THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Pro Git

Scott Chacon, Ben Straub

Table of Contents

Licence 1
Уводни думи от Scott Chacon 2
Уводни думи от Ben Straub 4
Посвещение 5
Сътрудници 6
Въведение 8
Начало 10
    За Version Control системите 10
    Кратка история на Git 14
    Какво е Git 14
    Конзолата на Git 18
    Инсталиране на Git 18
    Първоначална настройка на Git 22
    Помощна информация в Git 25
    Обобщение 26
Основи на Git 27
    Създаване на Git хранилище 27
    Запис на промени в хранилището 29
    Преглед на историята на действията 43
    Възстановяване на направени действия 49
    Работа с отдалечени хранилища 54
    Тагове в Git 59
    Псевдоними в Git 65
    Обобщение 67
Клонове в Git 68
    Накратко за разклоненията 68
    Основи на клоновете код и сливането 76
    Управление на клонове 85
    Стратегии за работа с клонове код 89
    Отдалечени клонове 92
    Пребазиране на клонове 102
    Обобщение 112
Git на сървъра 113
    Комуникационни протоколи 113
    Достъп до Git на сървъра 118
    Генериране на SSH публичен ключ 121
    Настройка на сървъра 122
    Git Daemon 125

- Smart HTTP ... 126
- GitWeb ... 128
- GitLab ... 130
- Други опции за хостване ... 135
- Обобщение ... 135

Git в разпределена среда ... 136
- Разпределени работни процеси ... 136
- Как да сътрудничим в проект ... 140
- Управление на проект ... 163
- Обобщение ... 177

GitHub ... 178
- Създаване и настройка на акаунт ... 178
- Как да сътрудничим в проект ... 183
- Управление на проект ... 203
- Управление на организация ... 218
- Автоматизиране с GitHub ... 221
- Обобщение ... 231

Git инструменти ... 232
- Избор на къмити ... 232
- Интерактивно индексиране ... 241
- Stashing и Cleaning ... 245
- Подписване на вашата работа ... 252
- Търсене ... 257
- Манипулация на историята ... 260
- Мистерията на командата Reset ... 269
- Сливане за напреднали ... 289
- Rerere ... 309
- Дебъгване с Git ... 316
- Подмодули ... 319
- Пакети в Git (Bundling) ... 342
- Заместване ... 346
- Credential Storage система ... 354
- Обобщение ... 359

Настройване на Git ... 360
- Git конфигурации ... 360
- Git атрибути ... 370
- Git Hooks ... 379
- Примерна Git-Enforced политика ... 383
- Обобщение ... 392

Git и други системи ... 393
- Git като клиент ... 393

Миграция към Git 432
Обобщение 451
Git на ниско ниво 452
Plumbing и Porcelain команди 452
Git обекти 453
Git референции 464
Packfiles 468
Refspec спецификации 471
Транспортни протоколи 474
Поддръжка и възстановяване на данни 480
Environment променливи 488
Обобщение 493
Appendix A: Git в други среди 495
Графични интерфейси 495
Git във Visual Studio 500
Git във Visual Studio Code 501
Git in IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine 502
Git в Sublime Text 502
Git в Bash 503
Git в Zsh 504
Git в PowerShell 506

Licence

Лизензирано според изискванията на лиценза Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported. Копие от лиценза може да се изтегли от https://creativecommons.org/licenses/by-nc-sa/3.0/ или чрез изпращане на писмо до Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Уводни думи от Scott Chacon

Добре дошли във второто издание на книгата Pro Git. Първата версия бе публикувана преди повече от 4 години. Оттогава насам се промениха много неща а също толкова важни неща не се промениха. Докато повечето основни команди и концепции са валидни и днес (понеже разработчиците на Git са просто фантастични в правенето на нещата обратно съвместими), то трябва да обърнем внимание на някои съществени нови неща и промени в общността обкръжаваща Git. Второто издание на тази книга има за задача да отрази промените и да опресни книгата, така че да е полезна за новия потребител.

Когато написах първото издание, Git беше все още сравнително трудна за използване система, освен за опитните хакери. Тогава тя започваше да набира популярност в определени общности от разработчици, но в никакъв случай не беше толкова вездесъща колкото е днес. Оттогава насам обаче, почти всяка open source общност я възприе и започна да я използва интензивно. Git бележи невероятен прогрес под Windows, броят на графичните Git интерфейси се увеличи експлозивно за всички платформи, както и поддръжката в IDE средите и цялостното използване в бизнеса. Pro Git в първото си издание отпреди 4 години не знае почти нищо за това. Една от основните задачи на това второ издание е да се докосне до всички тези нови граници в Git общността.

Open Source общността ползваща Git също се увеличи експлозивно. Когато първо седнах да пиша книгата преди 5 години (първата версия ми отне доста време), току що бях започнал работа в една много скромна компания разработваща Git хостинг и наричаща се GitHub. По времето на публикуването ѝ услугата се ползваше от няколко хиляди потребителя и само четирима от нас работеха по нея. Сега, когато пиша този увод, GitHub анонсира своя 10-милионен хостнат проект, с близо 5 млн. регистрирани акаунта на разработчици и над 230 служителя. Обичан или мразен, GitHub промени огромна част от open source общността по начин, какъвто трудно бихме могли да си представим по време на писането на първото издание на книгата.

В оригиналната версия на Pro Git аз написах малка секция за GitHub като пример за хостнато Git решение, с което никога не съм се чувствал комфортно. Не ми харесваше да описвам нещо, което възприемах като споделен ресурс и едновременно с това да говоря за моята компания в него. И макар и до днес да не обичам подобни конфликти на интерес, важността на GitHub в Git общността просто не може да бъде пропусната. Вместо като пример за Git хостинг, реших да превърна тази част от книгата в подробно описание на това какво е GitHub и как да го използваме ефективно. Ако започвате да учите как да използвате Git, тогава знанието за това как да използвате GitHub ще ви помогне да участвате в една огромна общност от разработчици - което е ценно знание без значение какво хостинг решение ще изберете за вашия собствен код.

Другата голяма промяна във времето от последната публикация е разработката и популяризирането на HTTP протокола за мрежовите транзакции на Git. Повечето примери в книгата бяха променени да ползват по-простичкия и интуитивен HTTP вместо SSH.

Беше невероятно да наблюдаваме как само за няколко години Git порасна от сравнително неизвестна система за контрол на версиите до практически доминиращо решение с отворен код в съвременния свят. Щастлив съм, че Pro Git се възприе толкова добре и че успя

едновременно да е една от малките технически ориентирани книги, които са едновременно успешни и с отворен код.

Надявам се тази обновена версия на Pro Git да ви хареса.

Уводни думи от Ben Straub

Първото издание на тази книга беше нещото, което ме свърза с Git. Това за мен беше илюстрация на съвсем нов начин за създаване на софтуер, който почувствах като по-естествен от всичко, което бях виждал преди. Дотогава бях разработчик в продължение на няколко години, но книгата ме накара да направя промяна и да поема по много по-интересен път.

Сега, години по-късно, аз съм сътрудник във важна Git имплементация, работих в най-голямата Git хостинг компания и обиколих света за да уча хората на Git. Когато Scott ме попита дали се интересувам да работя по второто издание, дори нямаше нужда да го обмисля.

За мен беше голямо удоволствие и привилегия да работя по тази книга. Надявам се тя да ви помогне така, както помогна на мен.

