коли ви завершили, треба виконати `git bisect reset`, щоб повернути HEAD туди, де ви були спочатку, інакше ви опинитесь у дивному становищі:

```
$ git bisect reset
```

Це могутній інструмент, що може допомогти перевірити сотні комітів на наявність вади за хвилини. Насправді, якщо у вас є скрипт, що поверне 0, якщо проект у гарному стані, та не 0, якщо у поганому, то можливо повністю автоматизувати `git bisect`. Спершу, ви знову повідомляєте проміжок `bisect`: задаєте відомі гарний та поганий коміти. Це можна зробити, якщо додати їх до команди `bisect start`, якщо бажаєте: наведіть відомий поганий коміт першим, а відомий гарний коміт другим:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Це автоматично виконає `test-error.sh` на кожному отриманому коміті, доки Git не знайде перший зламаний коміт. Ви також можете виконати щось на кшталт `make` або `make test` чи що завгодно, що виконує автоматичні тести для вас.

Підмодулі

Часто, під час роботи з проектом трапляється, що в ньому потрібно використати інший проект. Можливо, це бібліотека, яку хтось інший або ви розробляєте окремо та використовуєте в декількох проектах. У цих ситуаціях виникає поширена проблема: ви бажаєте мати можливість розглядати два проекти як окремі, проте все одно мати можливість використовувати один з іншого.

Ось приклад. Уявіть, що ви розробляєте веб-сайт та створюєте Atom feeds. Замість того, щоб писати власний код для генерації Atom, ви вирішуєте використати бібліотеку. Ви, вірогідно, матимете або включити цей код як бібліотеку на кшталт інсталяції CPAN або Ruby gem, або скопіювати програмний код до власного дерева проекту. Проблема зі включенням бібліотеки в тому, що її буде важко налаштовувати як завгодно, та часто складно постачати її, адже вам доведеться переконатись, що кожен клієнт має цю бібліотеку. Проблема з включенням коду до вашого власного проекту полягає в тому, що будь-які допасовані зміни важко зливати, коли з'являються зміни оригінального коду.

Git намагається вирішити цю проблему за допомогою підмодулів. Підмодулі дозволяють зберігати репозиторій Git у піддиректорії іншого Git репозиторію. Це дозволяє вам зробити клон іншого репозиторію до проекту та тримати ваші коміти окремо.

Основи підмодулів

Ми розглянемо розробку простого проекту, який було розбито на головний проект та декілька підпроектів.

Почнімо з додавання існуючого репозиторію Git як підмодуля репозиторію, над яким ми працюємо. Щоб додати новий підмодуль, використайте команду `git submodule add` з URL проекту, за яким ви бажаєте почати слідкувати. У даному прикладі, ми додамо бібліотеку “DbConnector”.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into '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.
```

Без додаткових опцій, директорії, в які підмодулі додаються як підпроекти, називаються так само, як і репозиторій, у даному випадку “DbConnector”. Ви можете додати інший шлях наприкінці команди, якщо бажаєте, щоб підмодуль додався в інше місце.

Якщо виконати зараз `git status`, ви помітите декілька речей.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitmodules
        new file:   DbConnector
```

Перше, що впадає в око — новий файл `.gitmodules`. Це конфігураційний файл, що зберігає відображення між URL проекту та локальною піддиректорією, в яку його було отримано:

```
$ cat .gitmodules
[submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
```

Якщо у вас декілька підмодулів, то у файлі буде декілька записів. Важливо зауважити, що цей файл знаходиться під версійним контролем, як й інші файли, наприклад файл `.gitignore`. Він надсилається та отримується з рештою проекту. Таким чином інші люди, які клонують цей проект, знають звідки отримати проекти підмодулі.

Оскільки інші люди спершу будуть клонувати та отримувати зміни з URL у файлі .gitmodules, переконайтеся, що вони мають доступ до цього URL, якщо можете. Наприклад, якщо URL, до якого ви надсилаєте зміни, та URL, до якого інші мають доступ, різні, то використовуйте той, до якого інші мають доступ. Ви можете переписати це значення локально для власного використання за допомогою `git config submodule.DbConnector.url PRIVATE_URL`.

Інший елемент у видруці `git status` — директорія проекту. Якщо виконати для неї `git diff`, побачимо щось цікаве:

```
$ 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
```

Хоча `DbConnector` є піддиректорією у вашій робочій директорії, Git розглядає її як підмодуль, та не слідкує за її вмістом, доки ви не в ній. Натомість, Git розглядає її як окремий коміт з того репозиторію.

Якщо ви бажаєте трохи гарнішого вигляду різниці, то передайте опцію `--submodule` до `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)
```

Коли ви збережете коміт, ви побачите щось таке:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Зверніть увагу на права доступу `160000` для елементу `DbConnector`. Це спеціальні права доступу в Git, що означають, що ви записуєте коміт як директорію, а не просто піддиректорію або файл.

Клонування проекту з підмодулями

Тепер ми зробимо клон проекту з підмодулем у ньому. Коли ви клонуєте такий проект, то отримуєте директорії для підмодулів, проте жодна з них наразі не містить файлів:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ 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
$
```

Директорію `DbConnector` створено, проте вона порожня. Ви маєте виконати дві команди: `git submodule init` щоб проініціалізувати ваш файл локальною конфігурації, та `git submodule update`, щоб отримати всі дані з того проекту та перейти до відповідного коміту, який вказано у вашому головному проекті.

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for
$ git submodule update
Cloning into '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 'c3f01dc8862123d317dd46284b05b6892c7b29b
```

Тепер піддиректорія `DbConnector` саме в тому стані, в якому ви її зберегли в коміті раніше. Однак, існує інший, трохи простіший спосіб зробити це. Якщо передати `--recursive` до команди `git clone`, вона автоматично зробить ініціалізацію та оновить кожен підмодуль у репозиторії.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for
Cloning into '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 'c3f01dc8862123d317dd46284b05b6892c7b29b
```

Робота над проектом з підмодулями

Тепер у нас є копія проекту з підмодулями в ньому і ми будемо співпрацювати з іншими учасниками команди як над головним проектом, як і над проектом підмодулем.

ОТРИМАННЯ ЗМІН З ПЕРШОДЖЕРЕЛА

Найпростіша модель використання підмодулів у проекті — просто користуватись підпроектом та отримувати оновлення з нього подеколи, проте не змінювати нічого у своїй копії. Розгляньмо тут простий приклад.

Якщо ви бажаєте перевірити, чи є щось новеньке у підмодулі, то можете перейти до його директорії та виконати `git fetch` та `git merge` з гілкою джерела, щоб оновити локальний код.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Тепер, якщо ви перейдете назад до головного проекту та виконаєте `git diff --submodule`, то побачите, що підмодуль було оновлено, та отримаєте список комітів, які були до нього додано. Якщо ви не бажаєте набирати `--submodule` щоразу при виконанні `git diff`, то можете встановити це типовим форматом, якщо встановите змінну конфігурації `diff.submodule` у значення “log”.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Якщо зараз зробити коміт, ви змусите підмодуль отримувати новий код, коли інші оновляться.

Також існує простіший метод це зробити, якщо вам не хочеться вручну отримувати та зливати зміни до піддиректорії. Якщо виконати `git submodule update --remote`, Git перейде до ваших підмодулів та отримає й оновить за вас.

```
$ 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)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c064
```

Без додаткових налаштувань, ця команда вважає, що ви бажаєте оновити свою копію з гілки `master` репозиторію підмодуля. Однак, ви можете встановити якусь іншу, якщо бажаєте. Наприклад, якщо ви хочете, щоб підмодуль DbConnector слідкував за гілкою "stable" свого репозиторію, ви можете задати це або у файлі `.gitmodules` (щоб всі інші також слідкували за нею), або просто в локальному файлі `.git/config`. Установімо гілку у файлі `.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)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae668
```

Якщо пропустити `-f .gitmodules`, команда зробить зміни лише для вас, проте напевно більш розумно зберігати цю інформацію в репозиторії, щоб усі інші робили те саме.

Наразі при виконанні `git status`, Git покаже нам, що є нові коміти ("new commits") у підмодулі.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

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

Якщо ви встановите налаштування `status.submodulesummary`, Git також покаже вам короткий виклад змін у підмодулях:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitmodules
        modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

Тепер, якщо виконати `git diff`, ми побачимо як наші зміни у файлі `.gitmodules`, як і декілька комітів, які ми отримали та готові зберегти до нашого проекту підмодуля.

```
$ 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
```

Це дуже файно, адже ми можемо бачити журнал комітів, які збираємося зберегти в коміті в нашому підмодулі. Після збереження також можна побачити цю інформацію за допомогою `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
```

Без додаткових опцій Git спробує оновити **всі** ваші підмодулі при виконанні `git submodule update --remote`, отже якщо у вас їх багато, то можливо варто передати ім'я саме того підмодуля, який ви бажаєте оновити.

РОБОТА З ПІДМОДУЛЯМИ

Цілком імовірно, якщо ви використовуєте підмодулі, ви робите це, бо насправді збираєтесь працювати над кодом підмодуля одночасно з працею над кодом у головному проекті (чи одночасно над декількома підмодулями). Інакше, імовірно замість підмодулів ви б використали простішу систему керування залежностями (таку як Maven чи Rubygems).

Отже, тепер розгляньмо приклад того, як можна робити зміни в підмодулях у той же час, як і в головному проекті, та зберігати й публікувати ці зміни одночасно.

Досі, коли ми виконували команду `git submodule update` щоб отримати зміни з репозиторіїв підмодулів, Git брав зміни та оновлював файли у піддиректоріях, проте залишав підрепозиторії у так званому стані "відокремлений HEAD" (detached HEAD). Це означає, що немає локальної гілки (як "master", наприклад), яка слідкує за змінами. Отже будь-які зміни, що ви зробите, також не відслідковуються.

Щоб налаштувати підмодулі для легшої розробки, треба зробити дві речі. Вам треба перейти в кожен підмодуль та переключитись на гілку, над якою ви будете працювати. Потім треба сказати Git, що робити, якщо зроблено зміни, а `git submodule update --remote` отримує зміни з першоджерела. Є варіант зливати їх з вашою локальною роботою, або спробувати перебазувати вашу локальну роботу поверху нових змін.

Спершу, перейдімо до директорії нашого підмодуля та переключимось до гілки.

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

Спробуймо варіант "зливати". Щоб задати його вручну, можна просто додати опцію `--merge` до виклику `update`. Тут ми побачимо, що була зміна на сервері цього підмодуля, і її злито.

```
$ 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)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Якщо ми перейдемо до директорії `DbConnector`, то там вже злиті нові зміни до нашої локальної гілки `stable`. Тепер подивімося, що

станеться, коли ми зробимо власні локальні зміни до бібліотеки, а хтось інший надішле інші зміни до серверу в той же час.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Тепер, якщо ми оновимо наш підмодуль, то побачимо, що станеться, якщо ми зробили локальні зміни, а першоджерело також має зміни, які нам треба об'єднати.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da
```

Якщо ви забули передати `--rebase` чи `--merge`, Git просто оновить підмодуль до того, що б там не було на сервері, та переведе ваш проект до стану відокремленого HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da9
```

Якщо таке станеться, не хвилюйтесь, ви можете просто перейти назад до директорії та знову перейти до гілки (яка досі містить вашу роботу) та злити або перебазувати `origin/stable` (чи як називається віддалена гілку, яка вам потрібна) вручну.

Якщо ви не зберегли свої зміни в коміті у підмодулі, та викликали оновлення підмодуля, яке призвело до помилок, Git отримає зміни, проте не перепише незбережені зміни у директорії підмодуля.

```
$ 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)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
        scripts/setup.sh
```

```
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector
```

Якщо ви зробили зміни, що призвели до конфлікту з чимось з першоджерела, Git повідомить про це під час оновлення.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

Тепер ви можете перейти до директорії підмодуля та виправити конфлікт, як і зазвичай.

ПУБЛІКАЦІЯ ЗМІН З ПІДМОДУЛЯ

Тепер у нас є якісь зміни в директорії підмодуля. Деякі з них прийшли з першоджерела при оновленнях, а інші зроблені локально та не доступні покищо нікому, адже ми їх ще не надсилали.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

Якщо створити коміт у головному проекті та надіслати його, але не надіслати також зміни підмодуля, інші при спробі отримати зміни потраплять у халепу, оскільки у них не буде ніякої можливості отримати зміни підмодуля, від яких залежить проект. Ці зміни існують лише в нашій локальній копії.

Щоб переконатись, що такого не станеться, ви можете попросити Git перевірити, що всі підмодулі були відповідно надіслані перед надсиланням головного проекту. Команда `git push` приймає аргумент `--recurse-submodules`, який можна встановити або в “check” (перевірити) або в “on-demand” (за потребою). Опція “check” змусить

push просто зупинитися з помилкою, якщо будь-які збережені в підмодулі зміни не були надіслані.

```
$ 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.
```

Як бачите, нам також надано корисні поради щодо того, що ми можемо зробити далі. Простий варіант — перейти до кожного підмодуля та вручну надіслати до віддалених сховищ, щоб переконатись, що вони доступні ззовні, та потім знову спробувати надіслати.

Інший варіант — використати значення “on-demand”, що спробує зробити те саме за вас.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

Як тут можна побачити, Git перейшов до підмодуля DbConnector та надіслав його перед тим, як надсилати головний проект. Якщо

надіслати підмодуль чомусь не вдасться, надсилання головного проекту теж не вдасться.

ЗЛИВАННЯ ЗМІН У ПІДМОДУЛІ

Якщо ви змінили посилання підмодуля одночасно з кимось іншим, то можуть виникнути проблеми. Тобто, якщо історії підмодуля розійшлися та збережені в комітах на надпроекту, що розійшлися, виправити це може потребувати деяких зусиль.

Якщо один з комітів є прямим предком іншого (зливання перемотуванням), то Git просто вибере останній для зливання, отже це спрацює без проблем.

Однак, Git не спробує зробити навіть простого зливання. Якщо підмодульні коміти розійшлися та їх необхідно зливати, ви отримаєте щось таке:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
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)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Отже, по суті, сталося наступне: Git зрозумів, що дві записані точки гілок в історії підмодуля, що розійшлися, мають бути злиті. Це видно з “merge following commits not found” (злиття після комітів не знайдено), що важко зрозуміти, проте ми невдовзі все пояснимо.

Щоб вирішити проблему, вам треба зрозуміти, у якому стані мають бути підмодулі. Git дає на диво мало інформації, щоб вам допомогти: навіть не показує SHA-1 комітів з обох сторін історії. На щастя, зрозуміти це не складно. Якщо виконати `git diff`, то можна отримати SHA-1 комітів, записаних в обох гілках, які ви намагалися злити.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
```

```
--- a/DbConnector
+++ b/DbConnector
```

Отже, в даному випадку, `eb41d76` — це коміт у нашому підмодулі, який **ми** мали, а `c771610` — коміт, який був у першоджерелі. Якщо перейти до директорії підмодуля, вона вже має бути на `eb41d76`, оскільки зливання її не мало чіпати. Якщо з якоїсь причини це не так, ви можете просто створити та перейти до гілки, що вказує на нього.

Важливим є SHA-1 коміту з іншого боку. Він є тим, що вам треба злити та розв'язати. Ви можете або просто спробувати зробити зливання з SHA-1 безпосередньо, або створити гілку для нього, а потім спробувати її злити. Ми рекомендуємо останнє, навіть якщо це лише зробить повідомлення коміту гарнішим.

Отже, тепер ми перейдемо до директорії підмодуля, створимо гілку на базі другого SHA-1 з `git diff` та вручну зіллємо.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

Тепер ми отримали власне коміт, отже якщо його розв'язати та зберегти у коміті, потім можна просто оновити головний проект результатом.

```
$ vim src/main.c ❶
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ 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 "Merge Tom's Changes" ❺
[master 10d2c60] Merge Tom's Changes
```

❶ Спершу розв'язуємо конфлікт

❷ Потім повертаємось до директорії головного проекту

❸ Можемо знову перевірити SHA-1

❹ Розв'язуємо підмодуль у конфлікті

❺ Зберігаємо в коміті наше зливання

Це може бути трохи заплутано, проте насправді не таке вже й складне.

Цікаво, що є ще один випадок, який обробляє Git. Якщо існує коміт злиття у директорії підмодуля, що містить **обидва** коміти у своїй історії, Git запропонує його як можливе розв'язання. Він бачить, що колись у проекті підмодуля, хтось зливав гілки з цими двома комітами, отже можливо вам саме він і потрібен.

Ось чому повідомлення помилки раніше було "merge following commits not found", адже він не зміг **цього** зробити. Це збиває з пантелику, оскільки хто б очікував, що Git **намагається** це зробити?

Якщо він знайде єдиний прийнятний коміт злиття, ви побачите щось схоже на:

```
$ 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 "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
```

```
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Вам тут пропонується оновити індекс, нібито ви були виконали `git add`, що очистить конфлікт, а потім зробити коміт. Однак, вам, напевно, не варто цього робити. Ви можете так само легко перейти до директорії підмодуля, побачити, в чому різниця, перемотати до цього коміту, належним чином його перевірити, а потім зберігати це в коміті.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

Це досягає такого саме результату, проте, принаймні, таким чином можливо пересвідчитись, що код у вашому підмодулі працюватиме, коли ви закінчите.

Поради щодо підмодулів

Є декілька речей, що можуть дещо полегшити вашу роботу з підмодулями.

ДЛЯ КОЖНОГО ПІДМОДУЛЯ (SUBMODULE FOREACH)

Є команда `foreach` (для кожного), що дозволяє виконати довільну команду в кожному підмодулі. Це може бути дійсно корисним, якщо у вас багато підмодулів в одному проекті.

Наприклад, скажімо, ви бажаєте розпочати працювати над новим функціоналом чи виправленням, а у нас не закінчена робота над декількома підмодулями. Ми можемо легко сховати всю роботу в усіх підмодулях.

```
$ 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
```

Потім ми можемо створити нову гілку та перейти до неї в усіх підмодулях.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

Ви вхопили суть. Також ви можете отримати гарну об'єднану різницю того, що змінилося у головному проекті та у всіх підмодулях, що може буди дійсно корисним.

```
$ 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;
```

Як тут можна бачити, ми визначаємо функцію в підмодулі та викликаємо її з головного проекту. Це, очевидно, спрощений приклад, проте, сподіваємось, що він дає вам зрозуміти, наскільки це може бути корисним.

КОРИСНІ ПСЕВДОНІМИ

Можливо, вам захочеться налаштувати деякі псевдоніми, для деяких з цих команд, оскільки вони можуть бути доволі довгими, та ви не можете передати більшості з них опції, щоб зробити їх типовими. Ми розглянули налаштування псевдонімів у **“Псевдоніми Git”**, проте, ось приклад того, що ви можете забажати зробити, якщо плануєте багато працювати з підмодулями в 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'
```

Таким чином, можна просто виконати `git supdate`, коли вам треба оновити підмодулі, або `git spush`, щоб надіслати зміни з перевіркою залежних підмодулів.

Проблеми з підмодулями

Проте, використання підмодулів не є безхмарним.

Наприклад, переключення гілок з підмодулями в них також може бути хитромудрим. Якщо ви створите нову гілку, додасте туди підмодуль, та потім переключитесь назад до гілки без підмодуля, у вас досі буде присутня директорія підмодуля, як несупроводжувана директорія:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into '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
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

Видалити директорію не складно, проте те, що вона досі є, може збивати з пантелику. Якщо видалити її, та потім переключитися назад до гілки, яка містить підмодуль, доведеться виконати `submodule update --init`, щоб знову наповнити її.

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile        includes        scripts         src
```

Знову, не дуже складно, проте може бути трохи не очевидно.

З іншою поширеною проблемою багато людей стикаються при спробі перейти від піддиректорії до підмодуля. Якщо ви супроводжували файлі у своєму проекті та бажаєте винести їх до підмодуля, ви маєте бути обережними, щоб не роздратувати Git. Припустіть, у вас є файли в піддиректорії проекту, і ви бажаєте

перенести їх до підмодуля. Якщо ви видалите піддиректорію, а потім виконаєте `submodule add`, Git зчинить галас:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Спочатку ви мусите деіндексувати директорію `CryptoLibrary`. Потім можете додати підмодуль:

```
$ 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.
```

Тепер уявіть, що ви робили це в гілці. Якщо спробувати переключитись назад до гілки, де ці файли досі в справжньому дереві, а не в підмодулі – ви отримаєте таку помилку:

```
$ 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
```

Ви можете примусово переключитись за допомогою `checkout -f`, проте будьте обережні, якщо ви маєте незбережені зміни, оскільки вони можуть бути переписані цією командою.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Потім, при переключенні назад, ви отримуєте порожню директорію `CryptoLibrary` та з якогось дива `git submodule update` також не

допомагає. Вам можливо потрібно перейти в директорію підмодуля та виконати `git checkout .`, щоб отримати назад усі файли. Це можна виконати в скрипті `submodule foreach` для декількох підмодулів.

Важливо зазначити, що підмодулі нині зберігають усі свої дані в кореневій директорії проекту в директорії `.git`, отже, на відміну від старших версій Git, знищення директорії підмодуля не призводить до втрати ніяких комітів чи гілок, які у вас були.

За допомогою цих інструментів, підмодулі можуть бути доволі простим та ефективним методом розробки декількох пов'язаних, проте все ж таки окремих проектів одночасно.

Пакування

Хоч ми й розглянули звичайні способи передачі даних через мережу (HTTP, SSH тощо), насправді існує ще один, не такий поширений метод зробити це, що іноді може бути доволі корисним.

Git здатен до "пакування" (bundling) своїх даних в один файл. Це може бути корисним у різноманітних випадках. Можливо, ваша мережа не працює, а ви бажаєте відправити свої зміни співробітникам. Мабуть, ви працюєте десь поза офісом та не маєте доступу до локальної мережі через заходи безпеки. Можливо, ваша мережева картка просто зламалась. Можливо, наразі у вас немає доступу до спільного сервера, ви бажаєте надіслати електронного листа з оновленнями, проте не бажаєте відправляти 40 комітів через `format-patch`.

Ось тоді команда `git bundle` може допомогти. Команда `bundle` спакує все, що зазвичай передається через дріт командою `git push`, у двійковий файл, який ви можете передати поштою або записати на флеш-накопичувач, а потім розпакувати його в інший репозиторій.

Розгляньмо простий приклад. Скажімо, у вас є репозиторій з двома комітами:

```
$ 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
```

Якщо ви бажаєте відправити цей репозиторій комусь, проте не маєте туди доступу на запис, або просто не бажаєте налаштовувати доступ, ви можете спакувати його за допомогою `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Тепер у вас є файл `repo.bundle`, що містить всі необхідні для відтворення гілки `master` дані. Вам необхідно надати команді `bundle` кожне посилання або низку комітів, які ви бажаєте включити. Якщо ви збираєтесь таким чином створювати клон, то маєте також додати до посилань HEAD, як ми тут зробили.

Ви можете відправити файл `repo.bundle` електронною поштою комусь, або скопіювати його на USB накопичувач та віднести кудись.

З іншого боку, припустимо, що вам надіслали цей файл `repo.bundle`, та ви бажаєте попрацювати над проектом. Ви можете створити клон з цього двійкового файлу в директорію, ніби з 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
```

Якщо ви не включили HEAD до посилань, то маєте також додати опцію `-b master` чи будь-яку включену гілку, адже інакше Git не знатиме на яку гілку переключатись.

Тепер, припустімо, що ви створюєте три коміти у цій гілці та бажаєте відправити нові коміти назад за допомогою пакунку через електронну пошту або за допомогою USB накопичувача.

```
$ git log --oneline
71b84da last commit - second repo
```

```
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

Спочатку треба визначити, які саме коміти ми бажаємо включити в пакунок. На відміну від мережевих протоколів, які самі можуть знайти мінімальний набір даних для передачі через мережу, нам доведеться це зробити самотужки. Тепер, ви могли би просто зробити те саме та запакувати весь репозиторій, і це спрацює, проте краще спакувати лише різницю - лише три коміти, які ми щойно створили локально.

Щоб це зробити, вам треба обчислити різницю. Як було описано в **“Інтервали комітів”**, ви можете задати низку комітів безліччю методів. Щоб отримати три коміти, які є в гілці master, проте їх немає в гілці, з якої ми зробили клон, ми можемо використати щось подібне до `origin/master..master` або `master ^origin/master`. Ви можете перевірити це за допомогою команди `log`.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

Отже тепер, коли в нас є список комітів, які ми бажаємо включити в пакунок, спакуймо їх. Це робиться за допомогою команди `git bundle create`, їй треба передати назву файлу, в якому буде створено пакунок, та низку комітів, які мають до нього увійти.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

Тепер в нашій директорії є файл `commits.bundle`. Якщо відправити його нашій співробітниці, то вона зможе імпортувати його до оригінального репозиторію, навіть якщо там теж було щось зроблено за цей час.

Коли вона отримує пакунок, то може дослідити його зміст до того, як імпортувати його до репозиторію. Спершу варто скористатись

командою `bundle verify`, яка пересвідчиться, що файл дійсно є правильним пакунком Git, та у вас є всі необхідні для його правильного відновлення предки.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

Якби б пакувальник створив пакунок, що містить лише два останні коміти, а не всі три, то оригінальний репозиторій був би не в змозі імпортувати його, адже в ньому бракує необхідної історії. Команда `verify` виглядала би натомість так:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

Втім, наш перший пакунок правильний, отже ми можемо отримати з нього коміти. Якщо ви бажаєте дізнатись, які гілки є в пакунку та можуть бути імпортовані, існує також команда, що просто виводить список голів (heads):

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Підкоманда `verify` також розкаже вам про голови. Суть в тому, щоб побачити, що можна отримати, щоб ви могли використати команди `fetch` чи `pull`, щоб імпортувати коміти з пакунку. Тут ми отримаємо гілку `master` з пакунку в гілку нашого репозиторію під назвою `other-master`:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

Тепер ми бачимо, що імпортовані коміти є в гілці *other-master*, а також усі зроблені нами тим часом коміти в нашій власній гілці *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
```

Отже, `git bundle` може бути дійсно корисним, щоб надати комусь зміни або робити мережеві операції, коли у вас немає відповідної мережі або спільного репозиторію, щоб це робити.

Заміна

Об'єкти Git незмінні, проте Git надає цікавий засіб вдавати заміну об'єктів своєї бази на інші.

Команда `replace` дозволяє задати об'єкт у Git та сказати "щоразу як ти бачиш це, вдай нібито це інша річ". Це найбільш корисно для заміни одного коміту вашої історії на інший.

Наприклад, скажімо ви маєте неосяжну історію коду та бажаєте розділити репозиторій на один з короткою історією для нових розробників та один з набагато довшою та більшою історією для людей, що зацікавлені у видобуванні інформації. Ви можете прищепити одну історію до іншої, замінюючи (`replace`ing) найдавніший коміт у новій лінії останнім комітом старої. Це зручно, адже означає, що вам нема потреби насправді переписувати кожен коміт нової історії, як вам зазвичай доводиться робити, щоб поєднати їх разом (адже батьківство впливає на SHA-1 суми).

Спробуймо це. Візьмімо існуючий репозиторій, розділимо його на два, один новітній, інший історичний, та потім побачимо, як можна возз'єднати їх без редагування значень SHA-1 новітнього репозиторію за допомогою `replace`.

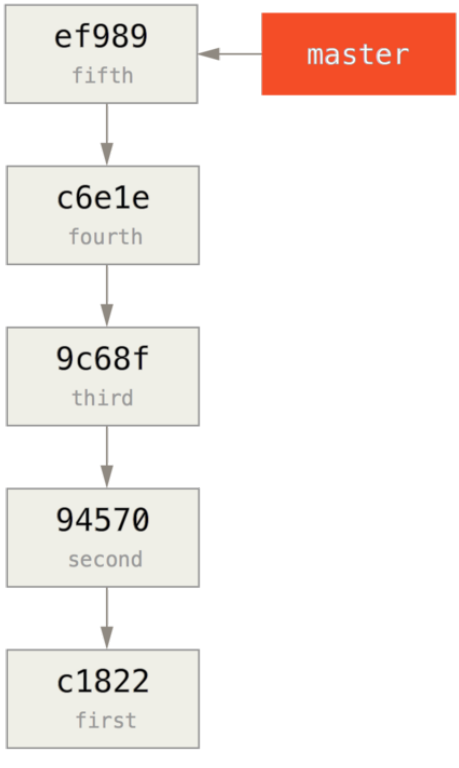
Ми використаємо простий репозиторій з п'ятьма простими комітами:

```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
```

```
945704c second commit
c1822cf first commit
```

Ми бажаємо розбити це на дві лінії історії. Одна буде з першого до четвертого коміту - це буде історичною. Друга лінія буде містити лише четвертий та п’ятий коміти - це буде новітня історія.

ЗОБРАЖЕННЯ 7-28

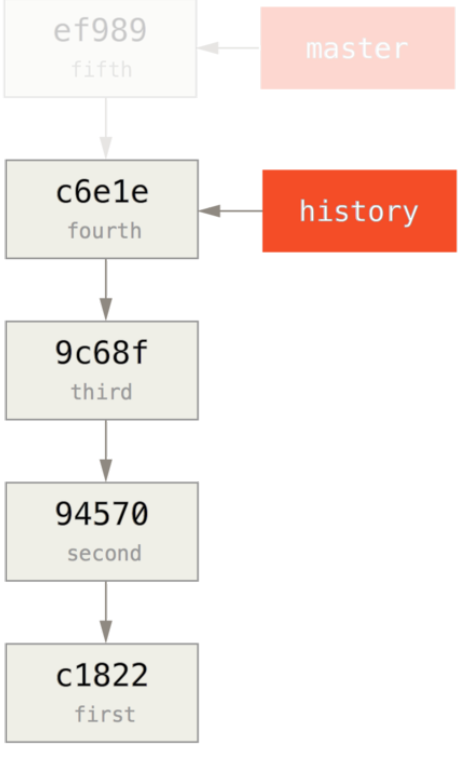


Ну, створити давню історію просто: треба лише покласти гілку в історію, а потім надіслати цю гілку до гілки master нового віддаленого репозиторію.

```
$ 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
```

ЗОБРАЖЕННЯ 7-29



Тепер ми можемо надіслати нову гілку `history` до гілки `master` нового репозиторію:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
```

```
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

Добре, отже наша історія опублікована. Тепер складніше завдання — зрізати нашу новітню історію, щоб зменшити її. Нам потрібне перекривання, щоб ми могли замінити коміт одного репозиторію еквівалентним комітом з іншого, отже ми збираємося зрізати до четвертого та п'ятого комітів (і четвертий коміт перекривається).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

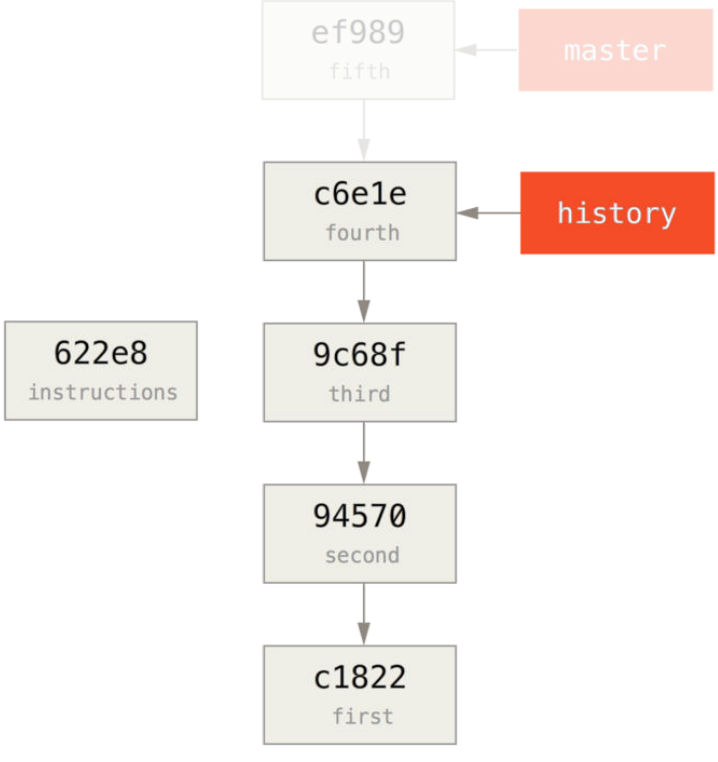
У даному випадку корисно створити базовий коміт, що має інструкції щодо розширення історії, щоб інші розробники знали, що треба робити, якщо наштовхнуться на перший коміт зрізаної історії, а потребують більшого. Отже, ми збираємось створити об'єкт первинного коміту в якості нашої базової точки з інструкціями, потім перебазуємо решту комітів (четвертий та п'ятий) поверх нього.

Щоб це зробити, нам треба вибрати точку розриву, у даному випадку це третій коміт, тобто `9c68fdc` словами SHA. Отже, наш базовий коміт буде засновано на цьому дереві. Ми можемо створити наш базовий коміт за допомогою команди `commit-tree`, яка просто бере дерево та поверне нам SHA-1 новісінького коміту без батьків.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

Команда `commit-tree` є однією з набору команд, які, зазвичай, називають *кухонними* (plumbing). Це команди, які, щиро кажучи, створені не для безпосереднього використання, а для використання **іншими** командами Git, щоб виконувати менші завдання. Коли ж доводиться робити більш чудернацькі речі, на кшталт описаного тут, вони дозволяють робити дійсно низькорівневі речі, проте, не призначені для щоденного користування. Ви можете дізнатись більше про кухонні команди в **"Сантехніка та порцеляна"**

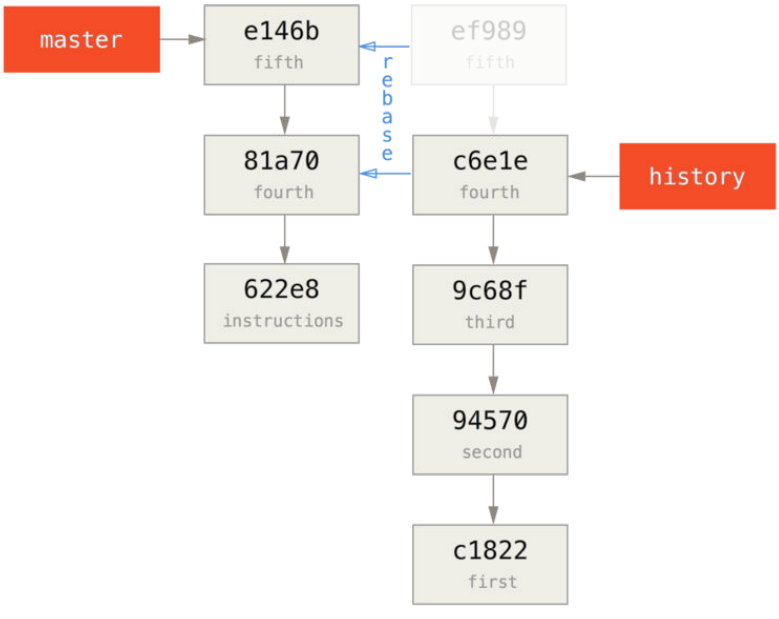
ЗОБРАЖЕННЯ 7-30



Добре, отже тепер, коли в нас є базовий коміт, ми можемо перебазувати решту нашої історії поверх нього за допомогою `git rebase --onto`. Параметр `--onto` треба встановити у SHA-1 суму, щойно отриману від `commit-tree`, а місцем перебазування буде третій коміт (батько першого коміту, який треба зберегти, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

ЗОБРАЖЕННЯ 7-31



Добре, отже тепер ми переписали нашу новітню історію поверх технічного базового коміту, який тепер містить інструкції щодо відновлення повної історії, якщо комусь треба. Тепер ми можемо надіслати цю історію до нового проекту, та тепер при клонуванні цього репозиторію, вони побачать лише останні два коміти, а потім базовий коміт з інструкціями.

Поміняймося роллю з кимось, хто клонує проект вперше, та потребує повної історії. Щоб отримати дані історії після клонування зрізаного репозиторію, потрібно додати друге віддалене сховище — історичний репозиторій, та отримати з нього зміни:

```
$ 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
 * [new branch]      master     -> project-history/master
```

Тепер співробітники матимуть нові коміти в гілці `master`, а історичні коміти у гілці `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
```

Щоб їх об’єднати, можна просто викликати `git replace` з комітом, який ви хочете замінити, та комітом, яким ви бажаєте його замінити. Отже, ми хочемо замінити “четвертий” коміт з гілки master “четвертим” комітом з гілки `project-history/master`:

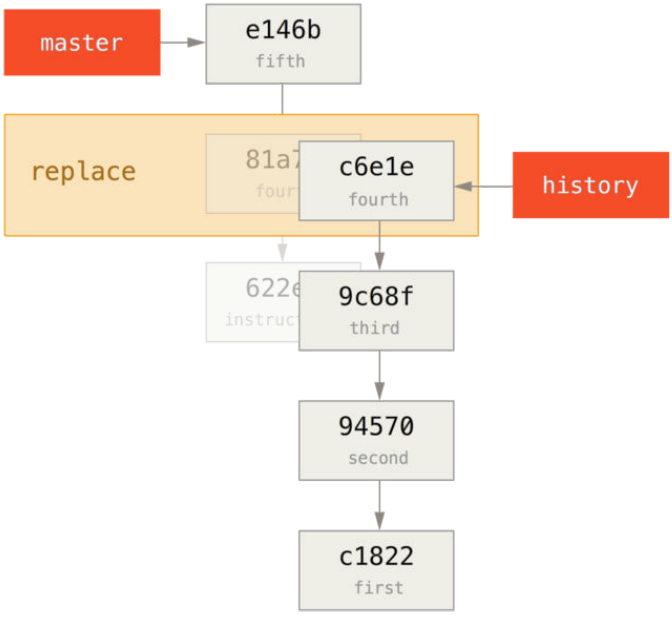
```
$ git replace 81a708d c6e1e95
```

Тепер, якщо подивитись на історію гілки `master`, виявляється, вона матиме такий вигляд:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Вражає, еге ж? Без необхідності змінювати всі SHA-1 суми першоджерела, ми впорались замінити один коміт історії геть іншим, і всі звичайні інструменти (`bisect`, `blame` тощо) працюватимуть, як ми й очікуємо.

ЗОБРАЖЕННЯ 7-32



Що цікаво, тут досі зазначено `81a708d` як SHA-1, хоч насправді використовуються дані коміту `c6e1e95`, яким ми його замінили. Навіть якщо виконати таку команду, як `cat-file`, вона все одно покаже замінені дані:

```
$ 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
```

Пам’ятайте, що справжній батько коміту `81a708d` був наш коміт-заповнювач (`622e88e`), а не `9c68fdce`, як тут зазначено.

Ще одна цікава річ: відомості про це зберігаються в посиланнях:

```
$ 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/81a708dd0e167a3f691541c7a646334
```

Це означає, що поділитися нашою заміною з іншими просто, адже ми можемо надіслати це посилання до сервера, а інші можуть легко завантажити його. Це не дуже корисно в розглянутому тут сценарії з прищепленням історії (адже тоді всім доведеться завантажувати обидві історії, то ж яка в тому користь?), проте це може бути корисним за інших обставин.

Збереження посвідчення (credential)

Якщо ви використовуєте протокол SSH для з'єднання з віддаленими сховищами, то можете використовувати ключ без пароля, що дозволяє безпечно передавати дані без набирання ім'я користувача та пароля. Втім, це неможливо з HTTP протоколами – кожне з'єднання потребує імені та пароля. Все стає ще складнішим з двох-кроковою авторизацією: значення, яке треба використати як пароль, випадково згенероване та невимовне.

На щастя, Git має систему посвідчень, яка може тут зарадити. Щойно встановлений Git пропонує чимало опцій:

- Типова поведінка — взагалі нічого не запам'ятовувати. Кожне з'єднання потребує від вас ім'я користувача та пароль.
- Режим "cache" зберігає посвідчення в пам'яті визначений термін. Жоден пароль не зберігається на диску, та вичищається з пам'яті за 15 хвилин.
- Режим "store" зберігає посвідчення до простого текстового файлу на диску, та ніколи не застаріває. Це означає, що доки ви не зміните пароль для Git, вам ніколи не доведеться набирати ваші дані знов. Недоліком цього методу є те, що ваші паролі зберігаються текстом у простому файлі в домашній директорії.
- Якщо ви використовуєте Mac, Git має режим "osxkeychain", який зберігає посвідчення у безпечному ланцюгу ключів (keychain), що є прив'язаним до вашого системного облікового запису. Цей метод зберігає посвідчення на диску і ніколи не застаріває, проте

його зашифровано так само, як система зберігає сертифікати HTTPS та автозаповнювачі Safari.

- Якщо ви використовуєте Windows, то можете встановити помічник (helper) під назвою "winstore." Це схоже на описаний вище "osxkeychain", проте використовує Windows Credential Store для контролю за приватною інформацією. Його можна знайти за адресою ***https://gitcredentialstore.codeplex.com***.

Щоб задати один з цих методів, треба встановити значення налаштування Git:

```
$ git config --global credential.helper cache
```

Деякі з цих помічників мають опції. Помагач "store" може приймати аргумент `--file <шлях>`, який задає, куди зберігається текстовий файл (типове значення `~/.git-credentials`). Помагач "cache" приймає опцію `--timeout <секунд>`, яка змінює термін, протягом якого демон працює (типове значення "900", тобто 15 хвилин). Ось приклад, як можна налаштувати помічник "store" особистим іменем файлу:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git навіть дозволяє налаштувати декілька помічників. При пошуку посвідчення для певного хосту, Git зробить запит до них по черзі та зупиниться при першій відповіді. При збереженні посвідчень, Git надішле ім'я користувача та пароль до **всіх** помічників зі списку і вони самі можуть вибрати, що з ними робити. Ось як виглядав би `.gitconfig`, якби у вас був файл посвідчень на зовнішньому носії, проте ви бажали б використати кеш у пам'яті, щоб заощадити набирання, якщо носій не підключено:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Під капотом

Як все це працює? Головною командою Git з системи помічників з посвідченнями є `git credential`, яка приймає команду в якості аргументу, а потім ще ввід через stdin.

Можливо це легше зрозуміти за допомогою прикладу. Припустімо, помічник посвідчень вже налаштовано, та він вже зберіг посвідчення для `mygithost`. Ось сесія, що використовує команду “fill”, яка викликається при спробі знайти посвідчення для хосту:

```
$ 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 Ця команда розпочинає взаємодію.

❷ Відтак Git-credential очікує вводу з stdin. Ми кажемо йому що знаємо: протокол та ім’я хосту.

❸ Порожній рядок означає, що ввід завершено, та система посвідчень має відповісти, що вона знає.

❹ Далі Git-credential приймає керування та пише до stdout інформацію, яку знайшов.

❺ Якщо посвідчень не знайдено, Git запитує в користувача ім’я та пароль, та видає їх назад до stdout, з якого був викликаний (у даному випадку вони під’єднані до однієї консолі).

Насправді, система посвідчень викликає програму, яка відокремлена від власно Git; яку саме залежить від значення налаштування `credential.helper`. Ось декілька форм, які воно може мати:

Значення налаштування	Поведінка
`foo`	Виконує `git-credential-foo`
`foo -a --opt=bcd`	Виконує `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Виконує `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Код після `!` передається на виконання до оболонки (shell)

Отже, вищеописані помічники насправді мають назви `git-credential-cache`, `git-credential-store` тощо, та ми можемо їх налаштувати, щоб вони приймали аргументи командного рядка. Загальна форма для цього “git-credential-foo [аргументи] <дія>.” Протокол stdin/stdout такий самий, як для git-credential, проте вони використовують трохи інших набір дій:

- `get` — це запит пари імені/пароля.
- `store` — це запит на зберігання набору посвідчень у пам’яті помічника.
- `erase` — очистити посвідчення для наданих властивостей з пам’яті помічника.

Для дій `store` та `erase` відповідь не потрібна (Git все одно її ігнорує). Втім, щодо дії `get`, Git дуже зацікавлений у тому, що скаже помічник. Якщо помічник не знає нічого корисного, він може просто вийти без виводу, проте, якщо він щось знає, він має доповнити прийняту інформацію тою, яку зберіг. Вивід сприймається як послідовність виразів присвоєння; будь-що надане замінить те, що Git вже знає.

Ось такий саме приклад, як і попередній, проте пропустимо git-credential та перейдемо відразу до 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 Тут ми кажемо `git-credential-store` зберегти деякі посвідчення: ім'я "bob" та пароль "s3cre7" мають використовуватись при доступі до `https://mygithost`.

❷ Тепер ми отримаємо це посвідчення. Ми надаємо вже відомі частини з'єднання (`https://mygithost`) та порожній рядок.

❸ `git-credential-store` відповідає збереженими вище ім'ям користувача та паролем.

Ось як виглядає файл `~/git.store`:

```
https://bob:s3cre7@mygithost
```

Це просто послідовність рядків, кожен з яких містить декорований посвідченням URL. Помагачі `osxkeychain` та `winstore` використовують локальний формат своїх сховищ, у той час як `cache` використовує власний формат в пам'яті (яку жоден інший процес не може прочитати).

Спеціальний кеш посвідчень

Враховуючи, що `git-credential-store` та її друзі є окремими від Git програмами, нескладно зрозуміти, що *будь-яка* програма може бути помічником посвідчень Git. Помагачі, які постачає Git, доречні в багатьох поширених випадках, проте не у всіх. Наприклад, скажімо, ваша команда має якісь посвідчення, які використовуються всією командою, можливо для розробки. Вони зберігаються у спільній директорії, проте, ви не бажаєте копіювати їх до вашого власного сховища посвідчень, адже вони часто змінюються. Жоден з існуючих помічників тут не зарадить; подивімося, що доведеться зробити, щоб написати свій власний. Є декілька ключових функцій, які ця програма має виконувати:

1. Єдина дія, якій треба приділити увагу — це `get`; `store` та `erase` є операціями запису, отже просто завершимо програму без наслідків при отриманні їх.
2. Формат спільного файлу посвідчень такий самий, як і той, що використовує `git-credential-store`.
3. Розташування цього файлу доволі стале, проте варто надати користувачу можливість змінювати шлях до нього, про всяк випадок.

Ще раз наголошуємо, ми напишемо цей додаток на Ruby, проте це можна зробити будь-якою мовою, якщо Git може виконати результат. Ось повний вихідний код нового помічника посвідчень:

```ruby
#!/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 Тут ми розбираємо опції командного рядка: дозволяємо користувачу задати вхідний файл. Типове значення — `~/.git-credentials`.

❷ Ця програма відповідає виключно, якщо задана дія `get` та файл з даними існує.

❸ Цей цикл читає з stdin доки не зустріне перший порожній рядок. Вхід зберігається в хеші `known` для подальшого використання.

❹ Цей цикл читає вміст файлу з даними — шукає відповідності. Якщо протокол та хост з `known` відповідають поточному рядку, програма друкує результати до stdout та завершує свою роботу.

Ми збережемо наш помічник як `git-credential-read-only`, покладемо кудись до нашого PATH та позначимо як виконáнний. Ось як виглядає інтерактивна сесія:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Оскільки його назва починається з “git-”, ми можемо використати простий синтаксис для значення налаштування:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Як ви можете бачити, розширення системи є доволі прямолінійним, та може вирішувати якісь повсякденні завдання для вас та вашої команди.

Підсумок

Ви побачили чимало інструментів підвищеної складності, що дозволяють більш тонко маніпулювати комітами та областю індексування. Коли у вашому коді виникнуть проблеми, вам має бути

легко довідатись, який коміт є їх причиною, коли та ким він був створений. Якщо ви бажаєте використовувати підпроекти у вашому проекті, ви дізналися, як задовольнити ці потреби. Наразі, ви можете робити більшість того, що вам буде потрібно в командному рядку Git у повсякденному його використанні та ви маєте почуватись зручно при цьому.

8 Налаштування Git

До цього часу ми розглянули основи роботи Git і як його використовувати, а також ми розповіли про декілька засобів, які надає Git для легшого та ефективнішого його використання. У цьому розділі ми побачимо, як можна змусити Git працювати бажаним чином, запровадивши декілька важливих конфігураційних налаштувань. Із цими засобами легко змусити Git працювати точно так, як потрібно вам, вашій компанії чи групі.

Конфігурація Git

Як ви вже коротко бачили в **Chapter 1**, можна вказати конфігураційні налаштування Git використовуючи команду `git config`. Одна з перших речей, що ви зробили — це вказали ваше ім'я та електронну адресу:

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

Тут ви вивчите декілька більш цікавих налаштувань для пристосування Git у такий же самий спосіб.

Спочатку швидке резюме: Git використовує ряд конфігураційних файлів для визначення нетипової поведінки, яка вам може знадобитися. Перше місце, куди Git заглядає за цими значеннями, знаходиться у файлі `/etc/gitconfig`, який містить значення для всіх користувачів системи та всіх їхніх репозиторіїв. Якщо передати опцію `--system` до `git config`, то він буде читати і записувати саме в цей файл.

Наступне місце, куди дивиться Git — це файл `~/.gitconfig` (або `~/.config/git/config`), який є специфічним для кожного користувача.

Ви можете змусити Git читати і записувати в цей файл за допомогою опції `--global`.

Нарешті, Git шукає конфігураційні значення у файлі налаштувань, що розміщений у директорії Git-а (`.git/config`) будь-якого поточного репозиторію, що ви використовуєте. Ці значення є специфічними для цього окремого репозиторію.

Кожен з цих "рівнів" (системний, глобальний, локальний) переписує значення попереднього рівня, тобто, наприклад, значення в `.git/config` перекриває значення в `/etc/gitconfig`.

Конфігураційні файли Git-а є звичайним текстом, тому ви можете задати ці значення, відредагувавши файл вручну дотримуючись коректного синтаксису. Однак у загальному випадку легше виконати команду `git config`.

Базові налаштування клієнта

Конфігураційні опції, що розпізнаються Git-ом, належать до двох категорій: клієнтські та серверні. Більшість опцій стосуються сторони клієнта — налаштування ваших персональних робочих вподобань. Існує багато, *багато* конфігураційних опцій, але велика частка їх корисна лише в певних крайніх випадках. Ми розкриємо тут лише найзагальніші та найкорисніші. Щоб побачити список усіх опцій, підтримуваних вашою версією Git, виконайте

```
$ man git-config
```

Ця команда виводить список усіх доступних опцій з доволі докладним описом. Також ви можете знайти цю інформацію на ***http://git-scm.com/docs/git-config.html***.

`CORE.EDITOR`

За замовчуванням, Git використовує будь-що вказане вами як типовий текстовий редактор (`$VISUAL` чи `$EDITOR`), або повертається до редактора `vi`, щоб створювати чи вредагувати ваші повідомлення до комітів та теґів. Щоб змінити типові налаштування на щось інше, використовуйте опцію core.editor`:

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

Тепер не важливо, що вказано як типовий редактор для терміналу, Git буде викликати Emacs для редагування повідомлень.

COMMIT.TEMPLATE

Якщо ви вкажете для цього параметра шлях до файла у вашій системі, Git буде використовувати той файл як типове значення для повідомлення при створенні коміту. Наприклад, припустимо, що ви створили файл шаблону `~/.gitmessage.txt` з наступним вмістом:

```
subject line

what happened

[ticket: X]
```

Щоб вказати Git, що потрібно використовувати його як типове значення для повідомлення, яке з'являтиметься в редакторі після виконання команди `git commit`, потрібно встановити значення для `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Після цього, під час створення коміту, ваш редактор буде відкривати щось на зразок наступного в якості заповнювача для повідомлення коміту:

```
subject line

what happened

[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
```

Якщо ваша команда має певні вимоги до повідомлення коміту, то встановлення шаблону для цих вимог у системі та налаштування Git, щоб він використовував його за замовчуванням, збільшить шанси регулярного дотримання цих вимог.

CORE.PAGER

Це налаштування визначає, який прогортач сторінок використовується, коли Git прогортає виведення, таке як `log` і `diff`. Ви можете встановити його в значення `more` чи ваш улюблений прогортач сторінок (за замовчуванням — це `less`), чи вимкнути його, задавши як значення порожній рядок:

```
$ git config --global core.pager ''
```

Якщо ви виконаєте це, Git буде відображати увесь результат всіх команд, незалежно від його довжини.

USER.SIGNINGKEY

Якщо ви створюєте підписані анотовані теґи (як це обговорювалось у **“Підписання праці”**), то встановлення вашого GPG ключа підпису в конфігураційних налаштуваннях зробить це простішим. Вкажіть ID вашого ключа наступним чином:

```
$ git config --global user.signingkey <gpg-key-id>
```

Тепер ви можете підписувати теґи без необхідності кожного разу вказувати ваш ключ у команді `git tag`:

```
$ git tag -s <tag-name>
```

CORE.EXCLUDESFILE

Ви можете задати шаблони у файлі `.gitignore` вашого проекту для того, щоб Git не бачив їх як невідслідковувані та не намагався індексувати при виконанні команди `git add` на них, як це обговорювалось у **“Ігнорування файлів”**.

Але іноді виникає потреба ігнорувати певні файли у всіх сховищах, з якими ви працюєте. Якщо ваш комп'ютер працює на Mac OS X, ви, напевно, знайомі з файлами `.DS_Store`. Якщо ж ви надаєте перевагу редактору Emacs, то знаєте про файли, що закінчуються символом ~.

Ця опція дозволяє написати щось на зразок глобального файлу `.gitignore`. Якщо ви створите файл `~/.gitignore_global` з наступним вмістом:

```
*~
.DS_Store
```

...та виконаєте команду `git config --global core.excludesfile ~/.gitignore_global`, то Git більше ніколи не буде турбувати вас щодо цих файлів.

HELP.AUTOCORRECT

Якщо ви помиляєтесь при наборі команди, то виводиться щось подібне до цього:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
    checkout
```

Git послужливо намагається зрозуміти, що ви мали на увазі, але відмовляється виконувати це. Якщо встановити значення `help.autocorrect` рівним 1, то Git буде все ж таки виконувати цю команду:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Зверніть увагу на це дивне "0.1 seconds". Насправді значення опції `help.autocorrect` є цілим числом, що представляє десяті секудни. Тому, якщо встановити його рівним 50, то Git дасть вам 5 секунд, щоб змінити свою думку перед виконанням відкоректованої команди.