Посвещение

На съпругата ми Беки, без която това приключение не би започнало. — Бен

Това издание е посветено на момичетата ми. На съпругата ми Джесика, която ме подкрепяше през годините и на дъщеря ми Жозефин, която ще ме подкрепя, когато остарея дотолкова, че да не зная какво правя. — Скот

Сътрудници

Понеже това е книга с отворен код, през годините получавахме множество съобщения за печатни грешки или грешки в съдържанието. В списъка отдолу са всички хора допринесли за английския вариант на Pro Git като проект с отворен код. Благодарности към всички тях за помощта да направим книгата по-добра за всички

```
                                grgbnc                          peterwwillis
4wk-                            Guthrie McAfee Armstrong        Petr Bodnar
Adam Laflamme                   HairyFotr                       Petr Janeček
Adrien Ollier                   Hamidreza Mahdavipanah          Petr Kajzar
ajax333221                      haripetrov                      petsuter
Akrom K                         Haruo Nakayama                  Philippe Blain
Alan D. Salewski                Helmut K. C. Tessarek           Philippe Miossec
Alba Mendez                     Hidde de Vries                  Phil Mitchell
Aleh Suprunovich                HonkingGoose                    Rafi
Alexander Bezzubov              Howard                          rahrah
Alexandre Garnier               i-give-up                       Raphael R
alex-koziell                    Ignacy                          Ray Chen
Alfred Myers                    Ilker Cat                       Rex Kerr
allen joslin                    iprok                           Reza Ahmadi
Amanda Dillon                   Jan Groenewald                  Richard Hoyle
andreas                         Jaswinder Singh                 Ricky Senft
Andreas Bjørnestad              Jean-Noel Avila                 Rintze M. Zelle
Andrei Dascalu                  Jean-Noël Avila                 rmzelle
Andrew Layman                   Jeroen Oortwijn                 Rob Blanco
Andrew MacFie                   Jim Hill                        Robert P. Goldman
Andrew Metcalf                  jingsam                         Robert P. J. Day
Andrew Murphy                   Joel Davies                     Rohan D'Souza
AndyGee                         Johannes Dewender               Roman Kosenko
AnneTheAgile                    Johannes Schindelin             Ronald Wampler
Anthony Loiseau                 johnhar                         root
Antonello Piemonte              John Lin                        Rüdiger Herrmann
Antonino Ingargiola             Jon Forrest                     Sam Ford
Anton Trunov                    Jon Freed                       Sam Joseph
Ardavast Dayleryan              Jordan Hayashi                  Sanders Kleinfeld
atalakam                        Joris Valette                   sanders@oreilly.com
Atul Varma                      Josh Byster                     Sarah Schneider
axmbo                           Joshua Webb                     SATO Yusuke
Benjamin Dopplinger             Junjie Yuan                     Saurav Sachidanand
Ben Sima                        Justin Clift                    Scott Bronson
Borek Bernard                   Kaartic Sivaraam                Sean Head
Brett Cannon                    KatDwo                          Sebastian Krause
bripmccann                      Katrin Leinweber                Severino Lorilla Jr
brotherben                      Kausar Mehmood                  sharpiro
Buzut                           Keith Hill                      Shengbin Meng
Cadel Watson                    Kenneth Kin Lum                 Shi Yan
Carlos Martín Nieto            Kenneth Lum                     Siarhei Bobryk
Carlos Tafur                    Klaus Frank                     Siarhei Krukau
```

```
Chaitanya Gurrapu
Changwoo Park
Christoph Bachhuber
Christopher Wilson
Christoph Prokop
C Nguyen
CodingSpiderFox
Cory Donnelly
Cullen Rhodes
Cyril
Damien Tournoud
Daniele Tricoli
Daniel Shahaf
Daniel Sturm
Daniil Larionov
Danny Lin
Dan Schmidt
Davide Angelocola
David Rogers
delta4d
Denis Savitskiy
dependabot[bot]
devwebcl
Dexter
Dexter Morganov
DiamondeX
Dieter Ziller
Dino Karic
Dmitri Tikhonov
Dmitriy Smirnov
dualsky
Duncan Dean
Dustin Frank
Eden Hochbaum
Eric Henziger
evanderiel
Explorare
eyherabh
Ezra Buehler
Fabien-jrt
Felix Nehrke
Filip Kucharczyk
flip111
flyingzumwalt
Fornost461
franjozen
Frank
Frederico Mazzone
Frej Drejhammar
goekboet
Kristijan "Fremen" Velkovski
Krzysztof Szumny
Kyrylo Yatsenko
Lars Vogel
Laxman
Lazar95
Leonard Laszlo
Linus Heckemann
Logan Hasson
Louise Corrigan
Luc Morin
Lukas Röllin
maks
Marcin Sędłak-Jakubowski
Marie-Helene Burle
Marius Žilėnas
Markus KARG
Marti Bolivar
Mashrur Mia (Sa'ad)
Masood Fallahpoor
Mathieu Dubreuilh
Matthew Miner
Matthieu Moy
Máximo Cuadros
Michael MacAskill
Michael Sheaver
Michael Welch
Michiel van der Wulp
Mike Charles
Mike Pennisi
Mike Thibodeau
mmikeww
mosdalsvsocld
nicktime
Niels Widger
Niko Stotz
Nils Reuße
Olleg Samoylov
Osman Khwaja
Owen
Pablo Schläpfer
Pascal Berger
Pascal Borreli
patrick96
Patrick Steinhardt
paveljanik
Pavel Janík
Paweł Krupiński
pedrorijo91
Peter Kokot
Skyper
Smaug123
Snehal Shekatkar
Song Li
spacewander
Stephan van Maris
Steven Roddis
Stuart P. Bentley
SudarsanGP
Suhaib Mujahid
Susan Stevens
Sven Selberg
td2014
Thanix
Thomas Ackermann
Thomas Hartmann
Tomas Fiers
Tomoki Aonuma
Tom Schady
Tvirus
twekberg
Tyler Cipriani
Ud Yzr
uerdogan
UgmaDevelopment
un1versal
Vadim Markovtsev
Vangelis Katsikaros
Vegar Vikan
Ventsyslav Vassilev
Victor Ma
Vipul Kumar
Vitaly Kuznetsov
wencywww
Wesley Gonçalves
William Gathoye
William Turrell
Wlodek Bzyl
Xavier Bonaventura
xJom
xtreak
yakirwin
Yann Soubeyrand
Yue Lin Ho
Yunhai Luo
Yusuke SATO
zwPapEr
狄
```

Въведение

Предстои да отделите няколко часа от живота си четейки за Git. Нека обясним накратко какво ще ви предложим. Това е кратко резюме на съдържанието на десетте глави и трите приложения съставящи тази книга.

В **Глава 1** ще опишем принципите на Version Control системите (VCS) и основите на Git — само кратко описание без технически детайли, така че да разберете защо е създаден Git при положение, че вече е имало десетки други решения за контрол на версиите, какво го отличава и защо се използва толкова масово. След това ще обясним как изтеглите и инсталирате Git и да направите първоначалните настройки, ако все още го нямате в системата си.

В **Глава 2**, ще минем през основните действия в работата с Git, покриващи 80% от случаите, с които ще се сблъскате. В края на тази глава би трябвало да можете да клонирате хранилища (repositories), да проследявате какво се е случило в историята на даден проект, да променяте файлове и да предлагате за интеграция вашите промени в общия проект. Ако в този миг книгата ви по някакъв начин изчезне, то вие ще сте вече напреднали с Git и ще работите пълноценно във времето докато се сдобиете с ново копие.

Глава 3 е посветена на branching архитектурата на Git (клоновете код), често изтъквана като най-добрата му страна. Тук ще разберете какво наистина отличава Git от останалите подобни проекти. След като приключите, твърде е възможно да почувствате нужда от малко почивка, в която да осъзнаете как е било възможно да работите без branching-функциите на Git.

Глава 4 покрива Git от страна на сървъра. Тази глава е за тези, които искат да инсталират собствено Git решение в рамките на организацията или върху собствен сървър за съвместна работа с множество разработчици. Ще разгледаме и няколко hosted-опции, ако предпочитате някой друг да се грижи за тази задача.

Глава 5 ще премине детайлно през различни разпределени работни процеси и как да ги изпълним с Git. В края на главата ще можете да работите като истински експерт с множество отдалечени хранилища, да използвате Git през имейл и ловко да жонглирате с множество отдалечени клонове код (branches) и предложени поправки код (patches).

Глава 6 е посветена на хостнатата услуга GitHub и подробна информация за инструментите, които тя предлага. Разглеждаме създаването и управлението на акаунт, създаване и ползване на Git хранилища, стандартните процедури за подпомагане с код на външни проекти (както и как да приемате контрибуции за вашите собствени такива), програмния интерфейс на GitHub и много други ценни съвети, които ще ви улеснят работата като цяло.

Глава 7 е посветена на някои по-сложни Git команди. Ще научите за неща като mastering и плашещата *reset* команда, за ползване на двоично търсене за намиране на грешки, историята на промените, детайлно обяснение на избора на версии и много други тънкости. Тази глава ще ви даде допълнителните знания, необходими за да станете истински Git експерти.

Глава 8 разглежда конфигурацията на вашата Git система. Това включва създаването на

hook-скриптове за да наложите или окуражите спазването на собствени политики и използването на настройки за конфигурация на работната среда - така че да можете да работите по начина, по който вие искате. Ще разгледаме и изграждането на ваши собствени колекции от скриптове, с чиято помощ да наложите желаните от вас правила за публикуване (commit) на код.