Кольори у Git

Git повністю підтримує кольорове виведення у терміналі, що робить візуальний розбір результатів виконання команд значно швидшим та легшим. Чимало опцій можуть допомогти налаштувати кольори відповідно до ваших вподобань.

`COLOR.UI`

Git автоматично розфарбовує більшу частину свого виведення, але є головний вимикач, якщо вам не подобається така поведінка. Щоб вимкнути кольорове виведення Git у терміналі, зробіть наступне:

```
$ git config --global color.ui false
```

Типовим налаштуванням є `auto`, що розфарбовує виведення, коли воно йде прямо у термінал, але уникає керування кольором, коли виведення переадресовується у канал чи файл.

Ви також можете встановити його у значення `always` для ігнорування різниці між терміналами та каналами. Вам рідко хотітиметься цього; у більшості сценаріїв, якщо вам потрібне кольорове кодування у переадресованому виведенні, можна передати параметр `--color` в команду Git, щоб змусити його використовувати кольорове кодування. Типове значення майже завжди є тим, чого ви бажаєте.

`COLOR.*`

Якщо ви хочете вказати більш конкретно, яку команду розфарбовувати і як — Git постачає налаштування кольору для конкретних команд-дієслів. Кожна з них може бути задана в `true`, `false` або `always`:

```
color.branch
color.diff
color.interactive
color.status
```

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. If you want an attribute like bold in the previous example, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground and background).

External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

If you want to try this out, P4Merge works on all major platforms, so you should be able to do so. We'll use path names in the examples that work on Mac and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, download P4Merge from ***http://www.perforce.com/downloads/Perforce/***. Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Set up a merge wrapper script named `extMerge` that calls your binary with all the arguments provided:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Because you only want the `old-file` and `new-file` arguments, you use the wrapper script to pass the ones you need.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

You also need to make sure these tools are executable:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. So, you can either run four config commands

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

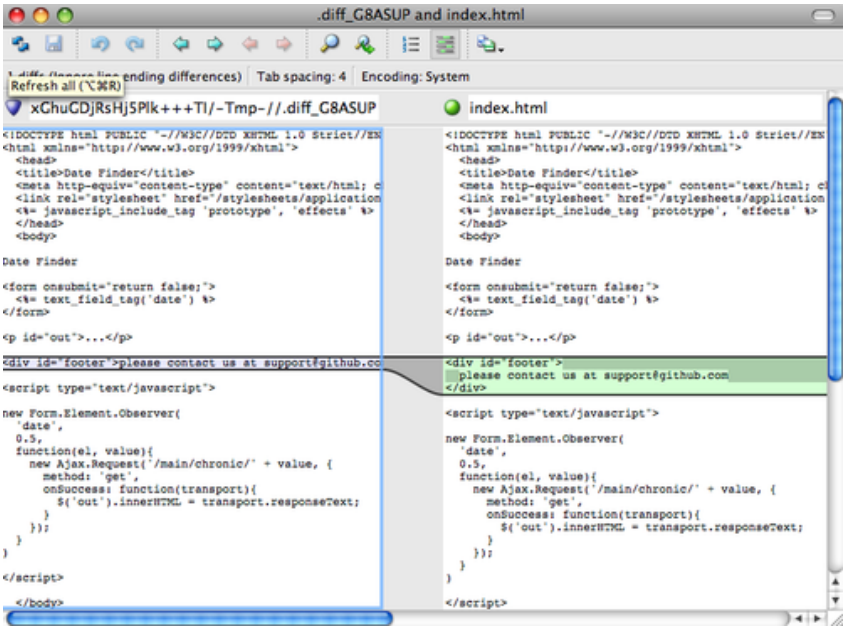
or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

After all this is set, if you run diff commands such as this:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:



ЗОБРАЖЕННЯ 8-1

P4Merge.

If you try to merge two branches and subsequently have merge conflicts, you can run the command `git mergetool`; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your `extDiff` and `extMerge` tools to run the KDiff3 tool instead, all you have to do is edit your `extMerge` file:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. To see a list of the tools it supports, try this:

```
$ 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.
```

If you're not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the kdiff3 command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

If you run this instead of setting up the `extMerge` and `extDiff` files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

CORE.AUTOCRLF

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the `core.autocrlf` setting. If you're on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:

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

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting `core.autocrlf` to input:

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

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

CORE.WHITESPACE

Git comes preset to detect and fix some whitespace issues. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The ones that are turned on by default are `blank-at-eol`, which looks for spaces at the end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a `-` in front of the value. For example, if you want all but `cr-at-eol` to be set, you can do this:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you've committed whitespace issues but haven't yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

RECEIVE.FSCKOBJECTS

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn't do this by default; it's a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren't introducing corrupt data.

RECEIVE.DENYNONFASTFORWARDS

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you're doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we'll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

RECEIVE.DENYDELETES

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn in "Приклад політики користування виконуваної Git-ом".

Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don't want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you'll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You'll see how to tell Git which is which.

IDENTIFYING BINARY FILES

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text Javascript data format) data-

set written out to disk by the IDE, which records your build settings and so on. Although it's technically a text file (because it's all UTF-8), you don't want to treat it as such because it's really a lightweight database – you can't merge the contents if two people change it, and diffs generally aren't helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj binary
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

DIFFING BINARY FILES

You can also use the Git attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the "word" filter when you try to view a diff that contains changes. What is the "word" filter? You have to set it up. Here you'll configure Git to use the

`docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from ***http://docx2txt.sourceforge.net***. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash
docx2txt.pl $1 -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the "word" filter, which is defined as the `docx2txt` program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:

```
$ 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 beginni
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system t
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image c
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another dire
```

Git successfully and succinctly tells us that we added the string "Testing: 1, 2, 3.", which is correct. It's not perfect – formatting changes wouldn't show up here – but it certainly works.

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

If you replace an image in your project and run `git diff`, you see something like this:

```
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
```

You can easily see that the file size and image dimensions have both changed.

Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `$Id$` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself:

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

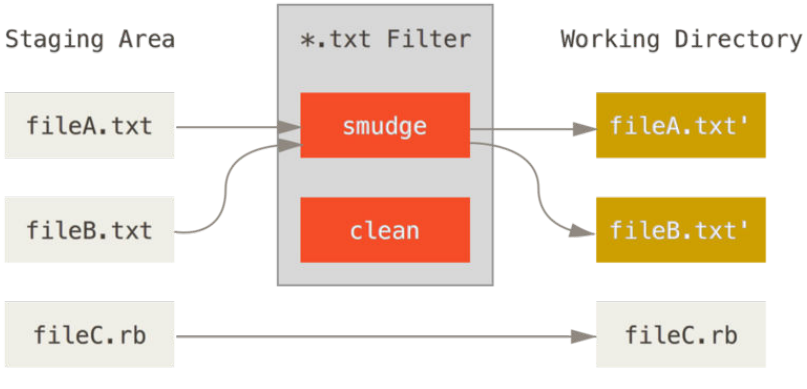
```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

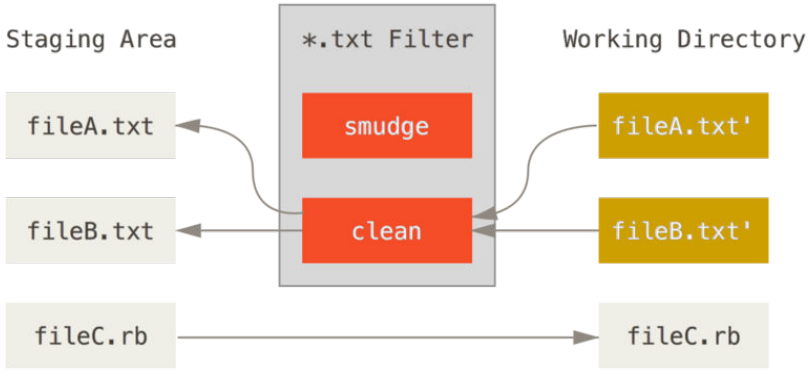
However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a datestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by looking at them.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are called "clean" and "smudge" filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out ("smudge", see **Figure 8-2**) and just before they're staged ("clean", see **Figure 8-3**). These filters can be set to do all sorts of fun things.

ЗОБРАЖЕННЯ 8-2

The "smudge" filter is run on checkout.





ЗОБРАЖЕННЯ 8-3

The “clean” filter is run when files are staged.

The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the “indent” filter:

```
*.c filter=indent
```

Then, tell Git what the “indent” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the indent program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you're most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files on checkout. You'll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a file with your `$Date$` keyword and then setting up a Git attribute for that file that engages the new filter:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ 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
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn't, so it won't work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

EXPORT-IGNORE

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run git archive to create a tarball of your project, that directory won't be included in the archive.

EXPORT-SUBST

When exporting files for deployment you can apply `git log`'s formatting and keyword-expansion processing to selected portions of files marked with the `export-subst` attribute.

For instance, if you want to include a file named `LAST_COMMIT` in your project, and have metadata about the last commit automatically injected into it when `git archive` runs, you can for example set up the file like this:

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of the archived file will look like this:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

The substitutions can include for example the commit message and any git notes, and git log can do simple word wrapping:

```
$ 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.
```

The resulting archive is suitable for deployment work, but like any exported archive it isn't suitable for further development work.

Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Installing a Hook

The hooks are all stored in the `hooks` subdirectory of the Git directory. In most projects, that's `.git/hooks`. When you initialize a new repository with `git init`, Git populates the hooks directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine – you can write them in Ruby or Python or what have you. If you want to use the bundled hook scripts, you'll have to rename them; their file names all end with `.sample`.

To enable a hook script, put a file in the `hooks` subdirectory of your Git directory that is named appropriately and is executable. From that point forward, it should be called. We'll cover most of the major hook filenames here.

Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, e-mail-workflow scripts, and everything else.

It's important to note that client-side hooks are **not** copied when you clone a repository. If your intent with these scripts is to enforce a policy, you'll probably want to do that on the server side; see the example in "Приклад політики користування виконуваної Git-ом".

COMMITTING-WORKFLOW HOOKS

The first four hooks have to do with the committing process.

The `pre-commit` hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with `git commit --no-verify`. You can

do things like check for code style (run `lint` or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

The `prepare-commit-msg` hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few parameters: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, We'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the `post-commit` hook runs. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Generally, this script is used for notification or something similar.

E-MAIL WORKFLOW HOOKS

You can set up three client-side hooks for an e-mail-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over e-mail prepared by `git format-patch`, then some of these may be helpful to you.

The first hook that is run is `applypatch-msg`. It takes a single argument: the name of the temporary file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted, or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via `git am` is `pre-applypatch`. Somewhat confusingly, it is run *after* the patch is applied but before a commit is made, so you can use it to inspect the snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don't pass, exiting non-zero aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`, which runs after the commit is made. You can use it to notify a group or the author of the patch you pulled in that you've done so. You can't stop the patching process with this script.

OTHER CLIENT HOOKS

The `pre-rebase` hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example `pre-rebase` hook that Git installs does this, although it makes some assumptions that may not match with your workflow.

The `post-rewrite` hook is run by commands that replace commits, such as `git commit --amend` and `git rebase` (though not by `git filter-branch`). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on `stdin`. This hook has many of the same uses as the `post-checkout` and `post-merge` hooks.

After you run a successful `git checkout`, the `post-checkout` hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don't want source controlled, auto-generating documentation, or something along those lines.

The `post-merge` hook runs after a successful `merge` command. You can use it to restore data in the working tree that Git can't track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

The `pre-push` hook runs during `git push`, after the remote refs have been updated but before any objects have been transferred. It receives the name and location of the remote as parameters, and a list of to-be-updated refs through `stdin`. You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).

Git occasionally does garbage collection as part of its normal operation, by invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before the garbage collection takes place, and can be used to notify you that this is happening, or to abort the collection if now isn't a good time.

Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre

hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that's as complex as you wish.

PRE-RECEIVE

The first script to run when handling a push from a client is `pre-receive`. It takes a list of references that are being pushed from stdin; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they're modifying with the push.

UPDATE

The `update` script is very similar to the `pre-receive` script, except that it's run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, `pre-receive` runs only once, whereas update runs once per branch they're pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the update script exits non-zero, only that reference is rejected; other references can still be updated.

POST-RECEIVE

The `post-receive` hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the `pre-receive` hook. Examples include e-mailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can't stop the push process, but the client doesn't disconnect until it has completed, so be careful if you try to do anything that may take a long time.

Приклад політики користування виконуваної Git-ом

У цьому розділі ви використаєте те, що вже вивчили для створення робочого процесу Git, який перевірятиме певний формат повідомлення коміту, а також дозволятиме лише обраним користувачам змінювати певні піддиректорії в проекті. Ви побудуєте клієнтські скрипти, які допомагатимуть розробникові перевіряти чи їхні надіслані зміни (push)

будуть відхилені, і серверні скрипти, які будуть примушувати дотримуватись політики користування.

Скрипти, які ми продемонструємо, написані на Ruby; частково через нашу інтелектувальну інертність, а частково тому, що Ruby читабельна, навіть якщо ви не можете на ній нічого написати. Втім, будь-яка мова підійде – всі приклади скриптів-гаків поширювані з Git, написані або на Perl або на Bash, тому ви можете знайти багато зразків гаків, імплементованих цими мовами.

Гак серверної частини

Вся робота для серверної частини потрапить до файлу `update` у вашій директорії `hooks`. Гак `update` виконується одноразово для кожної гілки, коли зміни надсилаються, і приймає три аргументи:

- Ім'я посилання, до якого надсилаються зміни
- Стара ревізія, де ця гілка була
- Нова ревізія, яка надсилається

У вас також є доступ до користувача, який надсилає зміни, якщо пуш відбувається через SSH. Якщо ви дозволили з'єднання кожному через одного користувача (як "git") з публічним ключем автентифікації, вам необхідно буде надати цьому користувачеві обгортку терміналу, що визначатиме користувача, який підключається на підставі публічного ключа, і відповідно задати змінну середовища. Тут ми припустимо, що користувач, який під'єднується, знаходиться в змінній середовища `$USER`, так що ваш скрипт оновлення починає зі збирання усієї необхідної вам інформації:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Так, це глобальні змінні. Не засуджуйте – так легше продемонструвати.

ВІДПОВІДНІСТЬ ПОВІДОМЛЕННЯ КОМІТУ ПЕВНОМУ ФОРМАТУ

Ваше перше випробування полягає у тому, щоб змусити кожне повідомлення коміту відповідати певному формату. Лише для прикладу, припустіть, що кожне повідомлення повинне містити рядок, який виглядає як “ref: 1234”, тому що ви хочете кожен коміт прив’язати до робочого елементу у вашій системі керування завданнями. Вам необхідно взяти кожен коміт, що надсилається, перевірити чи цей рядок є в повідомленні цього коміту, і якщо він відсутній, то вийти з не-нуль, щоб цей пуш був відхилений.

Ви можете дістати список SHA-1 значень усіх комітів, які надсилаються, взявши `$newrev` і `$oldrev` значення і передавши їх в кухонну команду Git, яка називається `git rev-list`. Це фактично команда `git log`, але типово вона видає лише SHA-1 значення і ніякої іншої інформації. Отже, щоб отримати список усіх проміжних SHA-1 комітів між одним комітом і іншим, ви можете виконати щось подібне до цього:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Ви можете взяти цей результат, пройтись по кожному з цих SHA-1 значень комітів, повернути його повідомлення, і протестувати це повідомленя за допомогою регулярних виразів, які шукатимуть відповідний паттерн.

Вам необхідно дізнатись, як отримати кожне повідомлення коміту для перевірки. Для того, щоб отримати необроблені дані коміту, ви можете використати іншу кухонну команду, яка називається `git cat-file`. Ми пройдемось цими кухонними командами детальніше в **Chapter 10**; тим часом, ось що ця команда видає:

```
$ 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
```

Найпростіше отримати повідомлення коміту, якщо у вас є його SHA-1 значення, це перейти до першої порожньої стрічки і згребти все після неї. Ви можете це зробити командою sed на Юніксових системах:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

Ви можете використовувати ці чари для отримання повідомлення з кожного коміту, що надсилається, і вийти, якщо ви не бачите нічого що б не співпало. Для того, щоб вийти зі скрипта і відхилити зміни, які надсилаються, вийдіть не-нуль. Весь метод виглядає так:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Якщо ви покладете це у ваш скрипт update, то він відхилить оновлення, які містять коміти з повідомленнями, що не відповідають вашому правилу.

ВИКОНАННЯ СИСТЕМИ ACL ДЛЯ КЕРУВАННЯ СПИСКОМ КОРИСТУВАЧІВ

Припустимо, що ви хочете додати механізм, який використовує контрольний список доступу (КСД) (ACL - access control list), що визначає яким користувачам дозволяється надсилати зміни, і в які частини вашого проекту. Деякі користувачі мають повний доступ, а інші можуть надсилати зміни лише в певні піддерикторії чи до певних файлів. Для того щоб зробити це примусовим, ви запишете ці правила у файл, який називається acl і який живе у вашому порожньому Git репозиторії на сервері. Гак update буде звірятись з цими правилами, визначати, які файли несуть зміни в усіх комітах, які надсилаються, і чи

користувач, що надсилає зміни, має доступ для того, щоб оновити всі ті файли.

Перше що ви зробите, ви створите ваш ACL. Тут, ви використаєте формат дуже схожий на CVS ACL механізм: він використовує серії рядків, де перше поле це `avail` чи `unavail`, наступне поле це список користувачів, розділених комою, до яких це правило може бути застосоване, і останнє поле це шлях, до якого правило застосовується (порожнє означає відкритий доступ). Усі ці поля розділені знаком вертикальної лінії (|).

В цьому випадку, у вас є декілька адміністраторів, декілька письменників документації з доступом до директорії `doc`, і один розробник, який має доступ лише до директорій `lib` і `tests`, і ваш ACL файл виглядає так:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Ви починаєте зі зчитування цих даних в структуру, яку зможете використовувати. У даному випадку, щоб не ускладнювати приклад, ви змушуватимете дотримання лише директиви `avail`. Ось метод, що дає вам асоціативний масив, де ключ це ім'я користувача, а значення це масив шляхів, до яких користувач має доступ запису:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

Метод `get_acl_access_data` з файлу ACL, який ви розлянули раніше, повертає структуру даних, яка виглядає так:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
```

```
  "nickh"=>[nil],
  "pjhyett"=>[nil],
  "schacon"=>["lib", "tests"],
  "cdickens"=>["doc"],
  "usinclair"=>["doc"],
  "ebronte"=>["doc"]}
```

Тепер, коли доступ на місці, вам потрібно визначити, які шляхи змінюють коміти, що надсилаються, щоб ви впевнились, що користувач, який надсилає зміни, має доступ до них усіх.

Ви можете дуже просто побачити, які файли були змінені в одному коміті за допомогою опції `--name-only` для команди `git log` (згадана коротко у Розділі 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Якщо ви використовуєте структуру ACL, яку повертає метод `get_acl_access_data`, і зіставите її зі списком файлів у кожному коміті, ви можете визначити чи користувач має доступ для надсилання всіх їхніх комітів:

```ruby
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
           || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
```

```
  end
end

check_directory_perms
```

Список нових комітів, які надсилаються до вашого серверу, можна дізнатись використавши `git rev-list`. Потім, для кожного з цих комітів ви дізнаєтесь, які файли змінені, і впевнитесь, що користувач, який надсилає, має доступ до всіх шляхів, які змінюються.

Тепер ваші користувачі не можуть надсилати будь-які коміти з погано сформульованими повідомленнями чи зі зміненими файлами, які не належать до шляхів визначених для них.

ТЕСТУВАННЯ

Якщо ви виконаєте `chmod u+x .git/hooks/update`, що є файлом, в який ви б мали покласти увесь цей код, і потім спробуєте надіслати коміт з неправильним повідомленням, ви отримаєте щось подібне до цього:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
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'
```

Тут є декілька цікавих речей. По-перше, ви бачите, де гак починає виконуватись.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Пам’ятаєте, ви надрукували це на самому початку вашого скрипта оновлення. Все, що скрипт відлунює в `stdout` буде передано клієнтові.

Наступна річ, яку ви помітите, це повідомлення про помилку.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Перший рядок був видрукуваний вами, а інших два передав Git, який вам повідомляє, що скрипт оновлення вийшов з не-нуль і що він відхиляє надіслані вами зміни. В результаті ви маєте це:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Ви побачите повідомлення про відхилення віддаленим сховищем для кожного посилання, яке ваш гак відхилив, і це вам говорить про те, що він був відхилений саме через помилку, що сталась в гаку.

Крім того, якщо хтось намагається відредагувати файл до якого вони не мають доступу, і надсилають коміт, в якому цей файл присутній, то вони побачать щось подібне. Наприклад, якщо письменник документації намагатиметься надіслати коміт, який змінює щось в директорії `lib`, то він побачить

```
[POLICY] You do not have access to push to lib/test.rb
```

З цього моменту доки скрипт `update` знаходиться на місці і є виконуваним, ваш репозиторій ніколи не матиме повідомлення коміту без вашого паттерну в ньому, і ваші користувачі матимуть свій простір.

Гаки клієнтської частини

Недоліком даного підходу є скиглення, що неминуче почнеться через відхилення комітів, надісланих вашими користувачами. Відхилення ретельно опрацьованої ними роботи в останню хвилину, може дуже дратувати і плутати; тим більше вони будуть змушені відредагувати історію, щоб виправити її, що зазвичай не найприємніше заняття.

Вирішенням цієї дилеми може стати який-небуть гак для клієнта, що користувачі можуть використовувати для того, щоб дізнатись чи вони роблять щось, що може бути відхилене сервером. Таким чином, вони можуть виправити будь-які проблеми до того, як зберігати коміт, і перед тим, як ці помилки стає важче виправити. Через те, що гаки не передаються з клоном проекту, ви змушені поширювати ці скрипти у інший спосіб, а також спонукати користувачів зкопіювати скрипти в їхню директорію `.git/hooks` і зробити їх такими, що виконуються. Ви

можете поширювати ці гаки з проектом чи в окремому проекті, але Git не встановить їх автоматично.

Щоб розпочати, ви повинні перевірити ваше повідомлення коміту перед тим, як коміт буде записаний, так, аби ви знали, що сервер не відхилить ваші зміни через те, що повідомлення було погано відформатоване. Для цього ви можете додати гак `commit-msg`. Якщо вам вдастся зробити так, щоб він зчитував повідомлення від файлу, що передається як перший аргумент і перевірити його на паттерн, ви можете змусити Git перервати коміт, якщо паттерн не підтвердився:

```ruby
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Якщо цей скрипт на місці (в `.git/hooks/commit-msg`) і виконуваний, і ви комітете з повідомленням, що неправильно відформатоване, то ви побачите це:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

Жоден коміт не був завершений в попередньому прикладі. Проте, якщо ваше повідомлення містить правильний паттерн, то Git дозволить вам закомітити:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Далі ви хочете упевнитись, що файл, який ви змінюєте, не знаходиться за межами вашого ACL. Якщо директорія `.git` вашого проекту містить копію файлу ACL, який ви використовували раніше, тоді наступний скрипт `pre-commit` змушуватиме дотримуватись цих обмежень для вас:

```ruby
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

Це приблизно той самий скрипт, що і для серверної частини, але з двома важливими відмінностями. Перше, цей ACL файл знаходиться в іншому місці, тому що цей скрипт виконується з вашої робочої директорії, а не з вашої директорії `.git`. Потрібно змінити шлях до ACL файлу з цього

```ruby
access = get_acl_access_data('acl')
```

на цей:

```ruby
access = get_acl_access_data('.git/acl')
```

Інша велика відмінність це те, як ви отримуєте список файлів, що були змінені. Через те, що серверна частина дивиться в журнал комітів, і в даний момент цей коміт ще не записаний, то натомість цей список файлів вам необхідно отримати з індексу. Замість

```ruby
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

ви маєте використовувати

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Проте, це тільки дві відмінності – інакше цей скрипт працює так само. Тут одне застереження: цей скрипт має бути виконаний локально тим самим користувачем, що надсилає на віддалену машину. Якщо він відрізняється, то ви мусите змінити змінну `$user` вручну.

Додатково ми можемо упевнитись, що користувач не надсилає посилання, що не можуть бути перемотаними вперед. Щоб отримати посилання, що не є перемотувальним, ви або мусите перебазувати коміт, який ви вже надіслали, або спробувати надіслати іншу локальну гілку до тієї ж самої віддаленої гілки.

Імовірно, що сервер вже сконфігурований з `receive.denyDeletes` і `receive.denyNonFastForwards` для виконання цієї політики, тому ви можете спробувати упіймати лише випадкове перебазування комітів, що вже були надіслані.

Ось приклад скрипта для передперебазування (pre-rebase), що перевіряє це. Він бере список усіх комітів, які ви хочете перезаписати і перевіряє чи вони існують в будь-якому з ваших віддалених посилань. Якщо він бачить один досяжний з одного з ваших віддалених посилань, то він перериває перебазування.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

Цей скрипт використовує синтаксис, що не був розлянутий у Розділі 6 про Вибір ревізії. Ви отримаєте список комітів, які вже були надіслані виконавши наступне:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

SHA^@ синтаксис знаходить усіх батьків того коміту. Ви шукаєте будь-який коміт, досяжний з останнього коміту на віддаленій машині, і який недосяжний з будь-якого батьківського будь-яких SHA-1 значень, що ви намагаєтесь надіслати – це означає, що це є швидке перемотування вперед.

Найголовнішим недоліком цього підходу є те, що він може бути дуже повільним і часто непотрібним – якщо ви не намагаєтесь силою надіслати з `-f`, то сервер вас попередить і не прийме пуш. Проте, це цікава вправа і теоретично може допомогти вам уникнути перебазування, яке необхідно буде виправити повернувшись пізніше.

Підсумок

Ми розкрили більшість шляхів налаштування ваших клієнта та сервера Git так, щоб вони максимально відповідали вашому робочому процесу та проектам. Ви дізнались про всі види конфігураційних налаштувань, атрибутів файлів та перехоплювачів подій, а також ви побудували приклад сервера з дотриманням політик. Тепер вам повинно бути під силу змусити Git відповідати майже всім робочим процесам, які тільки можна уявити.

Git та інші системи контролю версій

9

Світ не ідеальний. Зазвичай, ви не зможете швидко перевести будь-який проект, над яким працюєте, на використання Git. Іноді вам доведеться мати справу з проектами, де використовується інша система контролю версій, хоча вам би й хотілося, щоб це був Git. У першій частині цього розділу ви дізнаєтесь про способи використання Git в якості клієнта для роботи з проектом, який розміщений в іншій системі.

В якусь мить ви, можливо, захочете перевести ваш проект на Git. У другій частині цього розділу ви дізнаєтесь як провести міграцію з деяких поширених систем на Git, а також ознайомитесь з методом, який буде працювати в ситуаціях, коли готових інструментів для міграції не існує.

Git як клієнт

Git справляє настільки позитивне враження на розробників, що багато з них вигадують способи використання Git на своєму комп'ютері навіть тоді, коли решта команди використовує іншу систему контролю версій. Для цього розроблено багато спеціальних адаптерів, які називаються "мостами" ("bridges"). Тут ми розглянемо ті адаптери, з якими вам, найімовірніше, доведеться мати справу при роботі над реальними проектами.

Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS

choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

GIT SVN

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

SETTING UP

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion. For these tests, we created a new Subversion repository on Google Code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \
  http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ 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.
[…]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it's ridiculously inefficient, but it's the only easy way to do this.

GETTING STARTED

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository,

you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ 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
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn't that big, but Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ 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
```

Note how this tool manages Subversion tags as remote refs. Let's take a closer look with the Git plumbing command `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 doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ 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 fetches the tags directly into `refs/tags`, rather than treating them remote branches.

COMMITTING BACK TO SUBVERSION

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ 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
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ 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
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

PULLING IN NEW CHANGES

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ 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.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ 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.
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `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/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```
$ 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...
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ 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.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

GIT BRANCHING ISSUES

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ 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
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

SUBVERSION BRANCHING

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using git svn.

CREATING A NEW SVN BRANCH

To create a new branch in Subversion, you run `git svn branch [branch-name]`:

```
$ 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)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

SWITCHING ACTIVE BRANCHES

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, you can run

```
$ git branch opera remotes/origin/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say "Merge branch opera" instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you

can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the dcommit makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation – Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, `opera`) after you merge it into trunk.

SUBVERSION COMMANDS

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ 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
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't dcommited don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ 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
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `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)
```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two com-

mands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

GIT-SVN SUMMARY

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

Git і Mercurial

Всесвіт розподілених систем контролю версій значно більший, ніж просто Git. Насправді, існує багато інших систем, кожна зі своїм власним поглядом на процес розподіленого контролю версій. Окрім Git, найпопулярнішою з них є Mercurial, і ці дві системи мають багато спільного.

Хорошою новиною є те, що якщо ви надаєте перевагу Git, але вам доводиться працювати з проектами, код яких знаходиться в системі Mercurial, існує спосіб використання Git у якості клієнта для роботи з репозиторієм на Mercurial. Оскільки Git працює з серверами через концепцію "віддалених репозиторіїв" (remotes), не дивно, що цей міст реалізовано за допомогою своєрідного "помічника протоколу" (remote helper) для "віддалених репозиторіїв". Проект, який реалізує вищесказане, називається `git-remote-hg` і розміщений за адресою ***https://github.com/felipec/git-remote-hg***.

GIT-REMOTE-HG

Для початку, вам потрібно встановити `git-remote-hg`. Для цього скопіюйте файл до директорії, що є у вашому PATH, наприклад:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

…припускаючи, що `~/bin` включений у ваш $PATH. Git-remote-hg має ще одну залежність: бібліотеку `mercurial` для Python. Якщо у вас встановлено Python, просто виконайте:

```
$ pip install mercurial
```

(Якщо ж у вас не встановлено Python, перейдіть за посиланням ***https://www.python.org/*** і спочатку встановіть його.)

І останнє, що вам знадобиться, це клієнт Mercurial. Перейдіть за посиланням ***http://mercurial.selenic.com/*** і встановіть його, якщо ви цього ще не зробили.

Тепер ви готові побачити магію. Усе, що вам необхідно, це Mercurial-репозиторій, у який ви можете надсилати зміни. На щастя, так можна працювати з кожним Mercurial-репозиторієм, тому ми скористаємось репозиторієм "hello world", який використовується для вивчення Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

ОСНОВИ

Тепер, коли в нас є “серверний” репозиторій, ми можемо розглянути типові способи роботи з Mercurial. Як ви побачите згодом, ці дві системи дуже схожі, тому все повинно пройти гладко.

Як і при роботі з Git, спершу ми клонуємо репозиторій:

```
$ 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
```

Мабуть ви помітили, що для роботи з Mercurial-репозиторієм використовується стандартна команда `git clone`. Це тому, що git-remote-hg працює на доволі низькому рівні, і використовує механізм, подібний до HTTP/S протоколу в системі Git (помічники протоколу). Оскільки і Git, і Mercurial розраховані на те, що кожен клієнт має повну копію історії репозиторію, вищезгадана команда здійснює повне клонування, включно з усією історією проекту, і робить це досить швидко.

Команда `git log` показує два коміти, на останній з яких вказує безліч посилань. Насправді, не всі з них реально існують. Погляньмо, що знаходиться всередині директорії `.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 намагається нівелювати відмінності між Git та Mercurial, але "під капотом" він керує концептуальними перетвореннями між двома різними системами. У директорії `refs/hg` знаходяться посилання на об'єкти віддаленого репозиторія. Наприклад, `refs/hg/origin/branches/default` — це файл-посилання Git, який містить SHA-1, що починається з "ac7955c", який є комітом, на який вказує гілка `master`. Таким чином, директорія `refs/hg` — це щось схоже на `refs/remotes/origin`, але тут окремо зберігаються закладки та гілки.

Файл `notes/hg` — відправна точка для розуміння того, як git-remote-hg встановлює відповідність між хешами комітів у Git та ідентифікаторами змін у Mercurial. Погляньмо, що там:

```
$ 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
```

Отже, `refs/notes/hg` вказує на дерево, яке в базі об'єктів Git містить перелік інших об'єктів та їхніх імен. Команда `git ls-tree` виводить права доступу, тип, хеш та ім'я файлу для елементів дерева. Коли ми дістанемось першого елемента дерева, ми побачимо, що всередині знаходиться блоб з іменем "ac9117f" (SHA-1 хеш коміту, на який вказує гілка `master`), який містить "0a04b98" (ідентифікатор останньої зміни гілки `default` у Mercurial).

Хорошою новиною є те, що нам не потрібно турбуватися про все це. Типовий робочий процес не буде значно відрізнятися від роботи з віддаленим репозиторієм Git.

Є ще одна річ, яку ми повинні враховувати, перш ніж продовжувати: ігноровані файли. І Mercurial, і Git використовують для цього схожий механізм, але зберігати файл `.gitignore` в Mercurial-репозиторії — не найкраща ідея. На щастя, в Git є можливість ігнорувати файли, що

знаходяться в локальній копії репозиторія, а формат списку ігнорованих файлів в Mercurial сумісний з Git, тому вам достатньо скопіювати його:

```
$ cp .hgignore .git/info/exclude
```

Файл `.git/info/exclude` діє подібно до `.gitignore`, але не включається у коміт.

РОБОЧИЙ ПРОЦЕС

Нехай ми виконали певний обсяг роботи і зробили деякі коміти в гілку `master`, і тепер ви готові надіслати зміни до віддаленого репозиторію. Так виглядає наш репозиторій в цей момент:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/defa
* 65bb417 Create a standard "hello, world" program
```

Наша гілка `master` на два коміти попереду `origin/master`, але ці два коміти існують лише на нашій локальній машині. Погляньмо, раптом хтось інший зробив важливі зміни:

```
$ 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/branches/defa
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Оскільки ми використали `--all` ми бачимо посилання “notes”, які використовуються всередині git-remote-hg, але ми можемо не звертати уваги на них. Все інше — саме те, що ми очікували; `origin/master`

пішов далі на один коміт і наша історія розійшлася. На відміну від інших систем, з якими ми мали справу в цьому розділі, Mercurial вміє працювати зі злиттям, тому нам не потрібно робити жодних фокусів.

```
$ 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
```

Чудово. Ми запустили наші тести і всі вони пройшли, отже, ми готові ділитися нашими напрацюваннями з рештою команди:

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

І це все! Якщо ви поглянете на Mercurial-репозиторій, ви переконаєтесь, що відбулось саме те, що ми й очікували:

```
$ 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
```

Набір змін *2* був здійснений Mercurial'ом, а зміни *3* та *4* — за допомогою git-remote-hg шляхом надсилання комітів, зроблених з Git.

ГІЛКИ І ЗАКЛАДКИ

Git має лише один вид гілок: вказівник, який переміщується при комітах. У Mercurial цей вид вказівника називається "закладка", і вона поводить себе подібно до гілки в Git.

Поняття "гілка" в Mercurial більш складне. Гілка, в якій відбувається зміна, записується *всередині кожної зміни*, таким чином, вона завжди залишається в історії репозиторія. Ось приклад коміту, який зроблено в гілці `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
```

Зверніть увагу на рядок, що починається з "branch". Git не може насправді відтворити це (і не повинен; обидва типи гілок можуть бути представлені як Git-посилання), але git-remote-hg змушений розуміти цю різницю, оскільки це важливо для Mercurial.

Створення закладок у Mercurial настільки ж просте, як створення гілок у Git. У Git ми робимо наступне:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

Ось і все, що потрібно. А в Mercurial це виглядає так:

```
$ 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
```

Зверніть увагу на нову мітку `[featureA]` у п'ятій ревізії. Це працює так само, як гілки в Git, але з одним винятком: ви не можете видаляти закладки в Git (це обмеження помічників протоколу).

Ви можете працювати і з "повноцінними" Mercurial-гілками: просто розмістіть гілку в просторі імен `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
```

Ось як це виглядає в 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'
[...]
```

Ім'я гілки "permanent" було записано разом зі змінами під номером *7*.

З боку Git, робота з обома видами гілок однакова: просто переходите на гілку, робите коміт, отримуєте зміни, робите злиття і надсилаєте зміни (checkout, commit, fetch, merge, pull, і push) як завжди. Ще одна річ, про яку вам потрібно знати: Mercurial не підтримує перезапис історії, лише додавання. Ось як наш Mercurial-репозиторій виглядає після інтерактивної зміни історії та примусового надсилання змін:

```
$ 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
```

Були створені зміни *8*, *9* та *10* і тепер вони належать до гілки `permanent`, але старі зміни досі там. Це може **дуже** спантеличити ваших колег, які використовують Mercurial, тому краще уникати цього.

ПІДСУМОК ПО **MERCURIAL**

Git і Mercurial достатньо подібні для безболісної роботи один з одним. Якщо ви будете уникати змін вже опублікованої історії (що, загалом, рекомендовано), ви навіть не помітите, що працюєте в Mercurial.

Git і Perforce

Perforce — дуже популярна система контролю версій у корпоративному середовищі. Він з'явився у 1995 році, що робить його найстарішою системою контролю версій з тих, що розглядаються у цьому розділі. Perforce розроблений з обмеженнями тих часів; він передбачає постійне з'єднання з центральним сервером, а локально зберігається лише одна версія файлів. Насправді, його можливості та обмеження добре підходять для вирішення специфічного кола задач, та існує досить багато проектів, які використовують Perforce, але де Git працював би значно краще.

Існують два варіанти сумісного використання Perforce і Git. Перший, який ми розглянемо, — міст "Git Fusion" від розробників Perforce, який дозволить вам виставляти піддерева (subtrees) вашого Perforce-депо (Perforce depot) як Git репозиторії з можливістю читання-запису. Другий — git-p4 — клієнтський міст, який дозволяє вам використовувати Git як

клієнт Perforce без необхідності здійснювати будь-яке переналаштування сервера Perforce.

GIT FUSION

Perforce забезпечує продукт, який називається Git Fusion (доступний за посиланням ***http://www.perforce.com/git-fusion***), і синхронізує сервер Perforce з репозиторієм Git на стороні сервера.

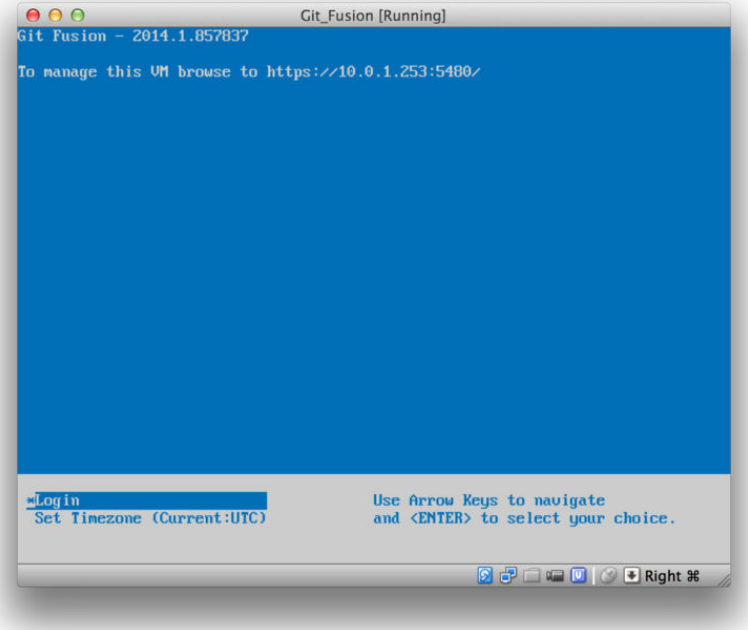
C

Для наших прикладів ми використаємо найпростіший метод встановлення Git Fusion, який полягає у завантаженні віртуальної машини, на якій виконується Perforce демон і Git Fusion. Ви можете отримати образ віртуальної машини за посиланням ***http://www.perforce.com/downloads/Perforce/20-User***, і коли завантаження буде завершено, імпортувати його у ваше улюблене програмне забезпечення для віртуалізації (ми використаємо VirtualBox).

Під час першого запуску віртуальної машини вам потрібно налаштувати паролі для трьох Linux-користувачів (`root`, `perforce` та `git`) і ввести ім'я хоста, яке буде відрізняти це встановлення від інших в одній мережі. Коли все буде готово, ви побачите наступне:

ЗОБРАЖЕННЯ 9-1

Екран віртуальної машини Git Fusion.



Вам потрібно занотувати цю IP адресу, пізніше ми будемо її використовувати. Далі ми створимо користувача Perforce. Виберіть знизу опцію “Login” та натисніть Enter (або скористайтесь SSH), і увійдіть як root. Потім використайте ці команди для створення користувача:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

Перша команда відкриє редактор VI для редагування користувача, але ви можете прийняти типові налаштування і ввести :wq та натиснути Enter. Друга команда двічі попросить вас ввести пароль. Це все, що нам потрібно виконати в оболонці системи, тому завершіть поточну сесію.

Наступне, що вам потрібно зробити, це заборонити Git перевіряти SSL сертифікати. Образ Git Fusion розповсюджується з сертифікатом, але він для домену, що не відповідає IP адресі вашої віртуальної

машини, тому Git відхилить HTTPS-з'єднання. Якщо це встановлення буде використовуватись на постійній основі, зверніться до документації Git Fusion, щоб встановити інший сертифікат; а для нашої навчальної мети підійде наступне:

```
$ export GIT_SSL_NO_VERIFY=true
```

Тепер ми можемо переконатися, що все працює.

```
$ 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.
```

Образ віртуальної машини розповсюджується з тестовим проектом, який ви можете клонувати. Тут ми клонуємо проект через HTTPS-з'єднання, з користувачем `john`, якого ми створили вище; Git запитає пароль для цього з'єднання, але кеш паролів дозволить нам пропускати цей крок для будь-яких наступних запитів.

Git Fusion

Після встановлення Git Fusion ви, мабуть, захочете налаштувати його. Насправді, це відносно легко зробити з використанням вашого улюбленого клієнта Perforce; просто відобразіть директорію `//.git-fusion` на сервері Perforce у ваш робочий простір. Структура файлів повинна бути подібною до:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
```

```
|       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

Директорія `objects` використовується Git Fusion для відображення об'єктів Perforce у об'єкти Git і навпаки, вам не варто тут нічого змінювати. Всередині цієї директорії знаходиться глобальний файл `p4gf_config`, а також по одному для кожного репозиторія - це файли конфігурації, які визначають поведінку Git Fusion. Погляньмо на файл, що знаходиться в корені:

```
[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
```

Ми не будемо детально розглядати ці опції, але зверніть увагу на те, що це звичайний INI-файл, подібний до файлів конфігурації, які використовує Git. Цей файл задає глобальні опції, які можуть бути перевизначені специфічними для кожного репозиторія файлами конфігурації, такими, як `repos/Talkhouse/p4gf_config`. Якщо ви відкриєте цей файл, то побачите секцію `[@repo]` з деякими

налаштуваннями, які відрізняються від типових глобальних налаштувань. Ви також побачите секції, які виглядають подібно до цього:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Вони задають відповідність між гілками Perforce та гілками Git. Назви таких секцій можуть бути довільними, поки ці назви залишаються унікальними. Команда `git-branch-name` дозволяє вам конвертувати недоладний для Git шлях всередині депо, щоб у Git мати більш дружелюбне ім'я. Параметр `view` керує відображенням файлів Perforce у репозиторії Git з використанням стандартного синтаксису відображення видів. Більш ніж одне відображення можна визначити як у прикладі нижче:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

Таким чином, якщо ваше нормальне відображення включає зміни структури директорій, ви можете відтворити це за допомогою Git-репозиторія.

Останній файл, про який ми поговоримо, це `users/p4gf_usermap`, який відображає користувачів Perforce у користувачів Git, і який, можливо, вам і не знадобиться.

Коли Perforce конвертує набір змін (changeset) у Git коміт, типова поведінка Git Fusion — пошук користувача Perforce і використання адреси його електронної пошти та повного імені для заповнення поля "автор коміту" в Git. При конвертуванні у зворотньому напрямку, типова поведінка — пошук користувача Perforce з адресою електронної пошти в полі "автор коміту" у Git, та застосовує набір змін від імені цього користувача (з використанням прав доступу). У більшості випадків така поведінка є нормальною, але розгляньмо наступний файл відповідностей:

```
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"
```

Кожний рядок має формат `<user> <email> "<full name>"` і створює відповідність для одного користувача. Перші два рядки відображають дві різні адреси електронної пошти для одного користувача Perforce. Це може бути корисним, якщо ви створили коміти у Git з різних адрес електронної пошти (або змінили свою електронну пошту), але хочете їх відобразити на одного й того ж користувача Perforce. При створенні комітів у Git з набору змін Perforce, перший рядок, який відповідає користувачу Perforce, використовується Git як інформація про авторство.

Останні два рядки маскують справжні імена та адреси електронної пошти Боба і Джо у комітах Git при їх створенні. Це корисно, якщо ви хочете вивести ваш внутрішній проект в open-source, але не хочете публікувати свій каталог співробітників для всього світу. Зауважте, що адреси електронної пошти і повні імена повинні бути унікальними, якщо ви не хочете, щоб всі коміти Git належали одному фіктивному автору.

Perforce Git Fusion — це двосторонній міст між системами контролю версій Perforce і Git. Погляньмо, як виглядає робота з боку Git. Нехай ми налаштували відображення проекту “Jam” з використанням файла конфігурації, як показано вище, який ми можемо клонувати таким чином:

```
$ 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.
[...]
```

Якщо ви робите клонування вперше, це може тривати деякий час. При цьому Git Fusion конвертує усі можливі набори змін з історії Perforce у коміти Git. Це відбувається локально на сервері, тому ця операція здійснюється відносно швидко, але якщо у вас багато історії, цей процес триватиме деякий час. Наступні отримання змін вимагають лише інкрементних перетворень, таким чином, швидкість роботи буде порівняна зі швидкістю роботи з Git.

Як ви бачите, наш репозиторій виглядає як і будь-який інший Git-репозиторій, з яким ви могли працювати. Тут є три гілки і Git доречно створив локальну гілку `master`, яка відслідковує `origin/master`. Попрацюймо трохи і створімо кілька нових комітів:

```
# ...
$ 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.
[...]
```

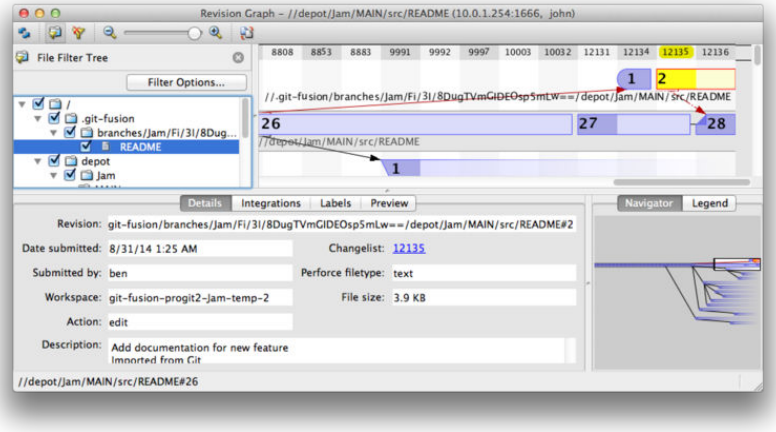
Ми маємо два нових коміти. Погляньмо, чи вносив зміни хтось інший:

```
$ 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.
[...]
```

Схоже, хтось таки вносив! Ви не дізнаєтесь цього з виводу команди `git fetch`, але коміт `6afeb15` був створений за допомогою клієнта Perforce. Він виглядає як інший звичайний коміт з точки зору Git, для чого й створений Git Fusion. Погляньмо, як сервер Perforce працює з комітом злиття:

```
$ 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
```

З боку Git це працює. Погляньмо на історію файла `README` з боку Perforce, скориставшись графом ревізій p4v:



ЗОБРАЖЕННЯ 9-2

Граф ревізій Perforce після надсилання даних з Git.

Якщо ви ніколи такого не бачили, це може вас спантеличити, але його концепція схожа з графічним переглядачем для історії Git. Ми дивимось на історію файла `README`, тому дерево директорії ліворуч вгорі показує лише цей файл, коли він змінюється в різноманітних гілках. Праворуч вгорі ми бачимо візуалізацію залежностей різних версій файла і повну версію цього графу внизу праворуч. Решта вікна залишена для детального відображення вибраної ревізії (у цьому випадку, другої).

Одна річ, яку потрібно зауважити, граф виглядає так само, як і в історії Git. У Perforce не було іменованої гілки для зберігання комітів 1 і 2, тому він створив “анонімну” гілку в директорії `.git-fusion` для їх збереження. Perforce буде діяти схожим чином і з іменованими гілками Git, які не відповідають іменам гілок Perforce (але ви можете задати для них відповідності у файлі конфігурації Perforce).

Більшість з цього відбувається всередині Git Fusion, але в результаті один член команди може користуватися Git, а інший — Perforce, і жоден з них не буде знати про вибір іншого.

Git-Fusion

Якщо у вас є (або ви можете отримати) доступ до вашого сервера Perforce, Git Fusion стане чудовим способом налагодження співпраці між Git та Perforce. Звичайно, для цього потрібно налаштувати конфігурацію, але загалом цей процес не є занадто складним. Це одна з небагатьох секцій цього розділу, де не з’являлись попередження про використання всього функціоналу Git. Але це не означає, що Perforce зможе виконати все, що ви захочете - якщо ви

спробуєте перезаписати історію, яка вже надіслана на сервер, Git Fusion відхилить такі зміни - але Git Fusion намагається зробити все можливе, щоб ви не відчували незручностей. Ви можете навіть використовувати підмодулі Git (хоча вони й будуть виглядати дивно для користувачів Perforce), і зливати гілки (з боку Perforce це буде виглядати як інтеграція).

Якщо ви не зможете вмовити адміністратора вашого серверу налаштувати Git Fusion, все одно існує спосіб використання цих інструментів разом.

GIT-P4

Git-p4 — це двосторонній міст між Git та Perforce. Він працює повністю всередині вашого репозиторію Git, тому вам взагалі не потрібно мати доступ до сервера Perforce (звичайно, вам знадобляться логін і пароль). Git-p4 не такий гнучкий та функціональний, як Git Fusion, але він дозволяє здійснювати більшість з того, що вам буде необхідно, без втручання у середовище сервера.

Для роботи з git-p4 вам знадобиться виконуваний файл p4, доступний у вашому PATH. На момент написання книги, він вільно доступний за посиланням *http://www.perforce.com/downloads/Perforce/20-User*.

C

Для навчальної мети, ми запустимо сервер Perforce з віртуальної машини Git Fusion OVA, як було показано вище, але ми будемо оминати сервер Git Fusion і напряму звертатися до системи контролю версій Perforce.

Для використання клієнта командного рядка p4 (від якого залежить git-p4), вам буде потрібно прописати кілька змінних середовища:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

C

Як і зазвичай при роботі з Git, перша команда — клонування:

```
$ 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
```

Ми створили те, що у термінах Git називається “поверхнева” (shallow) копія; лише найостанніша ревізія Perforce імпортована у Git; пам’ятайте, що Perforce не призначений віддавати кожну ревізію кожному користувачу. Цього достатньо для використання Git як клієнта Perforce, але цього недостатньо для інших задач.

Коли клонування завершиться, ми будемо мати повнофункціональний репозиторій Git:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from the st
```

Зверніть увагу на наявність віддаленого репозиторію “p4” для сервера Perforce, але все інше виглядає стандартно для клонованого репозиторія. Насправді, це нас трохи вводить в оману; виявляється, там немає ніякого віддаленого репозиторія.

```
$ git remote -v
```

Ніяких віддалених репозиторіїв не існує взагалі. Git-p4 створив кілька посилань для представлення стану сервера, і вони схожі на посилання віддаленого репозиторія для `git log`, але вони не обслуговуються Git і ви не можете надсилати зміни у них.

Отже, почнімо працювати. Припустимо, ви виконали певний обсяг роботи по дуже важливій задачі, і ви готові показати свою роботу решті команди.

```
$ 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
```

Ми зробити два нових коміти, які вже готові до відправлення на сервер Perforce. Погляньмо, чи хтось інший сьогодні робив зміни:

```
$ 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
```

Схоже, що так, і гілки `master` та `p4/master` розійшлися. Система галудження Perforce *абсолютно* не схожа на Git, тому відправлення комітів злиття не має жодного сенсу. Git-p4 рекомендує перебазувати ваші коміти, і навіть надає спеціальну команду для цього:

```
$ 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(-)
```

Ви, можливо, скажете, що `git p4 rebase` — це лише скорочення для `git p4 sync` з наступним `git rebase p4/master`. Насправді, ця команда трохи розумніша, особливо при роботі з багатьма гілками, але це вірна здогадка.

Тепер наша історія змін знову лінійна, і ми готові надіслати наші зміни до Perforce. Команда `git p4 submit` буде намагатися створити нову ревізію Perforce для кожного коміту Git між `p4/master` і `master`. Її запуск відкриє ваш улюблений редактор, і вміст файлу буде схожим на це:

```
# 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   2014-08-31
@@ -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>
```

Це практично ті ж самі дані, які б ви побачили, запустивши `p4 submit`, за винятком кількох рядків у кінці, які були доречно додані git-p4. Git-p4 намагається враховувати ваші індивідуальні налаштування для Git і Perforce, коли потрібно надати ім'я для коміту або набору змін, але у деяких випадках ви захочете змінити їх. Наприклад, якщо коміт Git, який ви імпортуєте, був створений користувачем, який не має облікового запису Perforce, можливо, ви все одно захочете, щоб кінцевий набір змін виглядав так, ніби він був створений тим користувачем (а не вами).