Глава 9 е за взаимодействието на Git с други VCS системи. Това включва използването на Git в Subversion (SVN) света и конвертирането на проекти от други VCS системи към Git. Множество организации все още ползват SVN и е малко вероятно това да се промени скоро. Но към този момент вие ще сте завладяни от невероятната мощ на Git и тази глава ще ви покаже как да процедирате ако трябва да си имате работа с SVN сървър. Разглеждаме също и как да импортираме проекти от няколко други системи в случай, че успеете да убедите всички да направят прехода.

Глава 10 се потапя в тъмните (и същевременно очарователни) дълбини на Git. След като вече знаето всичко за Git и неговата сила, време е да разберете още повече детайли - как Git съхранява своите обекти и какъв е техния модел, детайли за pack-файловете, протоколи на сървъра и др. Вътре в книгата, ще се обръщаме към секции от тази глава в случай, че пожелаете да научите какво се случва под повърхността. Разбира се, ако сте като нас и техническите детайли са на първо място за вас - можете да започнете книгата прочитайки първо Глава 10. Изборът е изцяло ваш.

В **Приложение A** даваме множество примери за ползване на Git в специфични условия. Разглеждаме няколко различни GUI приложения и IDE среди, в които бихте искали да използвате Git и какво можете да правите в тях. Ако искате резюме за ползването на Git в шела, във Visual Studio или в Eclipse - погледнете тук.

В **Приложение B** изследваме възможностите за автоматизация и разширение на Git чрез инструменти като libgit2 и JGit. Ако се интересувате от писане на сложни и бързи специализирани инструменти и се нуждаете от достъп до Git на ниско ниво, това е мястото където може да видите как стоят нещата.

Накрая, в **Приложение C**, преминаваме през всички основни Git команди една по една и разглеждаме къде в книгата сме ги обяснили и какво сме направили с тях. Ако искате да знаете къде в книгата сме ползвали всяка една Git команда, можете да потърсите тук.

Да започваме!

Начало

Тази глава ще ви помогне да започнете с Git. Ще стартираме с малко история на инструментите за контрол на версиите, след това ще продължим с инсталирането на Git във вашата система и накрая ще видим как да започнем да работим с нея. В края на тази глава би следвало да сте наясно защо е създаден Git, защо е добре да го използвате и как да започнете

За Version Control системите

Какво е “version control” и защо ви интересува? Системата за контрол на версиите е средство за проследяване и запис на промените по файл (или множество файлове) във времето, така че да можете да възстановите произволна версия на файла/файловете във всеки един момент по-късно. За примерите в тази книга, вие ще използвате софтуерен код под формата на файлове, чиито версии ще бъдат контролирани, макар в действителност да можете да правите това с произволен тип файл във вашия компютър.

Ако сте графичен или уеб дизайнер и желаете да пазите всяка версия на дадено изображение или layout (а е доста вероятно да искате), то Version Control системата (VCS) е мъдър избор за вас. Тя ви позволява да възстановите файловете си обратно към определен момент от миналото, да възстановите цял проект към даден предишен момент, да сравнявате промените във времето, да видите кой последно е променил нещо, което би могло да предизвика проблем, кой и кога е съобщил за грешка и т.н. Използването на VCS също така означава, че ако случайно объркате нещата и неволно загубите файлове - можете лесно да ги възстановите. Не на последно място - получавате всичко това на цената на съвсем малко изразходвани ресурси.

Локални Version Control Systems

Много хорат предпочитат да реализират контрол на версиите като просто копират файловете в друга директория (ако са достатъчно изобретателни - посочвайки дата и час в името ѝ). Този начин е често използван защото е много прост, но и също толкова податлив на грешки. Много лесно е да забравите в коя директория сте в момента и неволно да запишете в грешен файл или пък да копирате погрешни файлове не където трябва.

За да се справят с това неудобство, програмистите отдавна разбработиха локални VCS с прости бази данни, които съхраняваха промените по файловете, които се проследяват.

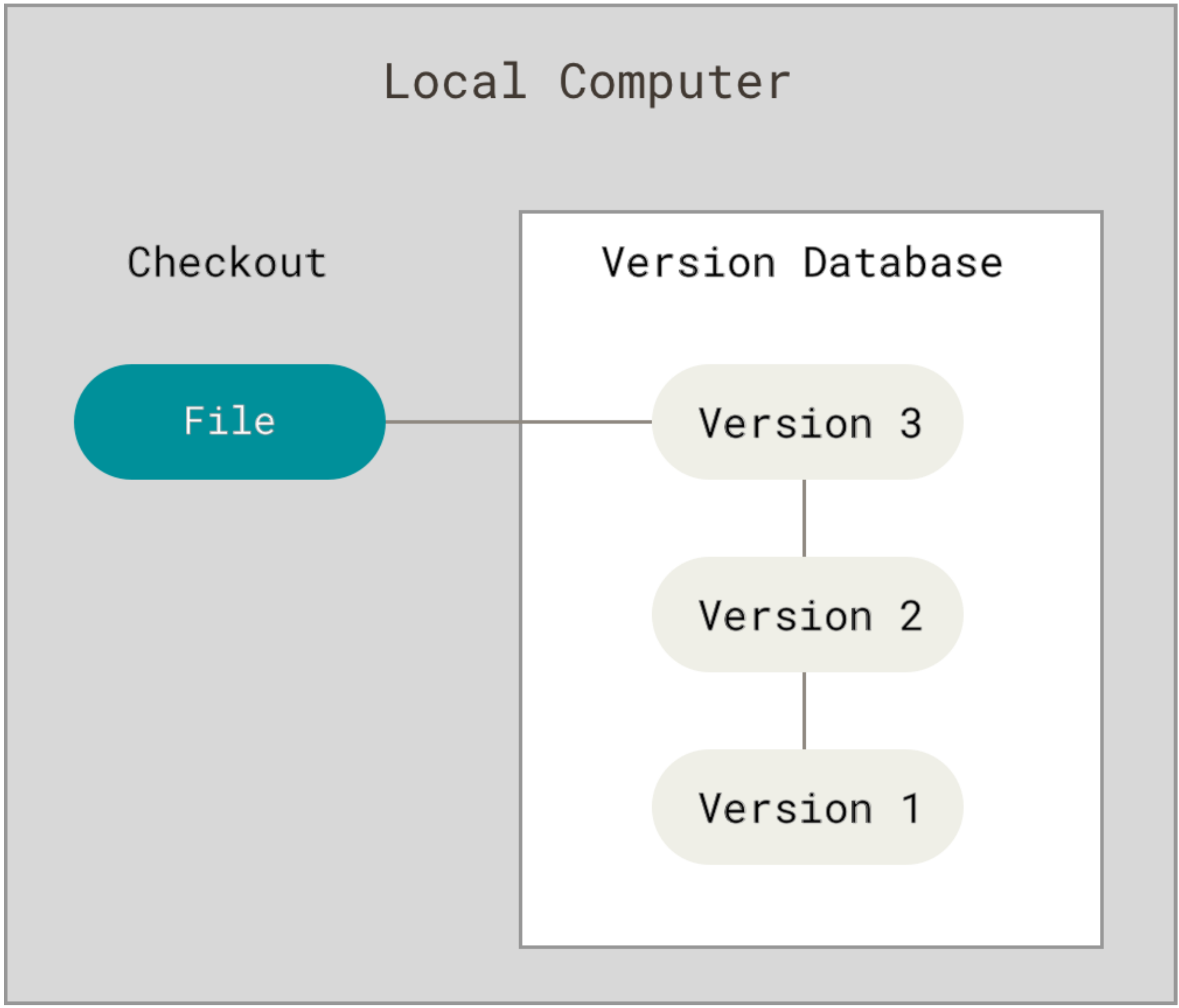


Figure 1. Local version control

Една от най-популярните подобни системи се нарича RCS и все още се разпространява с много компютри и до днес. RCS работи съхранявайки множество от пачове (разликите във файловете) в специален формат на диска и може да възстанови състоянието на файла към произволен момент добавяйки всички тези пачове.