Git-p4 доречно імпортував повідомлення з коміту Git як вміст для набору змін Perforce, тому все, що нам потрібно зробити, це зберегти і

вийти, двічі (по одному разу для кожного коміта). У результаті, вивід командного рядка буде виглядати так:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens
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
```

Результат виглядає так, ніби ми щойно виконали `git push`, і це дуже близько до того, що насправді відбулося.

Зверніть увагу, що під час цього процесу кожен коміт Git перетворюється у набір змін Perforce; якщо ви хочете зварити їх (squash) в один набір змін, ви можете скористатися інтерактивним перебазуванням перед запуском `git p4 submit`. Також зазначте, що хеші SHA-1 усіх комітів, перетворених у набори змін, також змінилися; це сталося тому, що git-p4 додає рядок у кінець кожного коміта, який конвертує:

```
$ 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]
```

А що станеться, якщо ви спробуєте відправити коміт злиття? Спробуймо. Припустимо, у нас є така ситуація:

```
$ 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
```

Історія змін Git розійшлась з історією Perforce після `775a46f`. У Git є два коміти, потім коміт злиття з Perforce, а потім інший коміт. Ми спробуємо надіслати це у набір змін на боці Perforce. Погляньмо, що трапиться, якщо ми спробуємо надіслати зараз:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

Прапорець `-n` — це скорочення для `--dry-run`, який намагається повідомити, що б сталося, якби б відправлення відбулось насправді. У цьому випадку, схоже, ми створили три набори змін Perforce, які відповідають трьом звичайним комітам (крім коміту злиття), які ще не

існують на сервері Perforce. Це звучить саме так, як ми й хотіли, погляньмо на вивід:

```
$ 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
```

Наша історія змін стала лінійною, ніби ми перебазувалися перед відправленням (що насправді й відбулося). Це означає, що ви можете вільно створювати, працювати, викидати, і зливати гілки на стороні Git без страху, що ваша історія змін якимось чином стане несумісною з Perforce. Якщо ви можете перебазувати зміни, ви можете відправити їх на сервер Perforce.

C

Якщо ваш проект Perforce має декілька гілок, не турбуйтеся; git-p4 може впоратися з цим так, що ви не відчуєте різниці у порівнянні з Git. Припустімо, ваш репозиторій Perforce має таку структуру:

```
//depot
  └── project
      ├── main
      └── dev
```

І нехай ви маєте гілку dev, яка налаштована наступним чином:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 може автоматично виявляти такі ситуації і виконувати потрібні дії:

```
$ 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
```

Зверніть увагу на “@all” в шляху; це говорить git-p4 клонувати не лише останній набір змін для цього піддерева, а й усі набори змін, яких стосуються ці шляхи. Це ближче до концепції клонування Git, але якщо ви працюєте над проектом з довгою історією, це може зайняти деякий час.

Прапорець `--detect-branches` говорить git-p4 використовувати налаштування гілок Perforce для відображення на посилання Git. Якщо таких відображень на сервері Perforce немає (що цілком коректно для Perforce), ви можете вказати git-p4 ці гілки вручну, і отримаєте такий самий результат:

```
$ 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 .
```

Задавши конфігураційну змінну `git-p4.branchList` рівною `main:dev`, ми говоримо git-p4, що і “main” і “dev” є гілками, а друга є нащадком першої.

Якщо ми тепер виконаємо `git checkout -b dev p4/project/dev` і зробимо кілька комітів, git-p4 достатньо розумний, щоб зрозуміти, у яку гілку надсилати зміни при виконанні команди `git p4 submit`. На жаль, git-p4 не може використовувати декілька гілок у поверхневих копіях; якщо у вас величезний проект і ви хочете працювати з більш ніж однією гілкою, вам доведеться виконувати `git p4 clone` для кожної гілки, у яку ви хочете надіслати зміни.

Для створення чи інтеграції гілок ви повинні використовувати клієнт Perforce. Git-p4 може лише синхронізувати і відправляти зміни у гілки, які вже існують, і може це робити лише з одним лінійним набором змін за раз. Якщо ви зіллєте дві гілки у Git та спробуєте надіслати новий

набір змін, все, що збережеться — це купа змін у файлах; метадані про гілки, які були змінені при інтеграції, будуть втрачені.

ПІДСУМОК ПО GIT І PERFORCE

Git-p4 робить можливим використання Git з сервером Perforce, і робить це досить добре. Однак, важливо пам'ятати, що Perforce все одно залишається джерелом даних, і ви використовуєте Git лише для локальної роботи. Будьте дуже обережні при публікації комітів Git; якщо ви маєте віддалений репозиторій, який використовують інші люди, не надсилайте жодних комітів, які ще не надіслані на сервер Perforce.

Якщо ви хочете вільно змішувати Perforce і Git як клієнти для контролю версій, і ви можете переконати адміністратора сервера встановити їх, Git Fusion зробить використання Git у якості клієнта контролю версій для сервера Perforce повноцінним.

Git and TFS

Git is becoming popular with Windows developers, and if you're writing code on Windows, there's a good chance you're using Microsoft's Team Foundation Server (TFS). TFS is a collaboration suite that includes defect and work-item tracking, process support for Scrum and others, code review, and version control. There's a bit of confusion ahead: **TFS** is the server, which supports controlling source code using both Git and their own custom VCS, which they've dubbed **TFVC** (Team Foundation Version Control). Git support is a somewhat new feature for TFS (shipping with the 2013 version), so all of the tools that predate that refer to the version-control portion as "TFS", even though they're mostly working with TFVC.

If you find yourself on a team that's using TFVC but you'd rather use Git as your version-control client, there's a project for you.

WHICH TOOL

In fact, there are two: git-tf and git-tfs.

Git-tfs (found at ***https://github.com/git-tfs/git-tfs***) is a .NET project, and (as of this writing) it only runs on Windows. To work with Git repositories, it uses the .NET bindings for libgit2, a library-oriented implementation of Git which is highly performant and allows a lot of flexibility with the guts of a Git repository. Libgit2 is not a complete implementation of Git, so to cover the difference git-tfs will actually call the command-line Git client for some operations, so there are no artificial limits on what it can do with Git repositories. Its support of

TFVC features is very mature, since it uses the Visual Studio assemblies for operations with servers. This does mean you'll need access to those assemblies, which means you need to install a recent version of Visual Studio (any edition since version 2010, including Express since version 2012), or the Visual Studio SDK.

Git-tf (whose home is at ***https://gittf.codeplex.com***) is a Java project, and as such runs on any computer with a Java runtime environment. It interfaces with Git repositories through JGit (a JVM implementation of Git), which means it has virtually no limitations in terms of Git functions. However, its support for TFVC is limited as compared to git-tfs – it does not support branches, for instance.

So each tool has pros and cons, and there are plenty of situations that favor one over the other. We'll cover the basic usage of both of them in this book.

You'll need access to a TFVC-based repository to follow along with these instructions. These aren't as plentiful in the wild as Git or Subversion repositories, so you may need to create one of your own. Codeplex (***https://www.codeplex.com***) or Visual Studio Online (***http://www.visualstudio.com***) are both good choices for this.

GETTING STARTED: `GIT-TF`

The first thing you do, just as with any Git project, is clone. Here's what that looks like with `git-tf`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

The first argument is the URL of a TFVC collection, the second is of the form `$/project/branch`, and the third is the path to the local Git repository that is to be created (this last one is optional). Git-tf can only work with one branch at a time; if you want to make checkins on a different TFVC branch, you'll have to make a new clone from that branch.

This creates a fully functional Git repository:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

This is called a *shallow* clone, meaning that only the latest changeset has been downloaded. TFVC isn't designed for each client to have a full copy of the history, so git-tf defaults to only getting the latest version, which is much faster.

If you have some time, it's probably worth it to clone the entire project history, using the `--deep` option:

```
$ 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
```

Notice the tags with names like TFS_C35189; this is a feature that helps you know which Git commits are associated with TFVC changesets. This is a nice way to represent it, since you can see with a simple log command which of your commits is associated with a snapshot that also exists in TFVC. They aren't necessary (and in fact you can turn them off with `git config git-tf.tag false`) – git-tf keeps the real commit-changeset mappings in the `.git/git-tf` file.

GETTING STARTED: `GIT-TFS`

Git-tfs cloning behaves a bit differently. Observe:

```
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
```

Notice the `--with-branches` flag. Git-tfs is capable of mapping TFVC branches to Git branches, and this flag tells it to set up a local Git branch for every TFVC branch. This is highly recommended if you've ever branched or

merged in TFS, but it won't work with a server older than TFS 2010 – before that release, "branches" were just folders, so git-tfs can't tell them from regular folders.

Let's take a look at the resulting Git repository:

```
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/Trunk;C16
```

There are two local branches, `master` and `featureA`, which represent the initial starting point of the clone (`Trunk` in TFVC) and a child branch (`featureA` in TFVC). You can also see that the `tfs` "remote" has a couple of refs too: `default` and `featureA`, which represent TFVC branches. Git-tfs maps the branch you cloned from to `tfs/default`, and others get their own names.

Another thing to notice is the `git-tfs-id:` lines in the commit messages. Instead of tags, git-tfs uses these markers to relate TFVC changesets to Git commits. This has the implication that your Git commits will have a different SHA-1 hash before and after they have been pushed to TFVC.

GIT-TF[S] WORKFLOW

Regardless of which tool you're using, you should set a couple of Git configuration values to avoid running into issues.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

The obvious next thing you're going to want to do is work on the project. TFVC and TFS have several features that may add complexity to your workflow:

1. Feature branches that aren't represented in TFVC add a bit of complexity. This has to do with the **very** different ways that TFVC and Git represent branches.

2. Be aware that TFVC allows users to “checkout” files from the server, locking them so nobody else can edit them. This obviously won’t stop you from editing them in your local repository, but it could get in the way when it comes time to push your changes up to the TFVC server.
3. TFS has the concept of “gated” checkins, where a TFS build-test cycle has to complete successfully before the checkin is allowed. This uses the “shelve” function in TFVC, which we don’t cover in detail here. You can fake this in a manual fashion with git-tf, and git-tfs provides the `checkintool` command which is gate-aware.

In the interest of brevity, what we’ll cover here is the happy path, which sidesteps or avoids most of these issues.

WORKFLOW: `GIT-TF`

Let’s say you’ve done some work, made a couple of Git commits on `master`, and you’re ready to share your progress on the TFVC server. Here’s our Git repository:

```
$ 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
```

We want to take the snapshot that’s in the 4178a82 commit and push it up to the TFVC server. First things first: let’s see if any of our teammates did anything since we last connected:

```
$ 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
```

Looks like someone else is working, too, and now we have divergent history. This is where Git shines, but we have two choices of how to proceed:

1. Making a merge commit feels natural as a Git user (after all, that's what `git pull` does), and git-tf can do this for you with a simple `git tf pull`. Be aware, however, that TFVC doesn't think this way, and if you push merge commits your history will start to look different on both sides, which can be confusing. However, if you plan on submitting all of your changes as one changeset, this is probably the easiest choice.
2. Rebasing makes our commit history linear, which means we have the option of converting each of our Git commits into a TFVC changeset. Since this leaves the most options open, we recommend you do it this way; git-tf even makes it easy for you with `git tf pull --rebase`.

The choice is yours. For this example, we'll be rebasing:

```
$ 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
```

Now we're ready to make a checkin to the TFVC server. Git-tf gives you the choice of making a single changeset that represents all the changes since the last one (`--shallow`, which is the default) and creating a new changeset for each Git commit (`--deep`). For this example, we'll just create one changeset:

```
$ 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
```

There's a new `TFS_C35348` tag, indicating that TFVC is storing the exact same snapshot as the `5a0e25e` commit. It's important to note that not every Git commit needs to have an exact counterpart in TFVC; the `6eb3eb5` commit, for example, doesn't exist anywhere on the server.

That's the main workflow. There are a couple of other considerations you'll want to keep in mind:

- There is no branching. Git-tf can only create Git repositories from one TFVC branch at a time.
- Collaborate using either TFVC or Git, but not both. Different git-tf clones of the same TFVC repository may have different commit SHA-1 hashes, which will cause no end of headaches.
- If your team's workflow includes collaborating in Git and syncing periodically with TFVC, only connect to TFVC with one of the Git repositories.

WORKFLOW: `GIT-TFS`

Let's walk through the same scenario using git-tfs. Here are the new commits we've made to the `master` branch in our Git repository:

```
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
```

Now let's see if anyone else has done work while we were hacking away:

```
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
```

Yes, it turns out our coworker has added a new TFVC changeset, which shows up as the new `aea74a0` commit, and the `tfs/default` remote branch has moved.

As with git-tf, we have two fundamental options for how to resolve this divergent history:

1. Rebase to preserve a linear history.
2. Merge to preserve what actually happened.

In this case, we're going to do a "deep" checkin, where every Git commit becomes a TFVC changeset, so we want to rebase.

```
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
```

Now we're ready to complete our contribution by checking in our code to the TFVC server. We'll use the `rcheckin` command here to create a TFVC changeset for each Git commit in the path from HEAD to the first `tfs` remote branch found (the `checkin` command would only create one changeset, sort of like squashing Git 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'
```

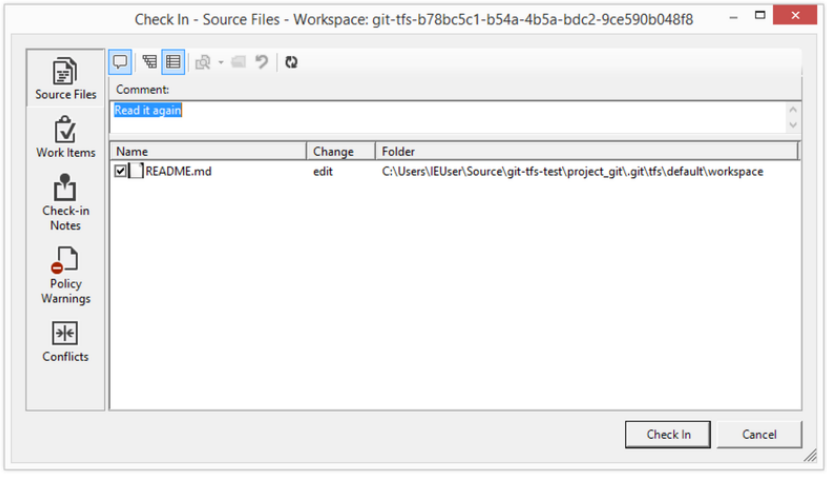
```
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-comm
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-comm
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
```

Notice how after every successful checkin to the TFVC server, git-tfs is rebasing the remaining work onto what it just did. That's because it's adding the `git-tfs-id` field to the bottom of the commit messages, which changes the SHA-1 hashes. This is exactly as designed, and there's nothing to worry about, but you should be aware that it's happening, especially if you're sharing Git commits with others.

TFS has many features that integrate with its version control system, such as work items, designated reviewers, gated checkins, and so on. It can be cumbersome to work with these features using only a command-line tool, but fortunately git-tfs lets you launch a graphical checkin tool very easily:

```
PS> git tfs checkintool
PS> git tfs ct
```

It looks a bit like this:



ЗОБРАЖЕННЯ 9-3

The git-tfs checkin tool.

This will look familiar to TFS users, as it's the same dialog that's launched from within Visual Studio.

Git-tfs also lets you control TFVC branches from your Git repository. As an example, let's create one:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 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
```

Creating a branch in TFVC means adding a changeset where that branch now exists, and this is projected as a Git commit. Note also that git-tfs **created** the `tfs/featureBee` remote branch, but `HEAD` is still pointing to `master`. If you want to work on the newly-minted branch, you'll want to base your new commits on the `1d54865` commit, perhaps by creating a topic branch from that commit.

GIT AND TFS SUMMARY

Git-tf and Git-tfs are both great tools for interfacing with a TFVC server. They allow you to use the power of Git locally, avoid constantly having to round-trip to the central TFVC server, and make your life as a developer much easier, without forcing your entire team to migrate to Git. If you're working on Windows (which is likely if your team is using TFS), you'll probably want to use git-tfs, since its feature set is more complete, but if you're working on another platform, you'll be using git-tf, which is more limited. As with most of the tools in this chapter, you should choose one of these version-control systems to be canonical, and use the other one in a subordinate fashion – either Git or TFVC should be the center of collaboration, but not both.

Міграція на Git

Якщо ваш код вже зберігається в іншій системі контролю версій, але ви вирішили почати використовувати Git, вам необхідно так чи інакше здійснити міграцію проекту. У цій секції описано деякі варіанти імпорту для поширених систем, а потім показано, як розробити власні варіанти імпорту. Ви дізнаєтесь як імпортувати дані з деякий найбільших професійних систем контролю версій, оскільки вони використовуються більшістю розробників та для них легко знайти якісні інструменти міграції.

Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log --xml | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

That generates the log output in XML format, then keeps only the lines with author information, discards duplicates, strips out the XML tags. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your users.txt file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command. This makes your `import` command look like this:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this

```
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
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Not only does the Author field look a lot better, but the `git-svn-id` is no longer there, either.

You should also do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

This takes the references that were remote branches that started with `remotes/origin/tags/` and makes them real (lightweight) tags.

Next, move the rest of the references under `refs/remotes` to be local branches:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Now all the old branches are real Git branches and all the old tags are real Git tags. The last thing to do is add your new Git server as a remote and push to it. Here is an example of adding your server as a remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Because you want all your branches and tags to go up, you can now run this:

```
$ git push origin --all
```

All your branches and tags should be on your new Git server in a nice, clean import.

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool called "hg-fast-export", which you'll need a copy of:

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

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a `bash` shell:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

This will take a few seconds, depending on how long your project's history is, and afterwards the `/tmp/authors` file will look something like this:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by adding `={new name and email address}` at the end of every line we want to change, and removing the lines for any usernames that we want to leave alone. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```
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>
```

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The `-r` flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the `-A` flag tells it where to find the author-mapping file. The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```
$ /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
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
      commits:        22208 (         0 duplicates          0 deltas of          0
      tags   :            0 (         0 duplicates          0 deltas of          0
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
```

That's pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you're ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Perforce

The next system you'll look at importing from is Perforce. As we discussed above, there are two ways to let Git and Perforce talk to each other: git-p4 and Perforce Git Fusion.

PERFORCE GIT FUSION

Git Fusion makes this process fairly painless. Just configure your project settings, user mappings, and branches using a configuration file (as discussed in **"Git Fusion"**), and clone the repository. Git Fusion leaves you with what looks like a native Git repository, which is then ready to push to a native Git host if you desire. You could even use Perforce as your Git host if you like.

GIT-P4

Git-p4 can also act as an import tool. As an example, we'll import the Jam project from the Perforce Public Depot. To set up your client, you must export the P4PORT environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

> In order to follow along, you'll need a Perforce depot to connect with. We'll be using the public depot at public.perforce.com for our examples, but you can use any depot you have access to.

Run the `git p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ 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%)
```

This particular project has only one branch, but if you have branches that are configured with branch views (or just a set of directories), you can use the `--detect-branches` flag to `git p4 clone` to import all the project's branches as well. See "Галуження" for a bit more detail on this.

At this point you're almost done. If you go to the `p4import` directory and run `git log`, you can see your imported work:

```
$ 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]
```

You can see that `git-p4` has left an identifier in each commit message. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so – before you start doing work on the new repository. You can use `git filter-branch` to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

If you run `git log`, you can see that all the SHA-1 checksums for the commits have changed, but the `git-p4` strings are no longer in the commit messages:

```
$ 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).
```

Your import is ready to push up to your new Git server.

TFS

If your team is converting their source control from TFVC to Git, you'll want the highest-fidelity conversion you can get. This means that, while we covered both git-tfs and git-tf for the interop section, we'll only be covering git-tfs for this part, because git-tfs supports branches, and this is prohibitively difficult using git-tf.

> This is a one-way conversion. The resulting Git repository won't be able to connect with the original TFVC project.

The first thing to do is map usernames. TFVC is fairly liberal with what goes into the author field for changesets, but Git wants a human-readable name and email address. You can get this information from the `tf` command-line client, like so:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

This grabs all of the changesets in the history of the project and put it in the AUTHORS_TMP file that we will process to extract the data of the *User* column (the 2nd one). Open the file and find at which characters start and end the column and replace, in the following command-line, the parameters `11-20` of the `cut` command with the ones found:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

The `cut` command keeps only the characters between 11 and 20 from each line. The `tail` command skips the first two lines, which are field headers and ASCII-art underlines. The result of all of this is piped to `uniq` to eliminate duplicates, and saved to a file named AUTHORS. The next step is manual; in order for git-tfs to make effective use of this file, each line must be in this format:

```
DOMAIN\username = User Name <email@address.com>
```

The portion on the left is the "User" field from TFVC, and the portion on the right side of the equals sign is the user name that will be used for Git commits.

Once you have this file, the next thing to do is make a full clone of the TFVC project you're interested in:

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.co
```

Next you'll want to clean the `git-tfs-id` sections from the bottom of the commit messages. The following command will do that:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' -- --all
```

That uses the `sed` command from the Git-bash environment to replace any line starting with "git-tfs-id:" with emptiness, which Git will then ignore.

Once that's all done, you're ready to add a new remote, push all your branches up, and have your team start working from Git.

A Custom Importer

If your system isn't one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write

the raw objects (see **Chapter 10** for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As we did in **"Приклад політики користування виконуваної Git-ом"**, we'll write this in Ruby, because it's what we generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines – git fast-import is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
last_mark = nil

# loop through the directories
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
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. “Mark” is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You’ll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you’ll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
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
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# print the import information
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
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word data, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, fast-import can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/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
```

```ruby
# Loop through the directories
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
```

If you run this script, you'll get content that looks something like this:

```
$ 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
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your 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 attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4 attempts)
      commits:            4 (         1 duplicates          0 deltas of          0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0 attempts)
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
---------------------------------------------------------------------
```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```
$ 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
```

There you go – a nice, clean Git repository. It's important to note that nothing is checked out – you don't have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

Підсумок

Ви повинні відчувати себе вільно при використанні Git як клієнта для інших систем контролю версій, чи імпортуючи будь-який із вже створених репозиторіїв у систему Git без втрати даних. Далі ми розглянемо внутрішню механіку системи Git, відтак ви зможете контролювати кожен байт даних, якщо це буде необхідно.

10 Git Internals

You may have skipped to this chapter from a previous chapter, or you may have gotten here after reading the rest of the book – in either case, this is where we'll go over the inner workings and implementation of Git. We found that learning this information was fundamentally important to understanding how useful and powerful Git is, but others have argued to us that it can be confusing and unnecessarily complex for beginners. Thus, we've made this discussion the last chapter in the book so you could read it early or later in your learning process. We leave it up to you to decide.

Now that you're here, let's get started. First, if it isn't yet clear, Git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it. You'll learn more about what this means in a bit.

In the early days of Git (mostly pre 1.5), the user interface was much more complex because it emphasized this filesystem rather than a polished VCS. In the last few years, the UI has been refined until it's as clean and easy to use as any system out there; but often, the stereotype lingers about the early Git UI that was complex and difficult to learn.

The content-addressable filesystem layer is amazingly cool, so we'll cover that first in this chapter; then, you'll learn about the transport mechanisms and the repository maintenance tasks that you may eventually have to deal with.

Сантехніка та порцеляна

Ця книга охоплює використання Git за допомогою близько трьох десятків команд-дієслів, таких як `checkout`, `branch`, `remote` й інших. Та оскільки Git спершу був більше інструментом для СКВ, ніж повною СКВ, дружньою користувачеві, то існують багато таких дієслів, які здійснюють низькорівневу роботу і були створені для послідовного використання у стилі UNIX або використання у скриптах. Ці команди зазвичай називають "кухонними" («plumbing» — трубопровід, сантехніка), а орієнтовані на користувача — "порцеляновими".

Перші дев'ять розділів книги були присвячені переважно користувацьким командам. У цьому розділі розглядатимуться саме низькорівневі кухонні команди, які дають доступ до внутрішніх процесів Git, та допомогають продемонструвати як і чому Git працює саме так, а не інакше. Багато з цих команд призначені не для ручного використання з командного рядка, а є блоками для побудови нових інструментів та користувацьких скриптів.

Коли ви виконуєте `git init` у новій чи існуючій теці, Git створює теку `.git`, де знаходиться майже все, що Git зберігає і чим оперує. Якщо ви бажаєте зробити резервну копію чи клон свого репозиторію, копіювання лише цієї теки в інше місце дає вам майже все, що необхідно. Даний розділ присвячений вмісту цієї теки. А ось так вона виглядає:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Там можуть бути й деякі інші файли, вищенаведений приклад є свіжим репозиторієм — це те, що ви типово бачитимете безпосередньо після виконання `git init`. Файл `description` використовується тільки програмою GitWeb, не звертайте на нього увагу. Файл `config` містить конфігураційні параметри, специфічні саме для конкретного репозиторію, а в теці `info` знаходиться файл з глобальними налаштуваннями ігнорування файлів — він дозволяє виключити шляхи, які ви не бажаєте додавати у файл .gitignore. Тека `hooks` містить клієнтські та серверні гаки, які були детальніше розглянуті у розділі **"Git Hooks"**.

Лишилися ще чотири важливі елементи: файли `HEAD` та (ще не створений) `index`, і теки `objects` та `refs`. Це основні складові Git. У теці `objects` зберігається увесь вміст вашої бази даних, тека `refs` містить вказівники на об'єкти комітів у цих даних (гілки), файл `HEAD` указує на поточну гілку — на яку ви зараз переключені, а у файлі `index` зберігається вміст індексу. Тепер ми детально розберемося з цими елементами, щоб зрозуміти як працює Git.

Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time. To demonstrate, you can use the plumbing command `hash-object`, which takes some data, stores it in your `.git` directory, and gives you back the key the data is stored as. First, you initialize a new Git repository and verify that there is nothing in the `objects` directory:

```
$ 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 has initialized the `objects` directory and created `pack` and `info` subdirectories in it, but there are no regular files. Now, store some text in your Git database:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

The `-w` tells `hash-object` to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from stdin; if you don't specify this, `hash-object` expects a file path at the end. The output from the command is a 40-character checksum hash. This is the SHA-1 hash – a checksum of the content you're storing plus a header, which you'll learn about in a bit. Now you can see how Git has stored your data:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

You can see a file in the `objects` directory. This is how Git stores the content initially – as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

You can pull the content back out of Git with the `cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to it instructs the `cat-file` command to figure out the type of content and display it nicely for you:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Then, write some new content to the file, and save it again:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Your database contains the two new versions of the file as well as the first content you stored there:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Now you can revert the file back to the first version

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

or the second version:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system – just the content. This object type is called a blob. You can have Git tell you the object type of any object in Git, given its SHA-1 key, with `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

The next type we'll look at is the tree, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more tree entries, each of which contains a SHA-1 pointer to a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

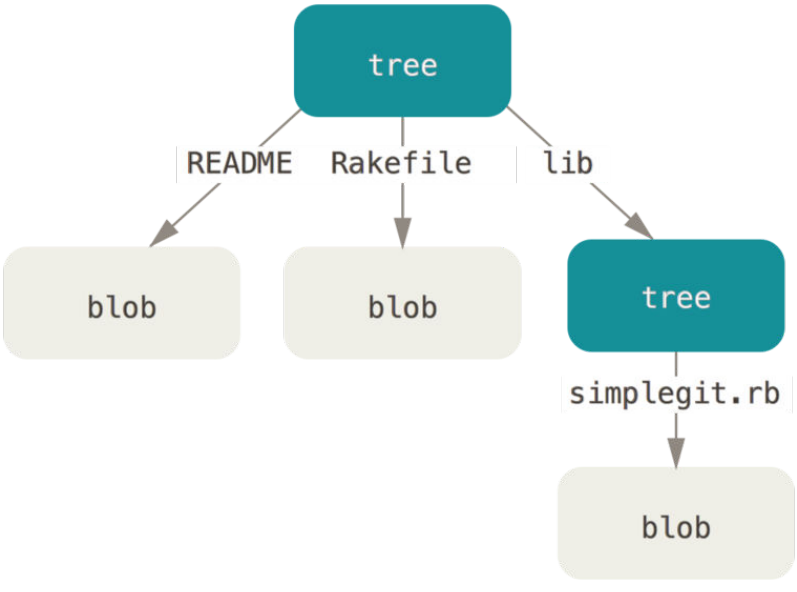
The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptually, the data that Git is storing is something like this:

ЗОБРАЖЕННЯ 10-1

Simple version of the Git data model.