Централизирани Version Control Systems

Следващото голямо предизвикателство пред разработчиците дойде в момента, когато се оказа, че те трябва да работят съвместно с други такива, на други компютри. За да се справят с това, бяха разработени централизираните Version Control системи (CVCS). Тези системи - CVS, Subversion, Preforce, използват един сървър, съдържащ всички контролирани файлове и множество от клиенти, които издърпват файловете от това централно място. В продължение на много години това беше стандарт за контрол на версиите.

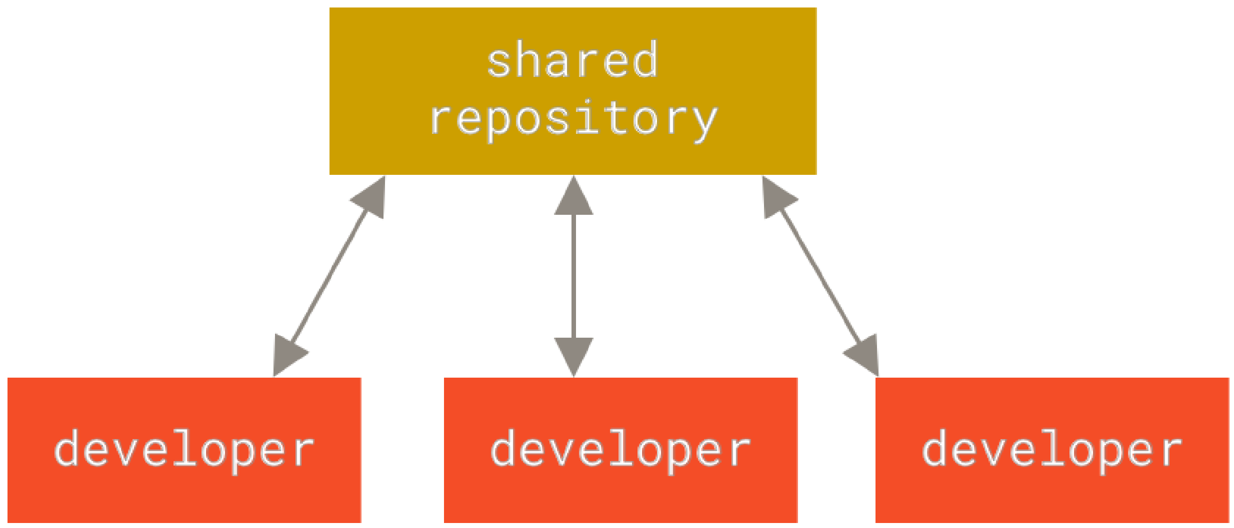


Figure 2. Centralized version control

Този подход предлага много предимства, особено спрямо локалните VCS. Например, всеки участник в проекта е запознат в доста добра степен на достоверност какво друг е правил по него. Администраторите имат подробен контрол върху това кой какво може да прави и освен това е много по-лесно да се администрира CVCS отколкото локалните бази данни на всеки клиент.

Обаче, този подход си има и сериозни недостатъци. Най-очевидният от тях е, че всички данни се пазят на едно място и зависят от надеждността на сървъра, който ги съхранява. Ако сървърът по някаква причина не е достъпен за час - тогава през този час никой не може да качва своите промени. Ако, в най-лошия случай, се повреди диск на сървъра и не разполагате с адекватни архиви - губите абсолютно всичко, цялата история на проекта, освен ако някой потребител няма snapshot на проекта на локалния си компютър. Локалните VCS са изложени на същия риск — ако цялата история на проекта се пази само на едно място, рискувате да го загубите изцяло.

Разпределени Version Control Systems

Тук е мястото, където се намесват разпределените (Distributed) Version Control системи (DVCS). В DVCS (каквито са Git, Mercurial, Bazaar или Darcs), клиентите не просто изтеглят последния snapshot на файловете - те изцяло клонират цялото хранилище, включително пълната му история. По този начин, ако сървърът загине, хранилището на даден проект може да се възстанови от локалното копие на всеки клиент. Всяко копие по същество е пълен архив на всички данни.

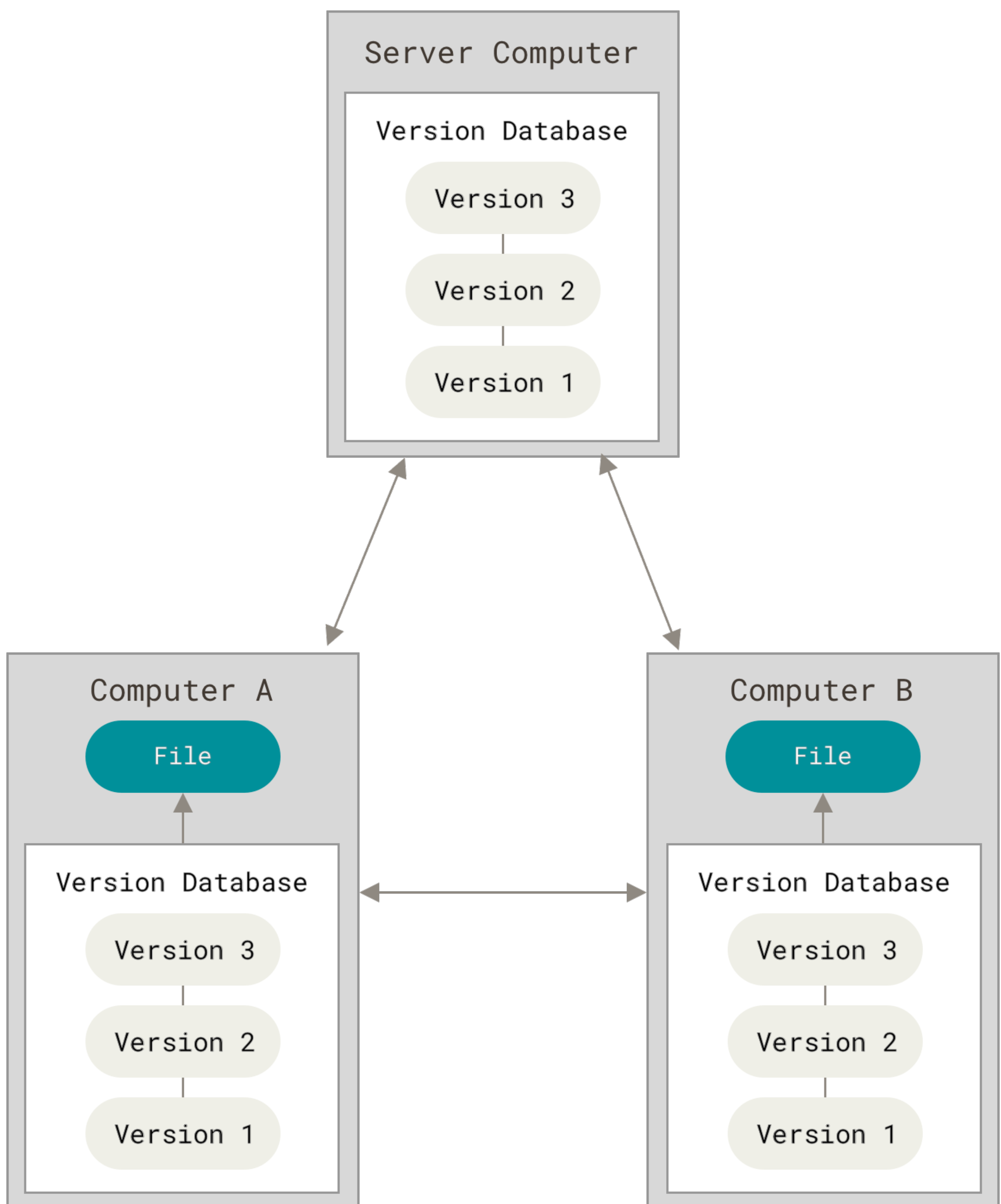


Figure 3. Distributed version control

В допълнение, много от тези системи се справят доста добре със задачата да могат да обслужват няколко отдалечени хранилища, така че можете да си сътрудничите с различни групи от разработчици едновременно, в рамките на един и същи проект. Това позволява да създадете няколко различни работни потока, което е невъзможно в централизираните системи като йерархични модели.

Кратка история на Git

Като много велики неща в живота, Git започва с малко креативна разрушителност и ожесточени спорове.

Linux ядрото е open source софтуерен проект от сравнително голям мащаб. За дълго време от поддръжката на проекта (1991-2002), промените в софтуера се изпращаха под формата на пачове и архивирани файлове. През 2002 г., Linux kernel проектът започна да използва патентована DVCS система наречена BitKeeper.