You can fairly easily create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a series of tree objects from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry – the first version of your test.txt file – you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the test.txt file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In this case, you're specifying a mode of `100644`, which means it's a normal file. Other options are `100755`, which means it's an executable file; and `120000`, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible – these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

Now, you can use the `write-tree` command to write the staging area out to a tree object. No `-w` option is needed – calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

You can also verify that this is a tree object:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

You'll now create a new tree with the second version of test.txt and a new file as well:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Your staging area now has the new version of test.txt as well as the new file new.txt. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Notice that this tree has both file entries and also that the test.txt SHA-1 is the "version 2" SHA-1 from earlier (`1f7a7a`). Just for fun, you'll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling `read-tree`. In this case, you can read an existing tree into your staging area as a subtree by using the `--prefix` option to `read-tree`:

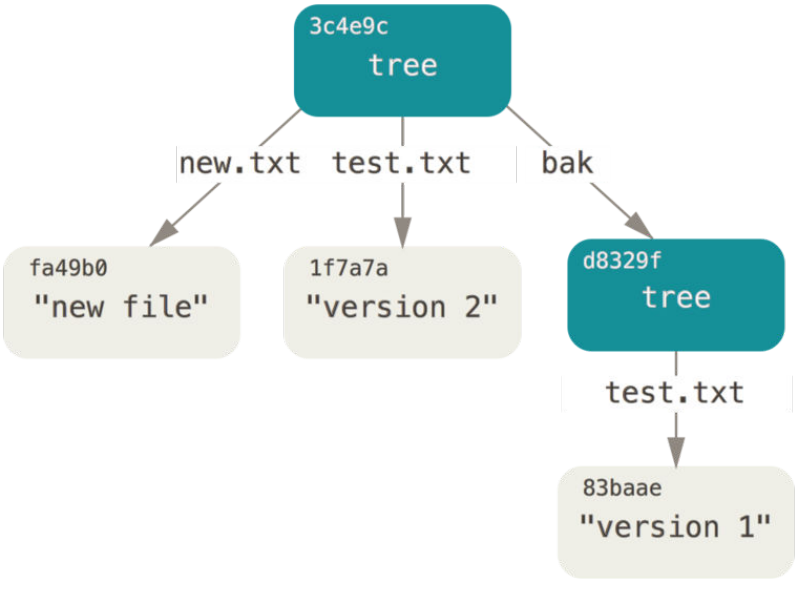
```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
```

```
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named bak that contained the first version of the test.txt file. You can think of the data that Git contains for these structures as being like this:

ЗОБРАЖЕННЯ 10-2

The content structure of your current Git data.



Commit Objects

You have three trees that specify the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all three SHA-1 values in order to recall the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

To create a commit object, you call `commit-tree` and specify a single tree SHA-1 and which commit objects, if any, directly preceded it. Start with the first tree you wrote:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Now you can look at your new commit object with `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
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information (which uses your `user.name` and `user.email` configuration settings and a timestamp); a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1:

```
$ 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

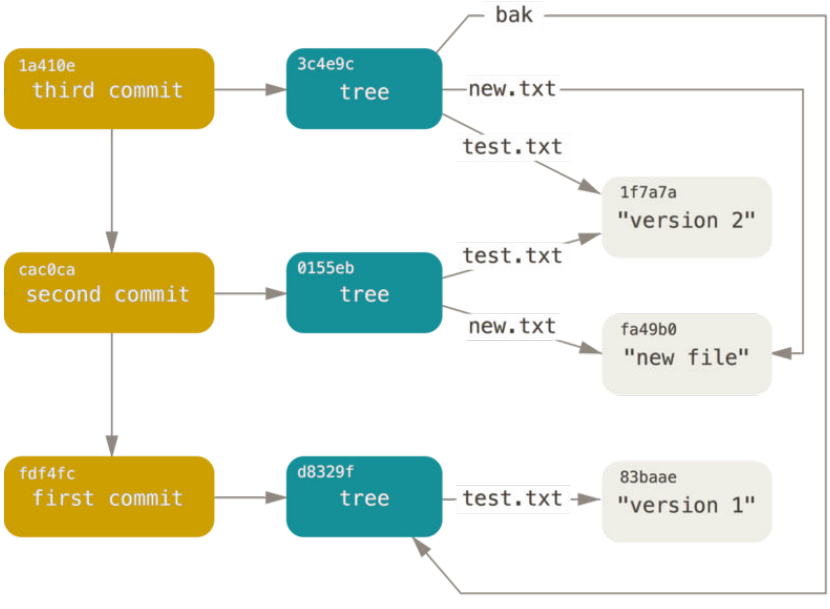
    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

Amazing. You've just done the low-level operations to build up a Git history without using any of the front end commands. This is essentially what Git does when you run the `git add` and `git commit` commands – it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects – the blob, the tree, and the commit – are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example directory now, commented with what they store:

```
$ 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
```

If you follow all the internal pointers, you get an object graph something like this:



ЗОБРАЖЕННЯ 10-3

All the objects in your Git directory.

Object Storage

We mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object – in this case, the string "what is up, doc?" – interactively in the Ruby scripting language.

You can start up interactive Ruby mode with the `irb` command:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the SHA1 digest library with the `require` command and then calling `Digest::SHA1.hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with zlib, which you can do in Ruby with the zlib library. First, you need to require the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> 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"
```

Finally, you'll write your zlib-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously zlib-compressed content to the file with a `write()` call on the resulting file handle:

```
>> 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
```

That's it – you've created a valid Git blob object. All Git objects are stored the same way, just with different types – instead of the string blob, the header will

begin with commit or tree. Also, although the blob content can be nearly anything, the commit and tree content are very specifically formatted.

Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value.

In Git, these are called "references" or "refs;" you can find the files that contain the SHA-1 values in the `.git/refs` directory. In the current project, this directory contains no files, but it does contain a simple structure:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren't encouraged to directly edit the reference files. Git provides a safer command to do this if you want to update a reference called `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

```
$ git update-ref refs/heads/test cac0ca
```

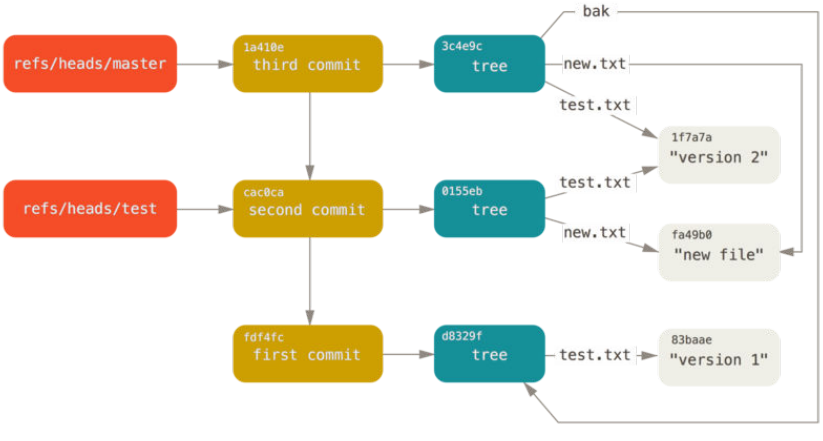
Your branch will contain only work from that commit down:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like this:

ЗОБРАЖЕННЯ 10-4

Git directory objects with branch head references included.



When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create.

The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 of the last commit? The answer is the HEAD file.

The HEAD file is a symbolic reference to the branch you're currently on. By symbolic reference, we mean that unlike a normal reference, it doesn't general-

ly contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout test`, Git updates the file to look like this:

```
$ cat .git/HEAD
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

You can also manually edit this file, but again a safer command exists to do so: `symbolic-ref`. You can read the value of your HEAD via this command:

```
$ git symbolic-ref HEAD
refs/heads/master
```

You can also set the value of HEAD:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

We just finished discussing Git's three main object types, but there is a fourth. The tag object is very much like a commit object – it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It's like a branch reference, but it never moves – it always points to the same commit but gives it a friendlier name.

As discussed in **Chapter 2**, there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That is all a lightweight tag is – a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (`-a` specifies that it's an annotated tag):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 value it created:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Now, run the `cat-file` command on that SHA-1 value:

```
$ 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
```

Notice that the object entry points to the commit SHA-1 value that you tagged. Also notice that it doesn't need to point to a commit; you can tag any Git object. In the Git source code, for example, the maintainer has added their GPG public key as a blob object and then tagged it. You can view the public key by running this in a clone of the Git repository:

```
$ git cat-file blob junio-gpg-pub
```

The Linux kernel repository also has a non-commit-pointing tag object – the first tag created points to the initial tree of the import of the source code.

Remotes

The third type of reference that you'll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the `refs/remotes` directory. For instance, you can add a remote called `origin` and push your `master` branch to it:

```
$ 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
```

Then, you can see what the `master` branch on the `origin` remote was the last time you communicated with the server, by checking the `refs/remotes/origin/master` file:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that they're considered read-only. You can `git checkout` to one, but Git won't point HEAD at one, so you'll never update it with a `commit` command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects – 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ 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 compresses the contents of these files with zlib, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. To demonstrate, we'll add the `repo.rb` file from the Grit library – this is about a 22K source code file:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > re
$ 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%)
```

If you look at the resulting tree, you can see the SHA-1 value your repo.rb file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

You have two nearly identical 22K objects on your disk. Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a "loose" object format. However, occasionally Git packs up several of these objects into a single binary file called a "packfile" in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ 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)
```

If you look in your objects directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ 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
```

The objects that remain are the blobs that aren't pointed to by any commit – in this case, the "what is up, doc?" example and the "test content" example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the `gc` were collectively about 22K in size, the new packfile is only 7K. You've cut your disk usage by ⅔ by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e300009055
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
```

Here, the `033b4` blob, which if you remember was the first version of your repo.rb file, is referencing the `b042a` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you

can see that `b042a` takes up 22K of the file, but that `033b4` only takes up 9 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta – this is because you're most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space, but you can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, we've used simple mappings from remote branches to local references, but they can be more complex. Suppose you add a remote like this:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

It adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching:

```
[remote "origin"]
        url = https://github.com/schacon/simplegit-progit
        fetch = +refs/heads/*:refs/remotes/origin/*
```

The format of the refspec is an optional +, followed by `<src>:<dst>`, where `<src>` is the pattern for references on the remote side and `<dst>` is where those references will be written locally. The + tells Git to update the reference even if it isn't a fast-forward.

In the default case that is automatically written by a `git remote add` command, Git fetches all the references under `refs/heads/` on the server and writes them to `refs/remotes/origin/` locally. So, if there is a `master` branch on the server, you can access the log of that branch locally via

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They're all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull down only the `master` branch each time, and not every other branch on the remote server, you can change the fetch line to

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

This is just the default refspec for `git fetch` for that remote. If you want to do something one time, you can specify the refspec on the command line, too. To pull the `master` branch on the remote down to `origin/mymaster` locally, you can run

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

You can also specify multiple refspecs. On the command line, you can pull down several branches like so:

```
$ 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
```

In this case, the master branch pull was rejected because it wasn't a fast-forward reference. You can override that by specifying the + in front of the refspec.

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the master and experiment branches, add two lines:

```
[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
```

You can't use partial globs in the pattern, so this would be invalid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the master branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[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/*
```

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? You accomplish that by using refspecs to push.

If the QA team wants to push their `master` branch to `qa/master` on the remote server, they can run

```
$ git push origin master:refs/heads/qa/master
```

If they want Git to do that automatically each time they run `git push origin`, they can add a `push` value to their config file:

```
[remote "origin"]
        url = https://github.com/schacon/simplegit-progit
        fetch = +refs/heads/*:refs/remotes/origin/*
        push = refs/heads/master:refs/heads/qa/master
```

Again, this will cause a `git push origin` to push the local `master` branch to the remote `qa/master` branch by default.

Deleting References

You can also use the refspec to delete references from the remote server by running something like this:

```
$ git push origin :topic
```

Because the refspec is `<src>:<dst>`, by leaving off the `<src>` part, this basically says to make the topic branch on the remote nothing, which deletes it.

Transfer Protocols

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP `GET` requests, where the client can assume the layout of the Git repository on the server.

> The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the simplegit library:

```
$ git clone http://server/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-server-info` command, which is why you need to enable that as a `post-receive` hook in order for the HTTP transport to work properly:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Now you have a list of the remote references and SHA-1s. Next, you look for what the HEAD reference is so you know what to check out when you’re finished:

```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the `master` branch when you’ve completed the process. At this point, you’re ready to start the walking process. Because your start-

ing point is the `ca82a6` commit object you saw in the `info/refs` file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back – that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ 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
```

Next, you have two more objects to retrieve – `cfda3b`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops – it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this – the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there – this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it – because the index lists the SHA-1s of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the `master` branch that was pointed to by the HEAD reference you downloaded at the beginning.

The Smart Protocol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

UPLOADING DATA

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

SSH

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to your server. It tries to

run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
        delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1+github-607-gfba4028 delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has – in this case, just the `master` branch and its SHA-1. The first line also has a list of the server's capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

Each line starts with a 4-character hex value specifying how long the rest of the line is. Your first line starts with 005b, which is hexadecimal for 91, meaning that 91 bytes remain on that line. The next line starts with 003e, which is 62, so you read the remaining 62 bytes. The next line is 0000, meaning the server is done with its references listing.

Now that it knows the server's state, your `send-pack` process determines what commits it has that the server doesn't. For each reference that this push will update, the `send-pack` process tells the `receive-pack` process that information. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0085ca82a6dff817ec66f44342007202690a93763949  15027957951b64cf874c3557a0f3547bd83b3ff6 \
        refs/heads/master report-status
006700000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d \
        refs/heads/experiment
0000
```

Git sends a line for each reference you're updating with the line's length, the old SHA-1, the new SHA-1, and the reference that is being updated. The first line also has the client's capabilities. The SHA-1 value of all '0's means that nothing was there before – because you're adding the experiment reference. If you were deleting a reference, you would see the opposite: all '0's on the right side.

Next, the client sends a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
000Aunpack ok
```

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
000000ab6c5f0e45abd7832bf23074a333f739977c9e8188 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
```

That's the end of the first client-server exchange. The client then makes another request, this time a POST, with the data that `git-upload-pack` provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The POST request includes the `send-pack` output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

DOWNLOADING DATA

When you download data, the `fetch-pack` and `upload-pack` processes are involved. The client initiates a `fetch-pack` process that connects to an `upload-pack` process on the remote side to negotiate what data will be transferred down.

SSH

If you're doing the fetch over SSH, `fetch-pack` runs something like this:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After `fetch-pack` connects, `upload-pack` sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADmulti_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
```

This is very similar to what `receive-pack` responds with, but the capabilities are different. In addition, it sends back what HEAD points to (`symref=HEAD:refs/heads/master`) so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at what objects it has and responds with the objects that it needs by sending "want" and then the SHA-1 it wants. It sends all the objects it already has with "have" and then the SHA-1. At the end of this list, it writes "done" to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a `GET` to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
000000e7ca82a6dff817ec66f44342007202690a93763949 HEADmulti_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
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We've tried to give you a sense of the general back-and-forth between client and server; if you need more knowledge than this, you'll probably want to take a look at the Git source code.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren’t reachable from any commit and are a few months old.

You can run auto gc manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real gc command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you’ll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ 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
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in **"Git References"**. You can see where you've been at any time by running `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
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ 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
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that 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 – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ 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
```

In this case, you can see your missing commit after the string "dangling commit". You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if

someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ 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
```

Oops – you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ 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
```

Now, gc your database and see how much space you're using:

```
$ 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)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ 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
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in **"Відповідність повідомлення коміту певному формату"**. If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

You must rewrite all the commits downstream from `7b30847` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in "Переписування історії":

```
$ 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
```

The `--index-filter` option is similar to the `--tree-filter` option used in "Переписування історії", except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ 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)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ 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
```

Змінні середовища

Git завжди виконується в командній оболонці `bash` і використовує чимало змінних середовища командної оболонки, щоб визначати свою

поведінку. Інколи буває зручно знати ці змінні та як вони можуть бути використані, щоб змусити Git поводитись так, як вам потрібно. Це не вичерпний перелік всіх змінних середовища, на які Git звертає увагу, але ми розглянемо найбільш корисні з них.

Глобальна поведінка

Дещо з глобальної поведінки Git як комп'ютерної програми, залежить від змінних середовища.

GIT_EXEC_PATH визначає, де Git шукає свої підпрограми (такі як `git-commit`, `git-diff` та інші). Ви можете переглянути поточні параметри, виконавши `git --exec-path`.

HOME зазвичай не вважається змінною, яку можна пристосовувати та змінювати (занадто багато інших речей залежать від неї), але це те місце, де Git шукає файл загальних налаштувань. Якщо ви бажаєте мати справді портативну інсталяцію Git разом з загальними налаштуваннями, ви можете перезаписати `HOME` в портативному профілі командної оболонки Git.

PREFIX аналогічно попередній, але для загальносистемних налаштувань. Git шукає цей файл з налаштуваннями в `$PREFIX/etc/gitconfig`.

GIT_CONFIG_NOSYSTEM, якщо задано, блокує використання файлу з загальносистемними налаштуваннями. Це є корисним у випадку, якщо ваші системні налаштування конфліктують з вашими командами, але ви не маєте в системі достатньо прав, щоб їх змінити або видалити.

GIT_PAGER визначає програму, що використовується для показу багатосторінкового виведення в командному рядку. Якщо не задано, то використовується змінна `PAGER` як запасний варіант.

GIT_EDITOR — це програма-редактор, яку Git стартує, коли користувачу потрібно змінити якийсь текст (наприклад повідомлення коміту). Якщо не задано, використовується змінна `EDITOR`.

Місцезнаходження сховищ

Git використовує декілька змінних середовища, щоб визначити, як він взаємодіятиме з поточним сховищем.

GIT_DIR — це місцезнаходження теки `.git`. Якщо цю змінну не задано, Git проходить по дереву директорій, поки не отримає `~` чи `/`, шукаючи щокроку директорію `.git`.

GIT_CEILING_DIRECTORIES контролює поведінку процесу пошуку `.git` директорії. Якщо ви працюєте з директоріями, які завантажуються надто повільно (знаходяться на стрічковому накопичувачі, чи передаються через повільну мережу), ви мабуть захочете, щоб Git зупинявся раніше ніж зазвичай, особливо в процесі запуску Git для створення запиту командного рядка.

GIT_WORK_TREE — це місцезнаходження кореня локальної робочої директорії, якщо сховище не є чистим. Якщо не задано, то використовується батьківська директорія `$GIT_DIR`.

GIT_INDEX_FILE — це шлях до індексного файлу (тільки для сховищ, які не є чистими).

GIT_OBJECT_DIRECTORY може використовуватись для вказівки місцезнаходження директорії з об'єктами, котра зазвичай розташована в `.git/objects`.

GIT_ALTERNATE_OBJECT_DIRECTORIES — це список розділений двокрапкою (відформатований як `/dir/one:/dir/two:…`) котрий вказує Git, де шукати об'єкти, якщо вони відсутні в `GIT_OBJECT_DIRECTORY`. Якщо трапилось так, що у вас є багато проектів з файлами великого розміру та однакового вмісту, цю змінну можна використати, щоб уникнути зберігання численних копій таких файлів.

Специфікації файлів

Специфікації шляхів (pathspecs) визначають, як задаються шляхи до різноманітних компонентів Git включно з використанням шаблонів заміни (джокер *). Ці специфікації зокрема використовуються у файлі `.gitignore`, а також у командному рядку (`git add *.c`).

GIT_GLOB_PATHSPECS та GIT_NOGLOB_PATHSPECS контролюють типову поведінку шаблонів заміни в специфікаціях шляхів. Якщо `GIT_GLOB_PATHSPECS` встановлено в 1, то символи шаблону заміни працюють саме як шаблон заміни (що є типовою поведінкою); якщо ж `GIT_NOGLOB_PATHSPECS` встановлено в 1, то символи шаблону заміни відповідають тільки своїм символам, а це означає, що шаблон `*.c` *відповідає тільки файлу з назвою* "*.c", а не будь-якому файлу з розширенням `.c`. Ви можете змінити цю задану поведінку символів шаблону заміни в кожному окремому випадку за допомогою запису префіксів `:(glob)` або `:(literal)` в шаблоні шляху, як наприклад в `:(glob)*.c`.

GIT_LITERAL_PATHSPECS вимикає обидві вище зазначені поведінки специфікаторів шляху; як символи шаблону заміни, так і спеціальні префікси не працюватимуть.

GIT_ICASE_PATHSPECS встановлює всі специфікатори шляхів нечутливими до регістра літер.

Фіксація комітів

Завершальне створення об'єкта коміту Git зазвичай робиться за допомогою `git-commit-tree`, яка використовує наступні змінні середовища, як основне джерело інформації, і повертається до значень змінних загальносистемних налаштувань тільки в тому випадку, коли ці змінні не задано.

GIT_AUTHOR_NAME — це сприйнятне для людини ім'я в полі "author".

GIT_AUTHOR_EMAIL — це адреса електронної пошти поля "author".

GIT_AUTHOR_DATE — це часова мітка коміта використана полем "author".

GIT_COMMITTER_NAME задає сприйнятне для людини ім'я в полі "committer".

GIT_COMMITTER_EMAIL — це адреса електронної пошти поля "committer".

GIT_COMMITTER_DATE використовується для часової мітки в полі "committer".

EMAIL — це запасна адреса електронної пошти для випадку, коли значення `user.email` в налаштуваннях не задано. Якщо *цю запасну адресу* не задано, то Git використовує системні ім*я користувача та мережеве ім*я комп'ютера.

Мережа

Git використовує бібліотеку `curl` для мережевих операцій через HTTP, отже **GIT_CURL_VERBOSE** вказує Git показувати всі повідомлення, які генеруються цією бібліотекою. Це схоже на старт `curl -v` в командному рядку.

GIT_SSL_NO_VERIFY вказує Git не перевіряти сертифікати SSL. Це може бути необхідно, якщо ви використовуєте самостійно підписаний сертифікат для обслуговування сховищ Git через HTTPS, або ви знаходитесь якраз в процесі налаштування сервера Git, але ще не встановили повноцінний сертифікат.

Якщо швидкість передачі даних деякої операції через HTTP нижча, ніж **GIT_HTTP_LOW_SPEED_LIMIT** байтів за секунду впродовж більш ніж **GIT_HTTP_LOW_SPEED_TIME** секунд, то Git скасує цю операцію. Ці значення перекривають значення налаштувань `http.lowSpeedLimit` та `http.lowSpeedTime`.

GIT_HTTP_USER_AGENT задає текстовий рядок User-Agent, який використовує Git під час мережевого спілкування через HTTP. Типове значення виглядає так: `git/2.0.0`.

Визначення відмінностей та злиття

GIT_DIFF_OPTS названо трохи невдало. Допустимими значеннями є тільки `-u<n>` чи `--unified=<n>`, які контролюють кількість контекстних рядків, що показуватимуться за допомогою команди `git diff`.

GIT_EXTERNAL_DIFF використовується для переозначення параметра `diff.external` в налаштуваннях. Якщо цю змінну задано, то Git викликатиме вибрану програму для команди `git diff`.

GIT_DIFF_PATH_COUNTER та **GIT_DIFF_PATH_TOTAL** є корисними всередині програми, заданої за допомогою `GIT_EXTERNAL_DIFF` чи `diff.external`. Перша змінна визначає індекс порівнюваного файлу в серії (починаючи з 1), а інша — це загальна кількість файлів в пакеті завдань.

GIT_MERGE_VERBOSITY контролює виведення інформації під час рекурсивної стратегії злиття. Допустимі значення:

- 0 не виводить нічого, крім, можливо, одного повідомлення про помилку.
- 1 показує тільки конфлікти.
- 2 показує також зміни у файлах.
- 3 показує пропущені внаслідок відсутності змін файли.
- 4 показує всі шляхи, як вони обробляються.
- 5 і більше, показують докладну інформацію для зневадження.

Типове значення змінної становить 2.

Зневадження

Бажаєте *справді* знати, що Git замислив? Git має достатньо повний набір вбудованих засобів трасування, і все, що вам потрібно зробити, це увімкнути їх. Можливі значення приведених нижче змінних наступні:

- “true”, “1”, або “2” — категорія трасування записується до стандартного потоку помилок stderr.
- Абсолютний шлях, що починається на / — виведення трасування буде записано в цей файл.

GIT_TRACE контролює загальні сліди трасування, які не вписуються до якої-небудь конкретної категорії. Це включає в себе розкриття псевдонімів, та делегацію до інших підпрограм.

```
$ 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 контролює трасування доступу до pack-файлів. Перше поле — це pack-файл, до якого здійснюється доступ, а друге — зсув у цьому файлі:

```
$ 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 вмикає трасування пакетів на рівні мережевих операцій.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-uploac
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git< 97b8860c071898d9e162
20:15:14.867088 pkt-line.c:46           packet:          git< 0f20ae29889d61f2e93a
20:15:14.867094 pkt-line.c:46           packet:          git< 36dc827bc9d17f80ed4f
# […]
```

GIT_TRACE_PERFORMANCE контролює журналювання даних швидкодії. Вивід показує, скільки часу займає кожен окремий виклик 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 показує інформацію, яку Git отримує про сховище та середовище, з якими він взаємодіє.

```
$ 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
```

Інше

GIT_SSH, якщо задано, — це програма, яка викликається замість `ssh`, коли Git намагається підключитися до SSH сервера. Вона стартує у вигляді `$GIT_SSH [username@]host [-p <port>] <command>`. Зверніть увагу, що це не найпростіший спосіб пристосувати виклик `ssh`; додаткові параметри командного рядка в цьому випадку не підтримуються, отже вам доведеться написати також скрипт-обгортку і встановити значення змінної `GIT_SSH`, що вказуватиме на цей скрипт. Простіше, напевно, використовувати файл `~/.ssh/config` для цього.

GIT_ASKPASS — це переозначення змінної `core.askpass` в налаштуваннях. Це програма, що викликається щоразу, коли Git потрібно запитати в користувача облікові дані, та яка може очікувати на текстовий запит, як аргумент командного рядка, і повинна повернути відповідь в стандартний потік `stdout`. (Див. **“Збереження посвідчення (credential)”**, щоб дізнатись докладніше про цю підсистему.)

GIT_NAMESPACE контролює доступ до посилань просторів імен, і еквівалентна параметру `--namespace`. Це в основному корисно на стороні сервера, де ви, можливо, захочете зберігати кілька форків єдиного сховища в одному сховищі, зберігаючи тільки окремо посилання на них.

GIT_FLUSH може використовуватись, щоб змусити Git вживати небуферизоване введення/виведення під час інкрементного запису в стандартний потік stdout. Значення 1 змушує Git очищувати буфер частіше, а для значення 0 буферизуються всі вихідні дані. Типове значення (якщо значення цієї змінної не задано) — вибирати доречну схему буферизації в залежності від виду діяльності та режиму виведення.

GIT_REFLOG_ACTION дозволяє вказати описовий текст, що записується в reflog. Ось приклад:

```
$ 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
```

Summary

You should have a pretty good understanding of what Git does in the background and, to some degree, how it’s implemented. This chapter has covered a number of plumbing commands – commands that are lower level and simpler than the porcelain commands you’ve learned about in the rest of the book. Understanding how Git works at a lower level should make it easier to understand why it’s doing what it’s doing and also to write your own tools and helping scripts to make your specific workflow work for you.

Git as a content-addressable filesystem is a very powerful tool that you can easily use as more than just a VCS. We hope you can use your newfound knowledge of Git internals to implement your own cool application of this technology and feel more comfortable using Git in more advanced ways.

Git in Other Environments A

If you read through the whole book, you've learned a lot about how to use Git at the command line. You can work with local files, connect your repository to others over a network, and work effectively with others. But the story doesn't end there; Git is usually used as part of a larger ecosystem, and the terminal isn't always the best way to work with it. Now we'll take a look at some of the other kinds of environments where Git can be useful, and how other applications (including yours) work alongside Git.

Graphical Interfaces

Git's native environment is in the terminal. New features show up there first, and only at the command line is the full power of Git completely at your disposal. But plain text isn't the best choice for all tasks; sometimes a visual representation is what you need, and some users are much more comfortable with a point-and-click interface.

It's important to note that different interfaces are tailored for different workflows. Some clients expose only a carefully curated subset of Git functionality, in order to support a specific way of working that the author considers effective. When viewed in this light, none of these tools can be called "better" than any of the others, they're simply more fit for their intended purpose. Also note that there's nothing these graphical clients can do that the command-line client can't; the command-line is still where you'll have the most power and control when working with your repositories.

`gitk` and `git-gui`

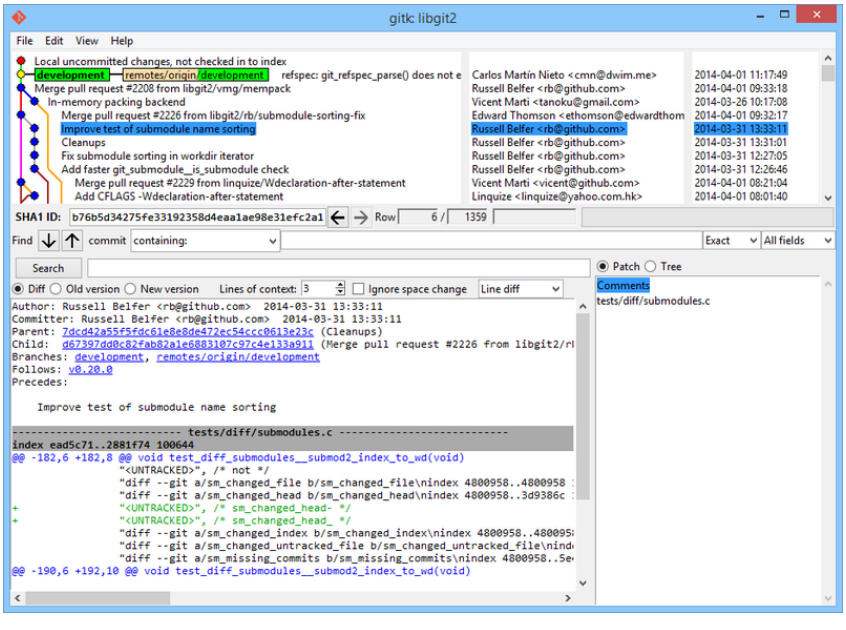
When you install Git, you also get its visual tools, `gitk` and `git-gui`.

`gitk` is a graphical history viewer. Think of it like a powerful GUI shell over `git log` and `git grep`. This is the tool to use when you're trying to find something that happened in the past, or visualize your project's history.

Gitk is easiest to invoke from the command-line. Just `cd` into a Git repository, and type:

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

Gitk accepts many command-line options, most of which are passed through to the underlying `git log` action. Probably one of the most useful is the `--all` flag, which tells gitk to show commits reachable from *any* ref, not just HEAD. Gitk's interface looks like this:



Зображення 1-1.

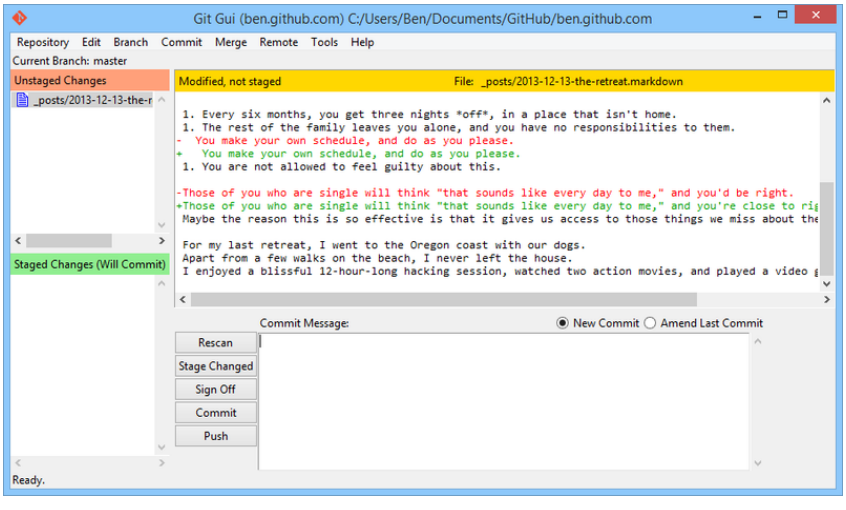
The `gitk` history viewer.

On the top is something that looks a bit like the output of `git log --graph`; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

`git-gui`, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

```
$ git gui
```

And it looks something like this:



Зображення 1-2.

The `git-gui` commit tool.

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to `git commit`. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

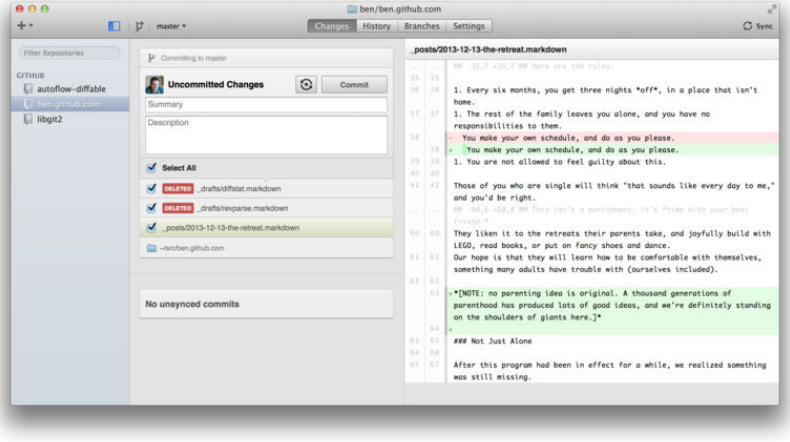
`gitk` and `git-gui` are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

GitHub for Mac and Windows

GitHub has created two workflow-oriented Git clients: one for Windows, and one for Mac. These clients are a good example of workflow-oriented tools – rather than expose *all* of Git's functionality, they instead focus on a curated set of commonly-used features that work well together. They look like this:

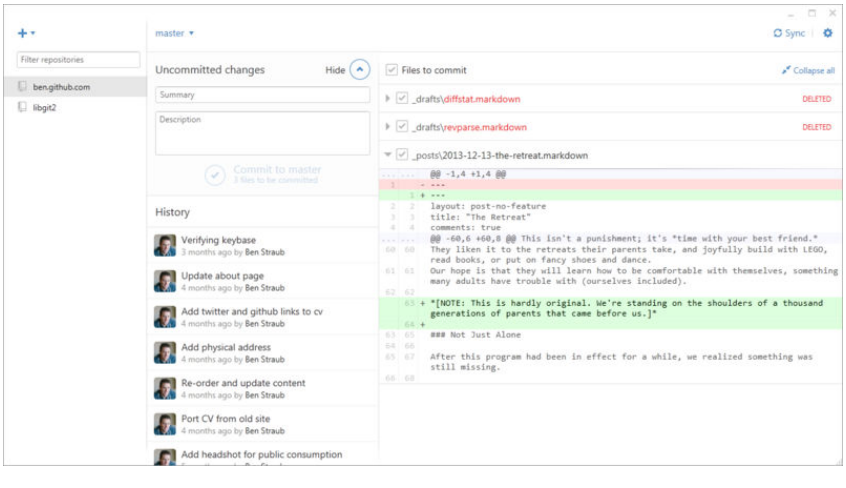
Зображення 1-3.

GitHub for Mac.



Зображення 1-4.

GitHub for Windows.



They are designed to look and work very much alike, so we'll treat them like a single product in this chapter. We won't be doing a detailed rundown of these tools (they have their own documentation), but a quick tour of the "changes" view (which is where you'll spend most of your time) is in order.

- On the left is the list of repositories the client is tracking; you can add a repository (either by cloning or attaching locally) by clicking the "+" icon at the top of this area.
- In the center is a commit-input area, which lets you input a commit message, and select which files should be included. (On Windows, the commit history is displayed directly below this; on Mac, it's on a separate tab.)
- On the right is a diff view, which shows what's changed in your working directory, or which changes were included in the selected commit.
- The last thing to notice is the "Sync" button at the top-right, which is the primary way you interact over the network.

You don't need a GitHub account to use these tools. While they're designed to highlight GitHub's service and recommended workflow, they will happily work with any repository, and do network operations with any Git host.

INSTALLATION

GitHub for Windows can be downloaded from ***https://windows.github.com***, and GitHub for Mac from ***https://mac.github.com***. When the applications are first run, they walk you through all the first-time Git setup, such as configuring your name and email address, and both set up sane defaults for many common configuration options, such as credential caches and CRLF behavior.

Both are "evergreen" – updates are downloaded and installed in the background while the applications are open. This helpfully includes a bundled version of Git, which means you probably won't have to worry about manually updating it again. On Windows, the client includes a shortcut to launch Powershell with Posh-git, which we'll talk more about later in this chapter.

The next step is to give the tool some repositories to work with. The client shows you a list of the repositories you have access to on GitHub, and can clone them in one step. If you already have a local repository, just drag its directory from the Finder or Windows Explorer into the GitHub client window, and it will be included in the list of repositories on the left.

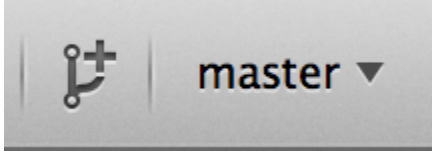
RECOMMENDED WORKFLOW

Once it's installed and configured, you can use the GitHub client for many common Git tasks. The intended workflow for this tool is sometimes called the "GitHub Flow." We cover this in more detail in **"Потік роботи GitHub"**, but the general gist is that (a) you'll be committing to a branch, and (b) you'll be syncing up with a remote repository fairly regularly.

Branch management is one of the areas where the two tools diverge. On Mac, there's a button at the top of the window for creating a new branch:

Зображення 1-5.

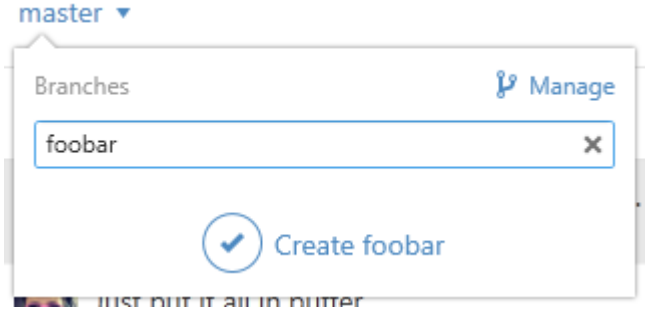
"Create Branch" button on Mac.



On Windows, this is done by typing the new branch's name in the branch-switching widget:

Зображення 1-6.

Creating a branch on Windows.



Once your branch is created, making new commits is fairly straightforward. Make some changes in your working directory, and when you switch to the GitHub client window, it will show you which files changed. Enter a commit message, select the files you'd like to include, and click the "Commit" button (ctrl-enter or ⌘-enter).

The main way you interact with other repositories over the network is through the "Sync" feature. Git internally has separate operations for pushing,

fetching, merging, and rebasing, but the GitHub clients collapse all of these into one multi-step feature. Here's what happens when you click the Sync button:

1. `git pull --rebase`. If this fails because of a merge conflict, fall back to `git pull --no-rebase`.
2. `git push`.

This is the most common sequence of network commands when working in this style, so squashing them into one command saves a lot of time.

SUMMARY

These tools are very well-suited for the workflow they're designed for. Developers and non-developers alike can be collaborating on a project within minutes, and many of the best practices for this kind of workflow are baked into the tools. However, if your workflow is different, or you want more control over how and when network operations are done, we recommend you use another client or the command line.

Other GUIs

There are a number of other graphical Git clients, and they run the gamut from specialized, single-purpose tools all the way to apps that try to expose everything Git can do. The official Git website has a curated list of the most popular clients at ***http://git-scm.com/downloads/guis***. A more comprehensive list is available on the Git wiki site, at ***https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces***.

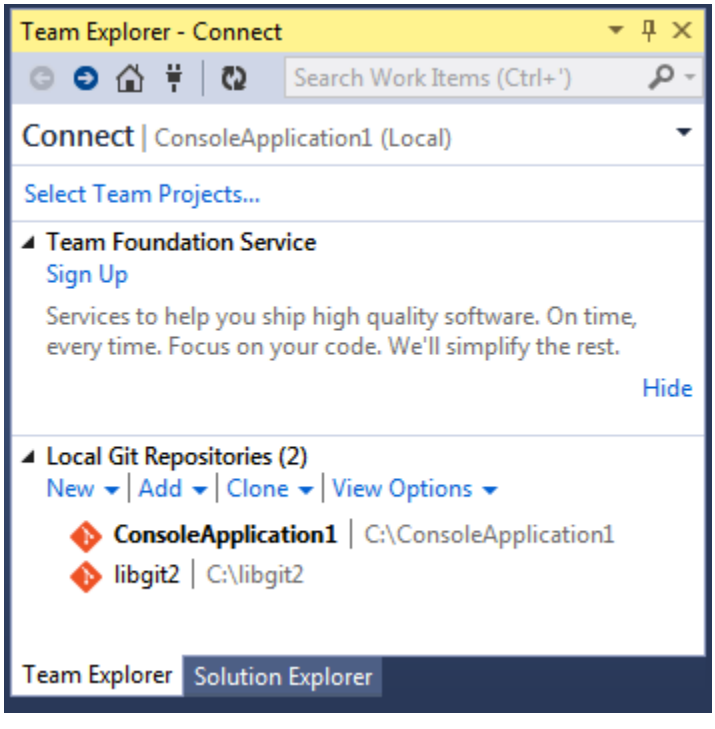
Git in Visual Studio

Starting with Visual Studio 2013 Update 1, Visual Studio users have a Git client built directly into their IDE. Visual Studio has had source-control integration features for quite some time, but they were oriented towards centralized, file-locking systems, and Git was not a good match for this workflow. Visual Studio 2013's Git support has been separated from this older feature, and the result is a much better fit between Studio and Git.

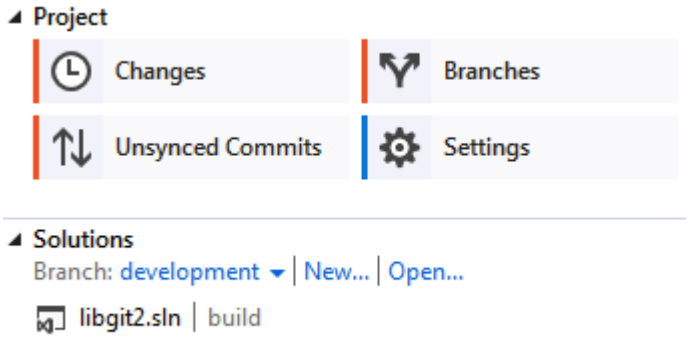
To locate the feature, open a project that's controlled by Git (or just `git init` an existing project), and select View > Team Explorer from the menu. You'll see the "Connect" view, which looks a bit like this:

Зображення 1-7.

Connecting to a Git repository from Team Explorer.



Visual Studio remembers all of the projects you've opened that are Git-controlled, and they're available in the list at the bottom. If you don't see the one you want there, click the "Add" link and type in the path to the working directory. Double clicking on one of the local Git repositories leads you to the Home view, which looks like **Figure A-8**. This is a hub for performing Git actions; when you're *writing* code, you'll probably spend most of your time in the "Changes" view, but when it comes time to pull down changes made by your teammates, you'll use the "Unsynced Commits" and "Branches" views.



Зображення 1-8.

The "Home" view for a Git repository in Visual Studio.

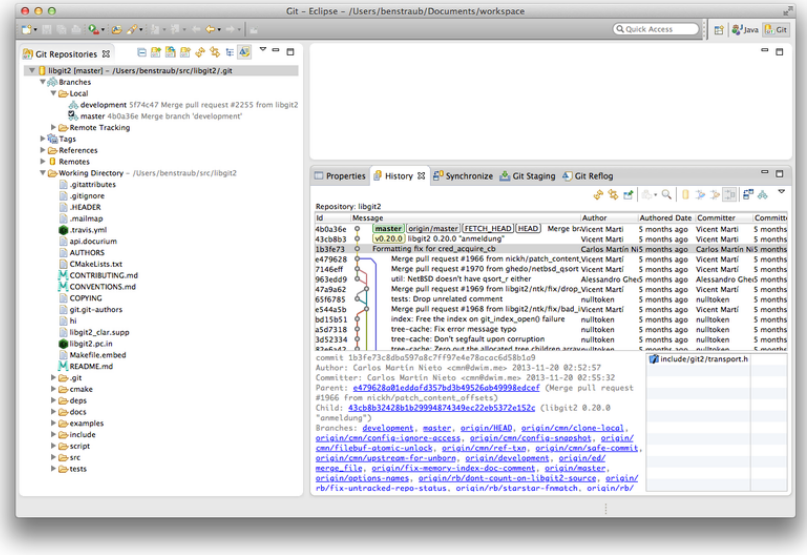
Visual Studio now has a powerful task-focused UI for Git. It includes a linear history view, a diff viewer, remote commands, and many other capabilities. For complete documentation of this feature (which doesn't fit here), go to ***http://msdn.microsoft.com/en-us/library/hh850437.aspx***.

Git in Eclipse

Eclipse ships with a plugin called Egit, which provides a fairly-complete interface to Git operations. It's accessed by switching to the Git Perspective (Window > Open Perspective > Other…, and select "Git").

Зображення 1-9.

Eclipse's EGit environment.



EGit comes with plenty of great documentation, which you can find by going to Help > Help Contents, and choosing the "EGit Documentation" node from the contents listing.

Git in Bash

If you're a Bash user, you can tap into some of your shell's features to make your experience with Git a lot friendlier. Git actually ships with plugins for several shells, but it's not turned on by default.

First, you need to get a copy of the `contrib/completion/git-completion.bash` file out of the Git source code. Copy it somewhere handy, like your home directory, and add this to your `.bashrc`:

```
. ~/git-completion.bash
```

Once that's done, change your directory to a git repository, and type:

```
$ git chec<tab>
```

…and Bash will auto-complete to `git checkout`. This works with all of Git's subcommands, command-line parameters, and remotes and ref names where appropriate.

It's also useful to customize your prompt to show information about the current directory's Git repository. This can be as simple or complex as you want, but there are generally a few key pieces of information that most people want, like the current branch, and the status of the working directory. To add these to your prompt, just copy the `contrib/completion/git-prompt.sh` file from Git's source repository to your home directory, add something like this to your `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

The `\w` means print the current working directory, the `\$` prints the `$` part of the prompt, and `__git_ps1 " (%s)"` calls the function provided by `git-prompt.sh` with a formatting argument. Now your bash prompt will look like this when you're anywhere inside a Git-controlled project:



Зображення 1-10.

Customized bash prompt.

Both of these scripts come with helpful documentation; take a look at the contents of `git-completion.bash` and `git-prompt.sh` for more information.

Git in Zsh

Git also ships with a tab-completion library for Zsh. Just copy `contrib/completion/git-completion.zsh` to your home directory and source it from your `.zshrc`. Zsh's interface is a bit more powerful than Bash's:

```
$ 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
```

Ambiguous tab-completions aren't just listed; they have helpful descriptions, and you can graphically navigate the list by repeatedly hitting tab. This works with Git commands, their arguments, and names of things inside the repository (like refs and remotes), as well filenames and all the other things Zsh knows how to tab-complete.

Zsh happens to be fairly compatible with Bash when it comes to prompt customization, but it allows you to have a right-side prompt as well. To include the branch name on the right side, add these lines to your `~/.zshrc` file:

```
setopt prompt_subst
. ~/git-prompt.sh
export RPROMPT=$'$(__git_ps1 "%s")'
```

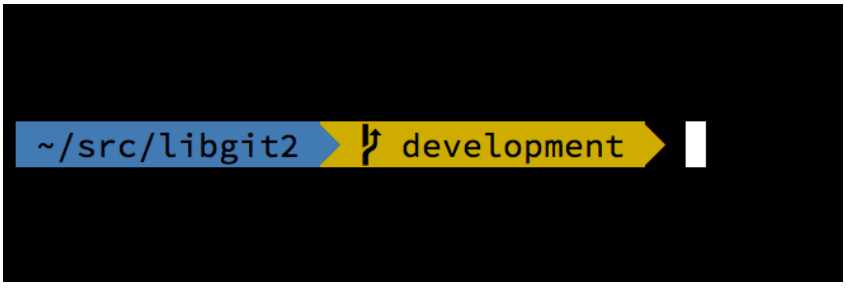
This results in a display of the current branch on the right-hand side of the terminal window, whenever your shell is inside a Git repository. It looks a bit like this:



Зображення 1-11.

Customized zsh prompt.

Zsh is powerful enough that there are entire frameworks dedicated to making it better. One of them is called “oh-my-zsh”, and it can be found at ***https://github.com/robbyrussell/oh-my-zsh***. oh-my-zsh’s plugin system comes with powerful git tab-completion, and it has a variety of prompt “themes”, many of which display version-control data. **Figure A-12** is just one example of what can be done with this system.



Зображення 1-12.

An example of an oh-my-zsh theme.

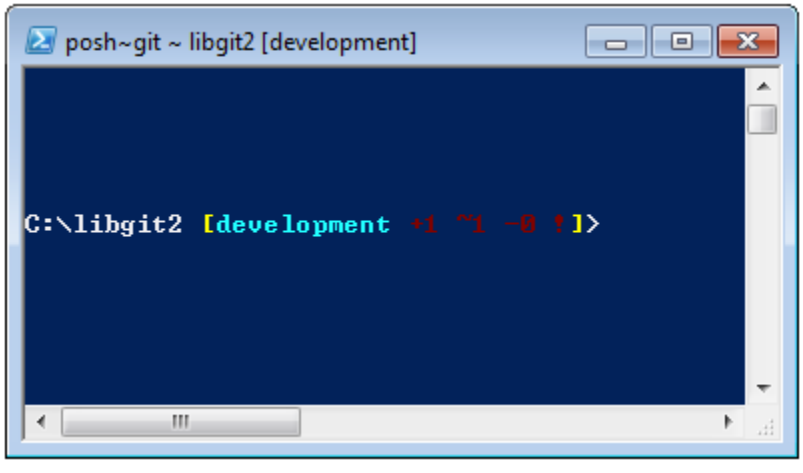
Git in Powershell

The standard command-line terminal on Windows (`cmd.exe`) isn’t really capable of a customized Git experience, but if you’re using Powershell, you’re in luck. A package called Posh-Git (***https://github.com/dahlbyk/posh-git***) pro-

vides powerful tab-completion facilities, as well as an enhanced prompt to help you stay on top of your repository status. It looks like this:

Зображення 1-13.

Powershell with Posh-git.



If you've installed GitHub for Windows, Posh-Git is included by default, and all you have to do is add these lines to your `profile.ps1` (which is usually located in `C:\Users\<username>\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

If you're not a GitHub for Windows user, just download a Posh-Git release from (***https://github.com/dahlbyk/posh-git***), and uncompress it to the `WindowsPowershell` directory. Then open a Powershell prompt as the administrator, and do this:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

This will add the proper line to your `profile.ps1` file, and posh-git will be active the next time you open your prompt.

Summary

You've learned how to harness Git's power from inside the tools that you use during your everyday work, and also how to access Git repositories from your own programs.

Embedding Git in your Applications

B

If your application is for developers, chances are good that it could benefit from integration with source control. Even non-developer applications, such as document editors, could potentially benefit from version-control features, and Git's model works very well for many different scenarios.

If you need to integrate Git with your application, you have essentially three choices: spawning a shell and using the Git command-line tool; Libgit2; and JGit.

Command-line Git

One option is to spawn a shell process and use the Git command-line tool to do the work. This has the benefit of being canonical, and all of Git's features are supported. This also happens to be fairly easy, as most runtime environments have a relatively simple facility for invoking a process with command-line arguments. However, this approach does have some downsides.

One is that all the output is in plain text. This means that you'll have to parse Git's occasionally-changing output format to read progress and result information, which can be inefficient and error-prone.

Another is the lack of error recovery. If a repository is corrupted somehow, or the user has a malformed configuration value, Git will simply refuse to perform many operations.