През 2005, връзката между общността разработваща Linux ядрото и комерсиалната компания стояща зад BitKeeper се разпадна, след което инструментариума на BitKeeper вече не можеше да се ползва свободно. Това провокира общността от разработчици на Linux (и по-специално създателя му Линус Торвалдс) да разработи свой собствен инструмент за version control, базиран на част от знанията натрупани по време на използването на BitKeeper. Част от поставените цели и изисквания на новата система бяха както следва:

- Скорост
- Опростен дизайн
- Мощна поддръжка за нелинейна разработка (хиляди паралелни клонове код)
- Напълно разпределена работа
- Възможност да облужва ефективно големи проекти като Linux ядрото (по отношение на скорост и обем на данните)

От създаването си през 2005 г. Git еволюира и узря така, че да е лесна за ползване и едновременно с това да поддържа първоначалните си цели описани по-горе. Git е впечатляващо бърза, ефективна с големи проекти и разполага с невероятна branching система за нелинейна разработка (виж Клонове в Git)

Какво е Git

И така, какво е Git? Това е важна за усвояване секция, защото ако разберете какво е Git и какви са основите, върху които работи, тогава ползването му по един ефективен начин ще ви е доста по-лесно и приятно. Изучавайки Git, опитайте се да се абстрахирате от нещата, които вече знаете за други VCS системи като Subversion или Perforce, това ще ви спести някои недоразумения и трудности. Git съхранява и разглежда информацията съвсем различно в сравнение с другите системи, макар потребителският ѝ интерфейс да е подобен с техните. Разберете ли разликите, ще ви е по-лесно да работите с Git.

Snapshot-и вместо разлики

Основната разлика между Git и другите системи за контрол на версиите е начинът, по който Git третира данните. Концептуално, повечето други системи записват информацията като списък от файлово-базирани промени. Тези системи (CVS, Subversion, Perforce, Bazaar) виждат информацията, която съхраняват като колекция от файлове и промените направени във файловете във времето (известно още като *delta-based* version control).

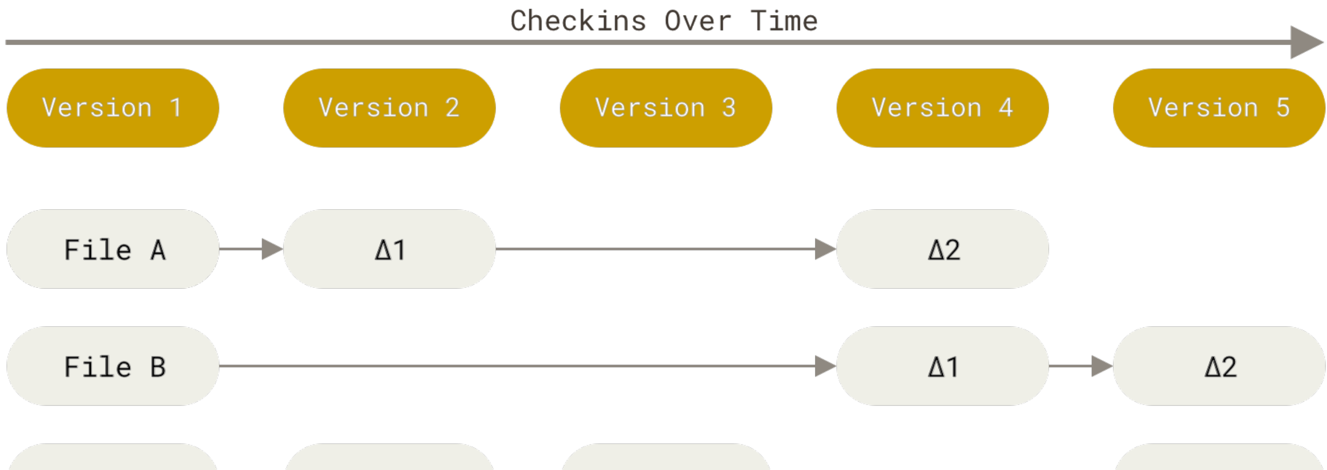


Figure 4. Съхраняване на данните като списък от промени в базовата версия на всеки от файловете

Git обаче не третира информацията така. Вместо това, Git възприема своите данни по-скоро като колекция от snapshots (моментни снимки на статуса) на една миниатюрна файлова система. Всеки път, когато къмитвате или записвате статуса на вашия проект в Git, системата най-общо казано прави снимка на това как изглеждат файловете ви в този момент и съхранява референция към този snapshot. За ефективност, ако някои файлове не са променени, Git не ги записва отново, а просто записва указател към предишния идентичен файл, който вече е съхранил. Така че - Git възприема своите данни като **поток от snapshot-и**.

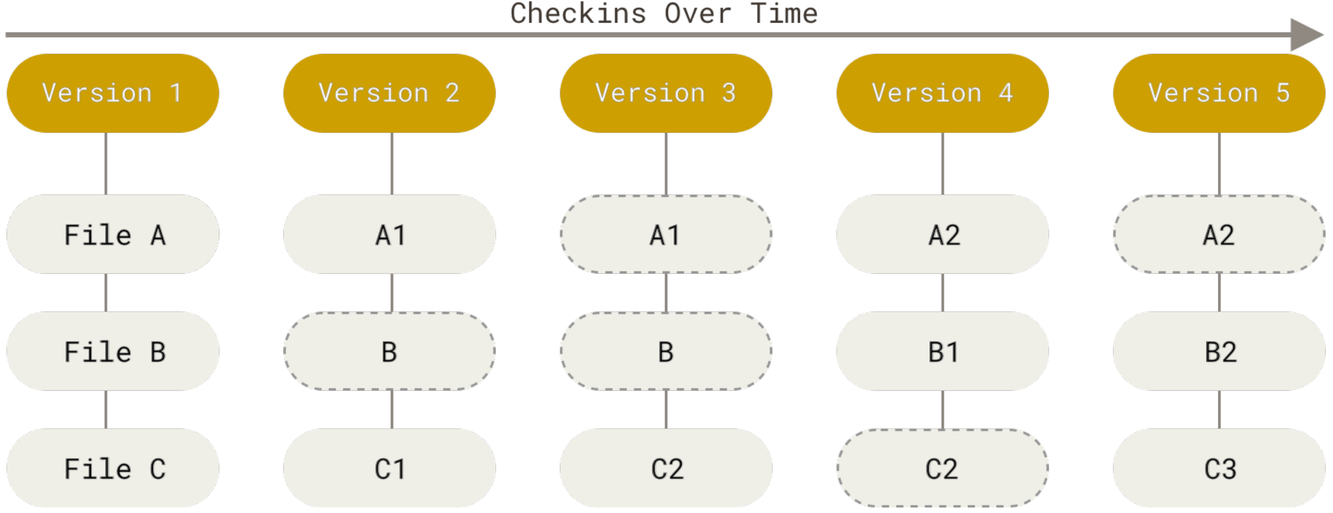


Figure 5. Записане на данните като поток от snapshot-и във времето

Това е важна разлика между Git и почти всички други VCS. Git променя почти всеки един аспект на контрола на версиите, работейки подобно на малка файлова система с мощни инструменти, вместо да бъде просто VCS. Ще разгледаме някои от предимствата, които дава тази концепция в главата посветена на клоновете код - Клонове в Git.

Почти всяка операция е локална

Повечето операции в Git се нуждаят само от локални файлове и ресурси - общо взето не се нуждаете от информация намираща се в мрежата. Ако сте ползвали CVCS, където производителността на повечето операции зависи от мрежата, ще си помислите, че

боговете на скоростта са благословили Git с извънземни сили. Понеже разполагате с цялата история на проекта директно на диска, повечето операции изглеждат почти светкавични.

Например, за да ви покаже историята на проекта, Git не се нуждае да контактува със сървъра а просто чете директно от локалната си база данни и вие виждате историята почти незабавно. Ако желаете да видите промените в даден файл между текущата му версия и тази отпреди месец, Git ще направи локална калкулация на разликите, вместо да трябва да пита отдалечения сървър за това или да трябва да издърпва по-стара версия от сървъра и едва след това да калкулира разликите локално.