Yet another is process management. Git requires you to maintain a shell environment on a separate process, which can add unwanted complexity. Trying to coordinate many of these processes (especially when potentially accessing the same repository from several processes) can be quite a challenge.

Libgit2

© Another option at your disposal is to use Libgit2. Libgit2 is a dependency-free implementation of Git, with a focus on having a nice API for use within other programs. You can find it at ***http://libgit2.github.com***.

First, let's take a look at what the C API looks like. Here's a whirlwind tour:

```
// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
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);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);
```

The first couple of lines open a Git repository. The `git_repository` type represents a handle to a repository with a cache in memory. This is the simplest method, for when you know the exact path to a repository's working directory or `.git` folder. There's also the `git_repository_open_ext` which includes options for searching, `git_clone` and friends for making a local clone of a remote repository, and `git_repository_init` for creating an entirely new repository.

The second chunk of code uses rev-parse syntax (see **"Гілкові посилання"** for more on this) to get the commit that HEAD eventually points to. The type returned is a `git_object` pointer, which represents something that exists in the Git object database for a repository. `git_object` is actually a "parent" type for several different kinds of objects; the memory layout for each of the "child" types is the same as for `git_object`, so you can safely cast to the right one. In this case, `git_object_type(commit)` would return `GIT_OBJ_COMMIT`, so it's safe to cast to a `git_commit` pointer.

The next chunk shows how to access the commit's properties. The last line here uses a `git_oid` type; this is Libgit2's representation for a SHA-1 hash.

From this sample, a couple of patterns have started to emerge:

- If you declare a pointer and pass a reference to it into a Libgit2 call, that call will probably return an integer error code. A `0` value indicates success; anything less is an error.
- If Libgit2 populates a pointer for you, you're responsible for freeing it.
- If Libgit2 returns a `const` pointer from a call, you don't have to free it, but it will become invalid when the object it belongs to is freed.
- Writing C is a bit painful.

That last one means it isn't very probable that you'll be writing C when using Libgit2. Fortunately, there are a number of language-specific bindings available that make it fairly easy to work with Git repositories from your specific language and environment. Let's take a look at the above example written using the Ruby bindings for Libgit2, which are named Rugged, and can be found at ***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
```

As you can see, the code is much less cluttered. Firstly, Rugged uses exceptions; it can raise things like `ConfigError` or `ObjectError` to signal error conditions. Secondly, there's no explicit freeing of resources, since Ruby is garbage-collected. Let's take a look at a slightly more complicated example: crafting a commit from scratch

```
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 Create a new blob, which contains the contents of a new file.

❷ Populate the index with the head commit's tree, and add the new file at the path `newfile.txt`.

❸ This creates a new tree in the ODB, and uses it for the new commit.

❹ We use the same signature for both the author and committer fields.

❺ The commit message.

❻ When creating a commit, you have to specify the new commit's parents. This uses the tip of HEAD for the single parent.

❼ Rugged (and Libgit2) can optionally update a reference when making a commit.

❽ The return value is the SHA-1 hash of a new commit object, which you can then use to get a `Commit` object.

The Ruby code is nice and clean, but since Libgit2 is doing the heavy lifting, this code will run pretty fast, too. If you're not a rubyist, we touch on some other bindings in **"Other Bindings"**.

Advanced Functionality

Libgit2 has a couple of capabilities that are outside the scope of core Git. One example is pluggability: Libgit2 allows you to provide custom "backends" for several types of operation, so you can store things in a different way than stock Git does. Libgit2 allows custom backends for configuration, ref storage, and the object database, among other things.

Let's take a look at how this works. The code below is borrowed from the set of backend examples provided by the Libgit2 team (which can be found at ***https://github.com/libgit2/libgit2-backends***). Here's how a custom backend for the object database is set up:

```
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
```

(Note that errors are captured, but not handled. We hope your code is better than ours.)

❶ Initialize an empty object database (ODB) "frontend," which will act as a container for the "backends" which are the ones doing the real work.

❷ Initialize a custom ODB backend.

❸ Add the backend to the frontend.

❹ Open a repository, and set it to use our ODB to look up objects.

But what is this `git_odb_backend_mine` thing? Well, that's the constructor for your own ODB implementation, and you can do whatever you want in there, so long as you fill in the `git_odb_backend` structure properly. Here's what it *could* look like:

```
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;
}
```

The subtlest constraint here is that `my_backend_struct`'s first member must be a `git_odb_backend` structure; this ensures that the memory layout is what the Libgit2 code expects it to be. The rest of it is arbitrary; this structure can be as large or small as you need it to be.

The initialization function allocates some memory for the structure, sets up the custom context, and then fills in the members of the `parent` structure that it supports. Take a look at the `include/git2/sys/odb_backend.h` file in the Libgit2 source for a complete set of call signatures; your particular use case will help determine which of these you'll want to support.

Other Bindings

Libgit2 has bindings for many languages. Here we show a small example using a few of the more complete bindings packages as of this writing; libraries exist for many other languages, including C++, Go, Node.js, Erlang, and the JVM, all in various stages of maturity. The official collection of bindings can be found by browsing the repositories at ***https://github.com/libgit2***. The code we'll write will return the commit message from the commit eventually pointed to by HEAD (sort of like `git log -1`).

LIBGIT2SHARP

If you're writing a .NET or Mono application, LibGit2Sharp (***https://github.com/libgit2/libgit2sharp***) is what you're looking for. The bindings are written in C#, and great care has been taken to wrap the raw Libgit2 calls with native-feeling CLR APIs. Here's what our example program looks like:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

For desktop Windows applications, there's even a NuGet package that will help you get started quickly.

OBJECTIVE-GIT

If your application is running on an Apple platform, you're likely using Objective-C as your implementation language. Objective-Git (***https://github.com/libgit2/objective-git***) is the name of the Libgit2 bindings for that environment. The example program looks like this:

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] err
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git is fully interoperable with Swift, so don't fear if you've left Objective-C behind.

PYGIT2

The bindings for Libgit2 in Python are called Pygit2, and can be found at ***http://www.pygit2.org/***. Our example program:

```
pygit2.Repository("/path/to/repo") # open repository
    .head                          # get the current branch
    .peel(pygit2.Commit)           # walk down to the commit
    .message                       # read the message
```

Further Reading

Of course, a full treatment of Libgit2's capabilities is outside the scope of this book. If you want more information on Libgit2 itself, there's API documentation at ***https://libgit2.github.com/libgit2***, and a set of guides at ***https://libgit2.github.com/docs***. For the other bindings, check the bundled README and tests; there are often small tutorials and pointers to further reading there.

JGit

If you want to use Git from within a Java program, there is a fully featured Git library called JGit. JGit is a relatively full-featured implementation of Git written natively in Java, and is widely used in the Java community. The JGit project is under the Eclipse umbrella, and its home can be found at ***http://www.eclipse.org/jgit***.

Getting Set Up

There are a number of ways to connect your project with JGit and start writing code against it. Probably the easiest is to use Maven – the integration is accomplished by adding the following snippet to the `<dependencies>` tag in your pom.xml file:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

The `version` will most likely have advanced by the time you read this; check ***http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit*** for updated repository information. Once this step is done, Maven will automatically acquire and use the JGit libraries that you'll need.

If you would rather manage the binary dependencies yourself, pre-built JGit binaries are available from ***http://www.eclipse.org/jgit/download***. You can build them into your project by running a command like this:

```
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
```

Plumbing

JGit has two basic levels of API: plumbing and porcelain. The terminology for these comes from Git itself, and JGit is divided into roughly the same kinds of areas: porcelain APIs are a friendly front-end for common user-level actions (the sorts of things a normal user would use the Git command-line tool for), while the plumbing APIs are for interacting with low-level repository objects directly.

The starting point for most JGit sessions is the `Repository` class, and the first thing you'll want to do is create an instance of it. For a filesystem-based repository (yes, JGit allows for other storage models), this is accomplished using `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

The builder has a fluent API for providing all the things it needs to find a Git repository, whether or not your program knows exactly where it's located. It can use environment variables (`.readEnvironment()`), start from a place in the working directory and search (`.setWorkTree(…).findGitDir()`), or just open a known `.git` directory as above.

Once you have a `Repository` instance, you can do all sorts of things with it. Here's a quick sampling:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
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");
```

There's quite a bit going on here, so let's go through it one section at a time.

The first line gets a pointer to the `master` reference. JGit automatically grabs the *actual* master ref, which lives at `refs/heads/master`, and returns an object that lets you fetch information about the reference. You can get the name (`.getName()`), and either the target object of a direct reference (`.getObjectId()`) or the reference pointed to by a symbolic ref (`.getTarget()`). Ref objects are also used to represent tag refs and objects, so you can ask if the tag is "peeled," meaning that it points to the final target of a (potentially long) string of tag objects.

The second line gets the target of the `master` reference, which is returned as an ObjectId instance. ObjectId represents the SHA-1 hash of an object, which might or might not exist in Git's object database. The third line is similar, but shows how JGit handles the rev-parse syntax (for more on this, see **"Гілкові посилання"**); you can pass any object specifier that Git understands, and JGit will return either a valid ObjectId for that object, or `null`.

The next two lines show how to load the raw contents of an object. In this example, we call `ObjectLoader.copyTo()` to stream the contents of the object directly to stdout, but ObjectLoader also has methods to read the type and size of an object, as well as return it as a byte array. For large objects

(where `.isLarge()` returns `true`), you can call `.openStream()` to get an InputStream-like object that can read the raw object data without pulling it all into memory at once.

The next few lines show what it takes to create a new branch. We create a RefUpdate instance, configure some parameters, and call `.update()` to trigger the change. Directly following this is the code to delete that same branch. Note that `.setForceUpdate(true)` is required for this to work; otherwise the `.delete()` call will return `REJECTED`, and nothing will happen.

The last example shows how to fetch the `user.name` value from the Git configuration files. This Config instance uses the repository we opened earlier for local configuration, but will automatically detect the global and system configuration files and read values from them as well.

This is only a small sampling of the full plumbing API; there are many more methods and classes available. Also not shown here is the way JGit handles errors, which is through the use of exceptions. JGit APIs sometimes throw standard Java exceptions (such as `IOException`), but there are a host of JGit-specific exception types that are provided as well (such as `NoRemoteRepositoryException`, `CorruptObjectException`, and `NoMergeBaseException`).

Porcelain

The plumbing APIs are rather complete, but it can be cumbersome to string them together to achieve common goals, like adding a file to the index, or making a new commit. JGit provides a higher-level set of APIs to help out with this, and the entry point to these APIs is the `Git` class:

```
Repository repo;
// construct repo...
Git git = new Git(repo);
```

The Git class has a nice set of high-level *builder*-style methods that can be used to construct some pretty complex behavior. Let's take a look at an example – doing something like `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw
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());
    }
```

This is a common pattern with the Git class; the methods return a command object that lets you chain method calls to set parameters, which are executed when you call `.call()`. In this case, we're asking the `origin` remote for tags, but not heads. Also notice the use of a `CredentialsProvider` object for authentication.

Many other commands are available through the Git class, including but not limited to add, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, and `reset`.

Further Reading

This is only a small sampling of JGit's full capabilities. If you're interested and want to learn more, here's where to look for information and inspiration:

- The official JGit API documentation is available online at ***http://download.eclipse.org/jgit/docs/latest/apidocs***. These are standard Javadoc, so your favorite JVM IDE will be able to install them locally, as well.
- The JGit Cookbook at ***https://github.com/centic9/jgit-cookbook*** has many examples of how to do specific tasks with JGit.
- There are several good resources pointed out at ***http://stackoverflow.com/questions/6861881***.

C Git Commands

Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

git config

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

In **"Початкове налаштування Git"** we used it to specify our name, email address and editor preference before we even got started using Git.

In **"Псевдоніми Git"** we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.

In “Перебазовування” we used it to make `--rebase` the default when you run `git pull`.

In “Збереження посвідчення (credential)” we used it to set up a default store for your HTTP passwords.

In “Keyword Expansion” we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of “Конфігурація Git” is dedicated to the command.

git help

The `git help` command is used to show you all the documentation shipped with Git about any command. While we're giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in “Отримання допомоги” and showed you how to use it to find more information about the `git shell` in “Налаштування Серверу”.

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

git init

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

We first introduce this in “Створення Git-репозіторія”, where we show creating a brand new repository to start working with.

We talk briefly about how you can change the default branch from “master” in “Віддалені гілки”.

We use this command to create an empty bare repository for a server in “Розміщення очищеного сховища на сервер”.

Finally, we go through some of the details of what it actually does behind the scenes in “Сантехніка та порцеляна”.

git clone

The `git clone` command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs `git init` to make it an empty Git repository, adds a remote (`git remote add`) to the URL that you pass it (by default named `origin`), runs a `git fetch` from that remote repository and then checks out the latest commit into your working directory with `git checkout`.

The `git clone` command is used in dozens of places throughout the book, but we'll just list a few interesting places.

It's basically introduced and explained in "Клонування існуючого репозиторію", where we go through a few examples.

In "Отримання Git на сервері" we look at using the `--bare` option to create a copy of a Git repository with no working directory.

In "Пакування" we use it to unbundle a bundled Git repository.

Finally, in "Клонування проекту з підмодулями" we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it's used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or "index") for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We'll quickly cover some of the unique uses that can be found.

We first introduce and explain `git add` in detail in "Контролювання нових файлів".

We mention how to use it to resolve merge conflicts in "Основи конфліктів зливання".

We go over using it to interactively stage only specific parts of a modified file in “Інтерактивне індексування”.

Finally, we emulate it at a low level in **“Tree Objects”**, so you can get an idea of what it’s doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In its normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in “Перевірка статусу ваших файлів”, both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

We first look at the basic uses of `git diff` in “Перегляд ваших доданих та недоданих змін”, where we show how to see what changes are staged and which are not yet staged.

We use it to look for possible whitespace issues before committing with the `--check` option in **“Commit Guidelines”**.

We see how to check the differences between branches more effectively with the `git diff A...B` syntax in **“Determining What Is Introduced”**.

We use it to filter out whitespace differences with `-b` and how to compare different stages of conflicted files with `--theirs`, `--ours` and `--base` in “Складне злиття”.

Finally, we use it to effectively compare submodule changes with `--submodule` in “Основи підмодулів”.

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in `git diff` command.

We only briefly mention this in “Перегляд ваших доданих та недоданих змін”.

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in “Збереження ваших змін у комітах”. There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.

In “Скасування речей” we cover using the `--amend` option to redo the most recent commit.

In “Гілки у кількох словах”, we go into much more detail about what `git commit` does and why it does it like that.

We looked at how to sign commits cryptographically with the `-S` flag in “Підписання комітів”.

Finally, we take a look at what the `git commit` command does in the background and how it’s actually implemented in “**Commit Objects**”.

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the HEAD pointer and optionally changes the `index` or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of `git reset` in “Вилучання файла з індексу”, where we use it to unstage a file we had run `git add` on.

We then cover it in quite some detail in “Усвідомлення скидання (reset)”, which is entirely devoted to explaining this command.

We use `git reset --hard` to abort a merge in “Припинення злиття”, where we also use `git merge --abort`, which is a bit of a wrapper for the `git reset` command.

git rm

The `git rm` command is used to remove files from the staging area and working directory for Git. It is similar to `git add` in that it stages a removal of a file for the next commit.

We cover the `git rm` command in some detail in **“Removing Files”**, including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.

The only other differing use of `git rm` in the book is in **“Removing Objects”** where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error out when the file we are trying to remove doesn’t exist. This can be useful for scripting purposes.

git mv

The `git mv` command is a thin convenience command to move a file and then run `git add` on the new file and `git rm` on the old file.

We only briefly mention this command in **“Пересування файлів”**.

git clean

The `git clean` command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files.

We cover many of the options and scenarios in which you might used the clean command in **“Очищення робочої директорії”**.

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

Most of **Chapter 3** is dedicated to the `branch` command and it’s used throughout the entire chapter. We first introduce it in **“Створення нової**

гілки” and we go through most of its other features (listing and deleting) in “Управління гілками”.

In “Відслідковувані гілки” we use the `git branch -u` option to set up a tracking branch.

Finally, we go through some of what it does in the background in **“Git References”**.

git checkout

The `git checkout` command is used to switch branches and check content out into your working directory.

We first encounter the command in “Переключення гілок” along with the `git branch` command.

We see how to use it to start tracking branches with the `--track` flag in “Відслідковувані гілки”.

We use it to reintroduce file conflicts with `--conflict=diff3` in “Отримання при конфліктах”.

We go into closer detail on its relationship with `git reset` in “Усвідомлення скидання (reset)”.

Finally, we go into some implementation detail in **“The HEAD”**.

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

The `git merge` command was first introduced in “Основи галуження”. Though it is used in various places in the book, there are very few variations of the `merge` command — generally just `git merge <branch>` with the name of the single branch you want to merge in.

We covered how to do a squashed merge (where Git merges the work but pretends like it’s just a new commit without recording the history of the branch you’re merging in) at the very end of **“Forked Public Project”**.

We went over a lot about the merge process and command, including the `-Xignore-space-change` command and the `--abort` flag to abort a problem merge in “Складне злиття”.

We learned how to verify signatures before merging if your project is using GPG signing in “Підписання комітів”.

Finally, we learned about Subtree merging in “Зливання піддерев”.

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in **"Основи конфліктів зливання"** and go into detail on how to implement your own external merge tool in **"External Merge and Diff Tools"**.

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

We introduce the command and cover it in some depth in **"Перегляд історії комітів"**. There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the `--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.

In **"Створення нової гілки"** we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.

In **"Private Small Team"** and **"Інтервали комітів"** we cover the `branchA..branchB` syntax to use the `git log` command to see what commits are unique to a branch relative to another branch. In **"Інтервали комітів"** we go through this fairly extensively.

In **"Журнал зливання"** and **"Потрійна крапка"** we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In **"Журнал зливання"** we also look at how to use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

In **"Скорочення reflog (журнал посилань)"** we use the `-g` option to view the Git reflog through this tool instead of doing branch traversal.

In **"Пошук"** we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In “Підписання комітів” we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in “Ховання та чищення”.

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in “Теґування” and we use it in practice in **“Tagging Your Releases”**.

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in “Підписання праці”.

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in “Отримання (`fetching`) та затягування (`pulling`) з ваших віддалених сховищ” and we continue to see examples of it use in “Віддалені гілки”.

We also use it in several of the examples in **“Contributing to a Project”**.

We use it to fetch a single specific reference that is outside of the default space in “Посилання (Refs) Запитів на Пул” and we see how to fetch from a bundle in “Пакування”.

We set up highly custom refspecs in order to make `git fetch` do something a little different than the default in **“The Refspec”**.

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you’re on.

We introduce it quickly in **“Отримання (fetching) та затягування (pulling) з ваших віддалених сховищ”** and show how to see what it will merge if you run it in **“Оглядання віддаленого сховища”**.

We also see how to use it to help with rebasing difficulties in **“Перебазовуйся коли перебазовуєшся”**.

We show how to use it with a URL to pull in changes in a one-off fashion in **“Checking Out Remote Branches”**.

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in **“Підписання комітів”**.

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in **“Надсилання змін до ваших віддалених сховищ”**. Here we cover the basics of pushing a branch to a remote repository. In **“Надсилання”** we go a little deeper into pushing specific branches and in **“Відслідковувані гілки”** we see how to set up tracking branches to automatically push to. In **“Видаляння віддалених гілок”** we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout **“Contributing to a Project”** we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in **“Розповсюдження теґів”**.

In **“Публікація змін з підмодуля”** we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In **“Other Client Hooks”** we talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in **“Pushing Refspecs”** we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in **“Взаємодія з віддаленими сховищами”**, including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project.

We use `git archive` to create a tarball of a project for sharing in **“Preparing a Release”**.

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in **“Підмодулі”**.

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in **“Анотовані теґи”**.

Later we use it quite a bit in **“Вибір ревізій”** to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in **“Повторне злиття файла вручну”** to extract specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in **“The Shortlog”**.

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It’s a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in **“Generating a Build Number”** and **“Preparing a Release”** to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in **“Двійковий пошук”** and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in **“Анотація файла”** and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in **“Git Grep”** and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you’re currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in **“Rebasing and Cherry Picking Workflows”**.

git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in **“Перебазовування”**, including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in **“Заміна”**, using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in **“Rerere”**.

We also use it in an interactive scripting mode with the `-i` option in **“Зміна декількох повідомлень комітів”**.

git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you’re targeting, essentially undoing or reverting it.

We use this in **“Вивертання коміту”** to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even GNU diff command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in **“Applying Patches from E-mail”**.

git am

The `git am` command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in **"Applying a Patch with am"** including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in **"E-mail Workflow Hooks"**.

We also use it to apply patch formatted GitHub Pull Request changes in **"Повідомлення електронною поштою"**.

git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in **"Public Project over E-Mail"**.

git imap-send

The `git imap-send` command uploads a mailbox generated with `git format-patch` into an IMAP drafts folder.

We go through an example of contributing to a project by sending patches with the `git imap-send` tool in **"Public Project over E-Mail"**.

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in **"Public Project over E-Mail"**.

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in **"Forked Public Project"**.

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in **"Git and Subversion"**.

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in **"A Custom Importer"**.

Administration

If you're administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs "garbage collection" on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in **"Maintenance"**.

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in **"Data Recovery"** to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in **"Скорочення reflog (журнал посилань)"**, where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in **"Data Recovery"**.

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In **"Вилучення файла з кожного коміту"** we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In **"Git-p4"** and **"TFS"** we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in **"Посилання (Refs) Запитів на Пул"** which we use to look at the raw references on the server.

We use `ls-files` in **"Повторне злиття файла вручну"**, **"Rerere"** and **"Індекс"** to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in **"Гілкові посилання"** to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in **Chapter 10**, which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.

Index

Symbols

$EDITOR, **400**
$VISUAL
    see $EDITOR, **400**
.gitignore, **402**
.NET, **568**
@{upstream}, **119**
@{u}, **119**
©, **564**
Гілки, 87

A

aliases, **85**
Apache, **149**
Apple, **568**
archiving, **418**
attributes, **412**
autocorrect, **403**

B

bash, **556**
binary files, **412**
BitKeeper, **34**
bitnami, **154**
branches
    basic workflow, **95**
    creating, **90**
    deleting remote, **120**
    diffing, **194**
    long-running, **106**
    managing, **105**
    merging, **100**
    remote, **110**, **193**
    switching, **91**
    topic, **108**, **189**
    tracking, **118**
    upstream, **118**
build numbers, **204**

C

C#, **568**
Cocoa, **568**
color, **404**
commit templates, **401**
contributing, **165**
    private managed team, **175**
    private small team, **168**
    public large project, **185**
    public small project, **181**
credential caching, **41**
credentials, **391**
CRLF, **41**
crlf, **409**

D

difftool, **405**
distributed git, **161**

E

Eclipse, **555**
editor
    changing default, **59**
email, **187**
    applying patches from, **190**
excludes, **402**, **504**

F

files
    moving, **63**
    removing, **61**
forking, **164**, **213**

G

Git as a client, **437**
git commands
  add, **51**, **51**, **52**
  am, **191**
  apply, **190**
  archive, **204**
  branch, **90**, **105**
  checkout, **91**
  cherry-pick, **200**
  clone, **48**
    bare, **141**
  commit, **59**, **88**
  config, **45**, **59**, **85**, **187**, **399**
  credential, **391**
  daemon, **148**
  describe, **204**
  diff, **56**
    check, **166**
  fast-import, **494**
  fetch, **77**
  fetch-pack, **530**
  filter-branch, **492**
  format-patch, **186**
  gitk, **547**
  gui, **547**
  help, **45**, **147**
  http-backend, **149**
  init, **47**, **51**
    bare, **142**, **145**
  instaweb, **152**
  log, **64**
  merge, **98**
    squash, **185**
  mergetool, **103**
  p4, **468**, **492**
  pull, **77**
  push, **77**, **83**, **116**
  rebase, **122**
  receive-pack, **528**
  remote, **74**, **76**, **78**, **79**
  request-pull, **182**
  rerere, **202**
  send-pack, **528**
  shortlog, **205**
  show, **82**
  show-ref, **441**
  status, **50**, **59**
  svn, **438**
  tag, **80**, **81**, **83**
  upload-pack, **530**
git-svn, **438**
git-tf, **476**
git-tfs, **476**
GitHub, **207**
  API, **259**
  Flow, **214**
  organizations, **250**
  pull requests, **217**
  user accounts, **207**
GitHub for Mac, **550**
GitHub for Windows, **550**
gitk, **547**
GitLab, **154**
GitWeb, **151**
GPG, **402**
Graphical tools, **547**
GUIs, **547**

H

hooks, **421**
  post-update, **137**

I

ignoring files, **54**
Importing
  from Mercurial, **488**
  from others, **494**
  from Perforce, **491**
  from Subversion, **486**
  from TFS, **493**
integrating work, **196**
Interoperation with other VCSs
  Mercurial, **449**
  Perforce, **458**
  Subversion, **437**
  TFS, **476**
IRC, **45**

J

java, **569**
jgit, **569**

K

keyword expansion, **415**

L

libgit2, **564**
line endings, **409**
Linus Torvalds, **34**
Linux, **34**
  installing, **40**
log filtering, **71**
log formatting, **67**

M

Mac
  installing, **40**
maintaining a project, **189**
master, **89**
Mercurial, **449**, **488**
mergetool, **405**
merging, **100**
  conflicts, **102**
  strategies, **420**
  vs. rebasing, **131**
Migrating to Git, **486**
Mono, **568**

O

Objective-C, **568**
origin, **111**

P

pager, **402**
Perforce, **458**, **491**
  Git Fusion, **459**
policy example, **424**
posh-git, **559**
Powershell, **41**
powershell, **559**
protocols
  dumb HTTP, **136**
  git, **139**
  local, **134**
  smart HTTP, **136**
  SSH, **138**
pulling, **120**
pushing, **116**
Python, **569**

R

rebasing, **121**
  perils of, **126**
  vs. merging, **131**
references
  remote, **110**
releasing, **204**
rerere, **202**
Ruby, **565**

S

serving repositories, **133**
  git protocol, **147**
  GitLab, **154**
  GitWeb, **151**
  HTTP, **149**
  SSH, **142**
SHA-1, **37**
shell prompts
  bash, **556**
  powershell, **559**
  zsh, **557**
SSH keys, **143**
  with GitHub, **208**
staging area
  skipping, **60**
Subversion, **162**, **437**, **486**

T

tab completion
  bash, **556**
  powershell, **559**
  zsh, **557**
tags, **80**, **202**
  annotated, **81**
  lightweight, **82**
  signing, **202**
TFS, **476**, **493**
TFVC (see TFS)

V

version control, **29**
  centralized, **31**
  distributed, **32**
  local, **29**
Visual Studio, **553**

W

whitespace, **408**
Windows
  installing, **41**

workflows, **161**
centralized, **162**
dictator and lieutenants, **164**
integration manager, **163**
merging, **196**
merging (large), **199**
rebasing and cherry-picking, **200**

X

Xcode, **40**

Z

zsh, **557**