Това също значи, че можете да правите почти всичко когато сте офлайн или когато VPN връзката ви не работи например. Ако сте в самолет или влак и искате да свършите малко работа, можете спокойно да къмитвате промените си (в *локалното* си копие, нали запомнихте?) и когато имате мрежова връзка - да ги качите на сървър. Ако сте вкъщи и VPN клиентът ви не работи - вие можете да продължите работата си. В много други системи това е трудно или невъзможно. В Perforce например, не можете да направите много офлайн, а в Subversion и CVS можете да редактирате файлове, но не можете да къмитвате промените към базата данни (защото тя не е достъпна офлайн). Това може да не изглежда толкова важно, но ще се изненадате каква разлика в стила ви на работа може да направи.

Git има интегритет

Всичко в Git се проследява с чексуми преди да се запише и след това обръщенията към съдържанието стават чрез чек-сумите. Това означава, че е невъзможно да промените съдържанието на файл или директория без Git да знае за това. Тази функционалност е вградена в Git на най-ниско ниво и е интегрална част от философията на системата. Вие не можете да загубите информация в етапите на работа или да повредите файл без Git да го усети.

Механизмът, който Git използва за чек-сумите се нарича SHA-1 хеш. Това е 40-символен стринг съставен от шестнадесетични числа (0–9 и a–f) и калкулиран на базата на съдържанието на файл или структура на директория в Git. Един SHA-1 хеш може да изглежда подобно на това:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Ще срещате тези хешове почти навсякъде, защото Git ги ползва повсеместно. В действителност, Git пази обектите в базата си данни не по имена на файлове, а посредством хешираните стрингове отразяващи съдържанието им.

Git само добавя данни

Когато правите нещо в Git, тези действия само *добавят* информация към базата данни на Git. Трудно е да накарате системата да направи каквото и да е без то да може да бъде възстановено или пък да изтриете данни безвъзвратно. Разбира се, подобно на всяка друга VCS, вие можете да загубите или объркате промените, които не са били къмитнати, но веднъж направили snapshot-а в Git - е много трудно да загубите данни, особено пък ако редовно изпращате базата си към отдалечено хранилище.

Това е хубавата страна на Git - знаем, че можем да експериментираме без опасност от тежки последствия. За повече информация относно това как Git съхранява своите данни и как можете да възстановите информация, която изглежда загубена, погледнете Възстановяване на направени действия.

Сега внимавайте! Това е най-важното, което трябва да помните за Git, ако искате да научите системата лесно. Git пази файловете ви в три възможни състояния: *modified*, *staged* и *committed*:

- Променен (modified) означава, че сте променили файла, но той все още не е къмитнат в локалната база данни.
- Статусът индексиран (staged), означава че сте маркирали променен файл в текущия му вариант, указвайки на Git да го включи в следващия къмитнат snapshot.
- Къмитнат (commited) означава, че данните са надеждно записани в локалната ви база данни.

Това ни довежда до трите главни секции на един Git проект: работната област, staging областта и Git директорията.

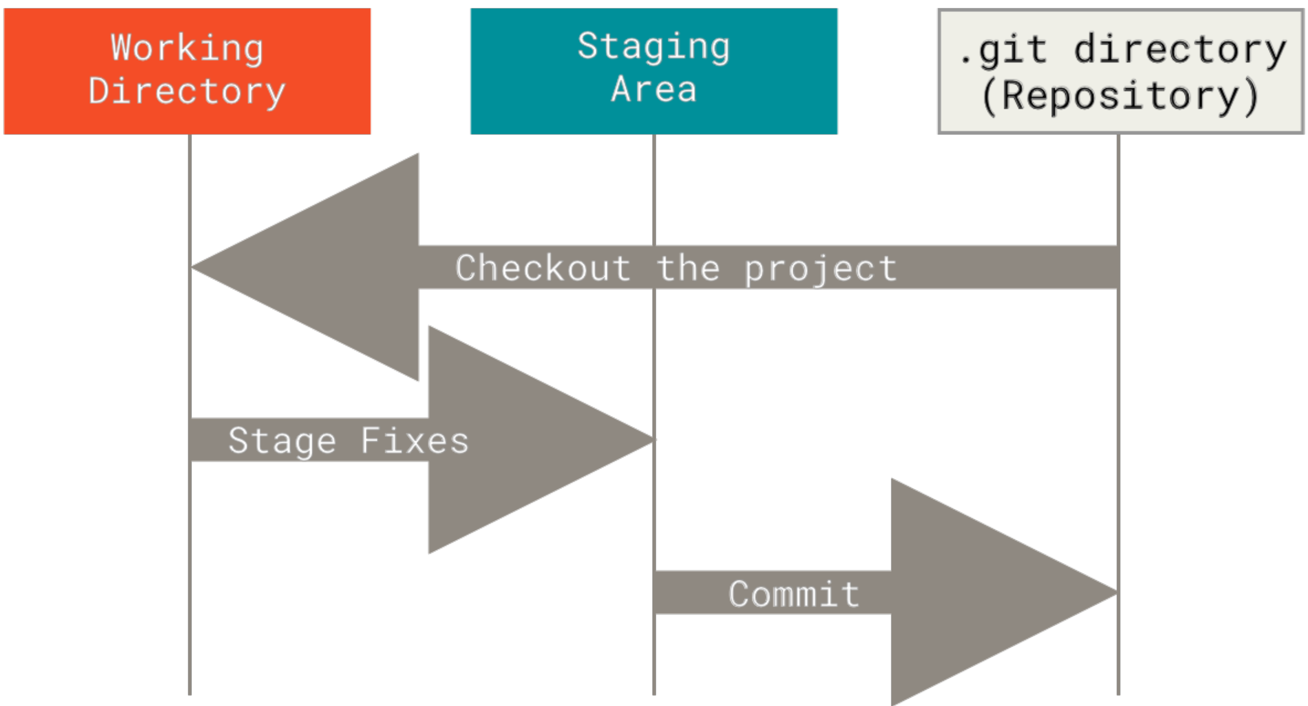


Figure 6. Работна област, staging област и Git директория

Работната област е моментното състояние на дадена версия от проекта. Файловете в тази област се издърпват от компресираната база в Git директорията и се поставят на диска за да можете да ги променяте.

Staging областта представлява файл, който се пази в Git директорията и който пази информация за това какво ще отиде в следващия ви къмит. Понякога тази област бива наричана също и "индекс", но по-често се нарича staging area.

Git директорията е мястото, където Git пази своите мета-данни и базата данни с обектите за вашия проект. Това е най-важната част на Git и е това, което се копира когато *клонирате*

хранилище от отдалечен компютър.

Един типичен Git работен процес изглежда по подобен начин:

1. Вие променяте файлове в работната област.
2. Вие селективно индексирате (stage-вате) само промените, които искате да са част от следващия ви къмит.
3. Вие правите къмит, при което файловете се вземат така както изглеждат в staging областта и този snapshot се записва за постоянно във вашата Git директория.

Ако дадена версия на определен файл се намира в Git директорията, тази версия се счита за *къмитната*. Ако файлът е променен и добавен в staging областта - той се разглежда като *staged/индексиран*. И ако файлът е променен откакто е издърпан, но не е бил индексиран - то той се счита за *променен*. В Основи на Git, ще научите повече за тези статуси на файловете ви и как да се възползвате от тях или пък директно да пропуснете staged-частта.

Конзолата на Git

Съществуват различни начини за ползване на Git. Налице са традиционните инструменти от командния ред, а също така и многобройни графични инструменти с различни възможности. За целите на тази книга ще използваме Git от конзола. Командният ред е единственото място, където можете да ползвате *всички* Git команди - повечето GUI имплементират само част от пълната Git функционалност за по-просто. Ако знаете как да използвате командния ред, вероятно лесно ще разберете как да ползвате и GUI инструмент, докато обратното не е непременно вярно. Също така, докато изборът на графичен клиент е въпрос на персонални предпочитания, то *всички* потребители ще имат инсталирани и готови за работа командите от конзолата.

Затова ще приемем, че знаете как да отворите терминал в macOS, Command Prompt или PowerShell под Windows. Ако се чудите за какво разговаряме, по-добре е да спрете с книгата дотук, да си опресните знанията за конзолата на операционната система, която ползвате и след това да продължите четенето. Така ще можете по-лесно да следвате примерите и описанията в следващите глави.

Инсталиране на Git

Преди да започнете да ползвате Git, трябва да го инсталирате на компютъра си. Дори ако вече е инсталиран, добра идея е да обновите до последната версия. Инсталацията става като пакет, чрез друг инсталатор или чрез компилиране на изходния код.

Тази книга е написана за Git версия **2.8.0**. Въпреки че повечето команди, които използваме трябва да работят дори в много стари версии на Git, някои от тях може да работят с леки разлики, ако вашата версия е по-стара. Понеже Git е достатъчно добър в поддържането на обратната съвместимост, всяка версия след 2.8 би трябвало да работи добре.

Инсталация в Linux

Ако искате да инсталирате основните Git инструменти под Linux с binary инсталатор, в общия случай това е лесно с пакетните инструменти на вашата дистибуция. Например под Fedora (или всяка друга подобна, RPM-базирана дистрибуция като RHEL или CentOS), можете да ползвате `dnf`:

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

Ако предпочитате Debian-базирана дистрибуция като Ubuntu, опитайте с `apt`:

```
$ sudo apt install git-all
```

За повече подробности и опции касаещи инсталацията в Linux, вижте сайта на Git: https://git-scm.com/download/linux.

Инсталация в macOS

Има няколко начина за инсталиране на Git в Mac. Може би най-лесният е да инсталирате Xcode Command Line Tools. Под Mavericks (10.9) и по-новите версии, можете да направите това просто като опитате да изпълните команда *git* в терминала първия път.

```
$ git --version
```

Ако не сте го инсталирали вече, системата ще ви предложи да го направите.

Ако желаете по-актуална версия, можете да я инсталирате и през binary инсталатор. OSX Git инсталатор за MacOS се поддържа и може да се изтегли от https://git-scm.com/download/mac.

Figure 7. Git macOS Installer

Можете да инсталирате Git и като част от софтуера GitHub for macOS. Техният GUI Git инструмент има опция за инсталация и на конзолните команди. Можете да изтеглите GitHub for macOS от https://desktop.github.com.

Инсталация в Windows

И тук има няколко опции да инсталирате Git. Официалната версия е налична за сваляне от сайта на Git. Отворете https://git-scm.com/download/win и изтеглянето ще започне автоматично. Имайте предвид, че това е проект наречен Git for Windows, който е отделен от самия Git, за повече информация за него, посетете https://gitforwindows.org.

Ако искате автоматизирана инсталация, можете да използвате Git Chocolatey package. Chocolatey пакетът се поддържа от общност доброволци.

Следващата възможност е да инсталирате GitHub Desktop. Инсталаторът включва конзолните инструменти заедно с графичните. Той също така работи добре с PowerShell и настройва надеждно credential кеширането и CRLF настройките. Ще научим за тях по-късно - засега приемаме, че това са неща, които ви трябват. GitHub Desktop е наличен за сваляне от сайта GitHub Desktop website.

Инсталация от сорс-код

Някои хора предпочитат да инсталират Git от изходния код, защото по този начин

получават възможно най-актуалната версия. Бинарните инсталатори обикновено са за една идея по-стари версии, макар че това не е толкова важно, защото Git е много съвместим.

Ако искате да инсталирате Git от изходен код, ще се нуждаете от библиотеките autotools, curl, zlib, openssl, expat, и libiconv, понеже Git зависи от тях. За Fedora или Debian-базирана дистрибуция, изпълнете долните команди съответно, така че да се сдобиете с минималните изисквания за компилация и инсталиране на Git:

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

За да можете да добавите документацията в различни формати (doc, html, info), са необходими допълнителните зависимости отдолу:

```
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

Потребителите на RHEL и RHEL деривати като CentOS и Scientific Linux трябва да разрешат EPEL хранилището за да изтеглят пакета `docbook2X`.

Ако използвате Debian-базирана дистрибуция (Debian/Ubuntu/Ubuntu-варианти), ще се нуждаете също и от пакета `install-info`:

```
$ sudo apt-get install install-info
```

Ако използвате RPM дистрибуция (Fedora/RHEL/RHEL-деривати), ще ви трябва пакета `getopt` (който е наличен по подразбиране в Debian-базираните дистрибуции):

```
$ sudo dnf install getopt
```

Освен това, ако ползвате Fedora/RHEL/RHEL-деривати, трябва да изпълните това:

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

поради различия в имената на двоичните файлове.

След като се уверите, че имате инсталирани всички зависимости, продължавате напред и изтегляте най-новия архив с изходен код на Git. Това може да стане от няколко места - сайта Kernel.org на адрес https://www.kernel.org/pub/software/scm/git, или от хранилището в GitHub - https://github.com/git/git/releases. Обикновено страницата в GitHub би следвало да е по-актуална, но и Kernel.org също разполага с контролни сигнатури, ако желаете да проверите какво сте изтеглили.

Следва компилация и инсталиране:

```
$ tar -zxf git-2.8.0.tar.gz
$ cd git-2.8.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

След като направите това, можете да изтеглите Git от самия Git, за обновявания:

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

Първоначална настройка на Git

Сега, когато имате Git в системата си, е добре да направите няколко неща за да настроите Git обкръжението си. Това се прави само веднъж на даден компютър и се съхранява между обновленията. Можете също по всяко време да промените настройките изпълнявайки командите отново.

Git пристига с инструмент наречен `git config`, който позволява да извличате и променяте конфигурационните променливи, които контролират всички аспекти на това как изглежда и работи Git. Тези променливи може да се пазят в три различни места:

1. `[path]/etc/gitconfig` файла: Съдържа стойности за всеки потребител в системата и всички техни хранилища с код. Ако изпълните командата `git config` с параметър `--system`, тя чете и пише в този файл. Понеже това е системен конфигурационен файл, ще се нуждаете от административни права за да го променяте.
2. `~/.gitconfig` или `~/.config/git/config` файл: Специфичен само за конкретния потребител. Git чете и пише в този файл ако горната команда е изпълнена с параметъра `--global` и това засяга *всички* хранилища, с които работите на дадена система.
3. `config` файл в конкретна Git директория (`.git/config`) в произволно хранилище, което ползвате: Съдържа настройки само за това конкретно хранилище. Може да накарате Git да чете и пише в този файл с опцията `--local`, но това всъщност се прави по подразбиране. Разбира се, трябва да сте в директория с Git хранилище, за да работи това коректно.

Всяко ниво от горните е с приоритет пред предишното, така че стойностите в `.git/config` се използват вместо аналогичните в `[path]/etc/gitconfig`.

В Windows системите, Git търси файла `.gitconfig` в `$HOME` директорията (`C:\Users\$USER` в повечето случаи). Git все пак търси за `[path]/etc/gitconfig`, въпреки че тя е релативна към MSys главната директория, която се определя от това къде сте указали на инсталатора да инсталира Git. Ако ползвате версия 2.x или по-нова на Git for Windows, съществува и конфигурационен файл на системно ниво, който се намира в `C:\Documents and Settings\All Users\Application Data\Git\config` при Windows XP, и в `C:\ProgramData\Git\config` при Windows

Vista и по-новите. Този конфигурационен файл може да се сменя само с командата `git config -f <file>` изпълнена от администраторски акаунт.

Можете да видите всички ваши настройки и откъде идват те така:

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

Вашата идентичност

Първото нещо, което трябва да направите след инсталацията е да зададете вашето потребителско име и имейл адрес. Това е важно, защото всеки Git къмит използва тези данни и те ще са неделима част в следващите къмити:

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

Отново, това се прави само веднъж ако зададете параметъра `--global` и после Git винаги ще ползва тази информация за всичко, което правите на този компютър. Ако искате за конкретен проект да ползвате различно потребителско име/имейл - можете да пуснете командата без --*global* параметъра вътре в самия проект.

Много от GUI инструментите ще ви помогнат с това първия път, когато ги стартирате.

Вашият редактор

След като идентичността ви е зададена, можете да конфигурирате текстов редактор по подразбиране, който Git ще стартира за вас винаги, когато трябва да напишете съобщение. Ако не направите това, ще се използва редактора по подразбиране за операционната система.

Ако искате да ползвате различен редактор, например Emacs, направете следното:

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

Под Windows, ако искате да използвате друг текстов редактор, трябва да посочите пълния път до изпълнимия файл. Това може да варира в зависимост от това как е инсталиран редакторът ви.

В случая с Notepad++, популярен редактор сред програмистите, може да искате да използвате 32-битовата версия, защото по времето на писането на книгата, 64-битовата не поддържаше всички плъгини. Ако сте на 32-битов Windows или имате 64-битов редактор на 64-битова система, ще трябва да напишете нещо такова:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe'
-multiInst -notabbar -nosession -noPlugin"
```

Vim, Emacs и Notepad++ са популярни текстови редактори, които програмистите често използват под Unix базирани системи като Linux/MacOS или Windows. Ако не ползвате никой от тях или ползвате 32-битова версия, може да намерите специфични инструкции за настройка на любимия ви редактор за работа с Git в [_core_editor].

Ако не сте настроили редактора правилно, може да изпаднете в конфузни ситуации, когато Git се опита да го стартира. Примерно под Windows, може да се стигне до преждевременно прекратена Git операция по време на редакция инициирана от Git.

Име на клон по подразбиране

По подразбиране, Git ще създава клон с има *master*, когато създавате хранилища с `git init`. От версия 2.28 нагоре, Git позволява задаването на различно име за първия клон.

За да зададете име по подразбиране *main*:

```
$ git config --global init.defaultBranch main
```

Проверка на настройките

Ако искате да проверите какви са текущите настройки, използвайте командата `git config --list`:

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

Можете да видите ключове повече от веднъж, защото Git чете един и същи ключ, но от различни файлове (`[path]/etc/gitconfig` и `~/.gitconfig` например). В такъв случай Git използва последната изписана настройка.

Можете също да проверите какво знае Git за специфична настройка изпълнявайки `git config <key>`:

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

Понеже Git може да чете една и съща конфигурационна стойност от повече от един файл, възможно е да получите неочаквана такава и да не знаете защо. В случаи като този, може да попитате Git за *източника* на конкретната стойност и ще получите в отговор кой конфигурационен файл е имал последната дума:

```
$ git config --show-origin rerere.autoUpdate
file:/home/johndoe/.gitconfig	false
```

Помощна информация в Git

Ако се нуждаете от помощ за Git, има три еквивалентни начина да получите страницата с помощна информация (manpage) за всяка Git команда:

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

Например, за повече информация за командата config, изпълнете:

```
$ git help config
```

Тези команди са полезни, защото имате достъп до тях винаги, дори когато сте офлайн. Ако помощните страници и тази книга не са достатъчни в специфични ситуации, можете да опитате `#git` или `#github` каналите на Freenode IRC сървъра на адрес https://freenode.net. Тези канали редовно са пълни със стотици потребители напреднали с Git и нямащи нищо против да помагат.

В допълнение, ако не се нуждаете от пълната manpage документация, а само от кратко пояснение за опциите на конкретна команда, можете да използвате параметрите `-h` or `--help` така:

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run         dry run
    -v, --verbose         be verbose

    -i, --interactive     interactive picking
    -p, --patch           select hunks interactively
    -e, --edit            edit current diff and apply
    -f, --force           allow adding otherwise ignored files
    -u, --update          update tracked files
    --renormalize         renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add   record only the fact that the path will be added later
    -A, --all             add changes from all tracked and untracked files
    --ignore-removal      ignore paths removed in the working tree (same as --no-all)
    --refresh             don't add, only refresh the index
    --ignore-errors       just skip files which cannot be added because of errors
    --ignore-missing      check if - even missing - files are ignored in dry run
    --chmod (+|-)x        override the executable bit of the listed files
```

Обобщение

Би трябвало вече да имате основните знания за това какво е Git и как се отличава от централизираните системи за контрол на версиите, които може би сте ползвали преди. Би трябвало също така да имате работеща версия на Git в компютъра си, настроена с данните за вашата идентичност. Сега е време да научим няколко основни неща за Git.

Основи на Git

Ако ви трябва само един раздел за да стартирате с Git - това е той. Ще разгледаме всяка една базова команда, от която бихте се нуждали в повечето време, в което ще използвате Git. В края, би трябвало да можете да конфигурирате и инициализирате хранилище, да стартирате/спирате следеното на файлове, да индексирате (stage) и публикувате (commit) своите промени по кода. Ще ви покажем също как да настроите Git така, че да игнорирате специфични файлове или типове файлове, как да отменяте погрешно направени промени лесно и бързо, как да разглеждате историята на вашия проект и промените между различните публикувания, и как да изпращате и издърпвате към/от отдалечени хранилища.

Създаване на Git хранилище

Можете да се сдобиете с Git хранилище (repository) по два основни начина:

1. Първият от тях взема съществуващ проект/директория, който в момента не под контрол на версиите и го импортира в Git
2. Вторият *клонира* съществуващо Git хранилище от друг сървър.

И в двата случая получавате Git хранилище на локалната си машина, готово за работа.

Инициализиране на хранилище в налична директория

Ако започвате да следите съществуващ проект, трябва да отидете в директорията му. Ако никога не сте го правили, това изглежда различно според операционната система:

за Linux:

```
$ cd /home/user/my_project
```

за macOS:

```
$ cd /Users/user/my_project
```

за Windows:

```
$ cd C:/Users/user/my_project
```

и напишете:

```
$ git init
```

Това създава нова под-директория с име `.git`, която съдържа всичко необходимо, от което се

нуждае Git — нещо като скелет на хранилището. В този момент - нищо от вашите файлове не се следи все още. Вижте Git на ниско ниво> за повече информация какви точно файлове се съдържат в тази новосъздадена директория.)

Ако желаете да започнете контрол на версиите на съществуващи файлове (вместо на празна директория), вероятно ще искате да започнете да следите файловете и да направите първоначален къмит. Можете да направите това с малко `git add` команди, които указват файловете, които искате да следите, последвани от `git commit`:

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

Ще се върнем на тези команди след малко. В този момент, вие разполагате с готово Git хранилище със следящи се файлове и сте ги къмитнали за пръв път.

Клониране на съществуващо хранилище

Ако искате да изтеглите копие от съществуващо Git хранилище — например проект, в който желаете да сътрудничите, то командата която ви трябва е `git clone`. Ако сте запознати с други VCS като например Subversion, веднага ще забележите разликата - командата е "clone", а не "checkout". Това е важна разлика - вместо да издърпва просто работещо копие, Git получава пълно копие на почти всички данни, които сървърът пази. Всяка версия на всеки файл от историята на проекта се издърпва по подразбиране когато изпълните `git clone`. Практически, ако сървърът се повреди, можете често да използвате почти всеки от клоновете на всеки клиент за да го възстановите в състоянието му към момента, в който хранилището е било клонирано (може да загубите някои server-side специфични елементи, но цялата ви следяща се и значима информация ще е налична — вижте Достъп до Git на сървъра за повече подробности)

Хранилище се клонира с `git clone <url>`. Например, ако искате да клонирате linkable библиотеката `libgit2`, можете да го направите така:

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

Командата ще създаде директория с име `libgit2`, ще инициализира `.git` под-директория, ще изтегли на компютъра ви всички данни от това хранилище и ще ви даде всички файлове от последната работеща версия. Ако влезете в новата папка `libgit2`, ще видите файловете вътре - готови за работа по тях.

Ако желаете клонирането да е в директория с различно от `libgit2` име - можете да го подадете като следващ параметър към командата:

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

Тази команда прави същото като предишната, но в резултат ще ви даде директория с име

`mylibgit`.

Git може да работи с различни протоколи за трансфер. Предишният пример използва `https://`, но може да видите също и `git://` или `user@server:path/to/repo.git`, което използва SSH като транспортен протокол. Достъп до Git на сървъра ще разкаже повече за всички налични опции, които един сървър може да ползва за да предостави достъп до вашите Git хранилища, в едно с предимствата и недостатъците им.

Запис на промени в хранилището

Вече имате *чисто ново* Git хранилище и *работещо копие* на файловете от проекта. Сега трябва да започнете да правите промените, които желаете и да записвате snapshot-и на промените в хранилището всеки път, когато проектът ви достигне състояние, което бихте желали да запишете на сигурно място.

Помнете, че всеки файл от работната ви директория може да бъде в два статуса - *проследяван* или *не* (tracked/untracked). Tracked файловете са тези от последния snapshot, те може да са непроменени, променени и индексирани (staged). Накратко, *tracked* файловете са тези, които Git познава.

Untracked файловете са всичко останало - всички файлове в работната ви директория, които не са били в последния ви snapshot и не са в staging областта. Когато за пръв път клонирате хранилище, всички ваши файлове ще бъдат tracked и същевременно - unmodified, защото Git току що ги е извлякъл и вие все още не сте променяли нищо по тях.

Когато започнете да променяте файловете, Git ги вижда вече като modified, понеже сте ги редактирали след последния къмит. Вие индексирате тези променени файлове, след това къмитвате промените им и този цикъл се повтаря в течение на работата ви.

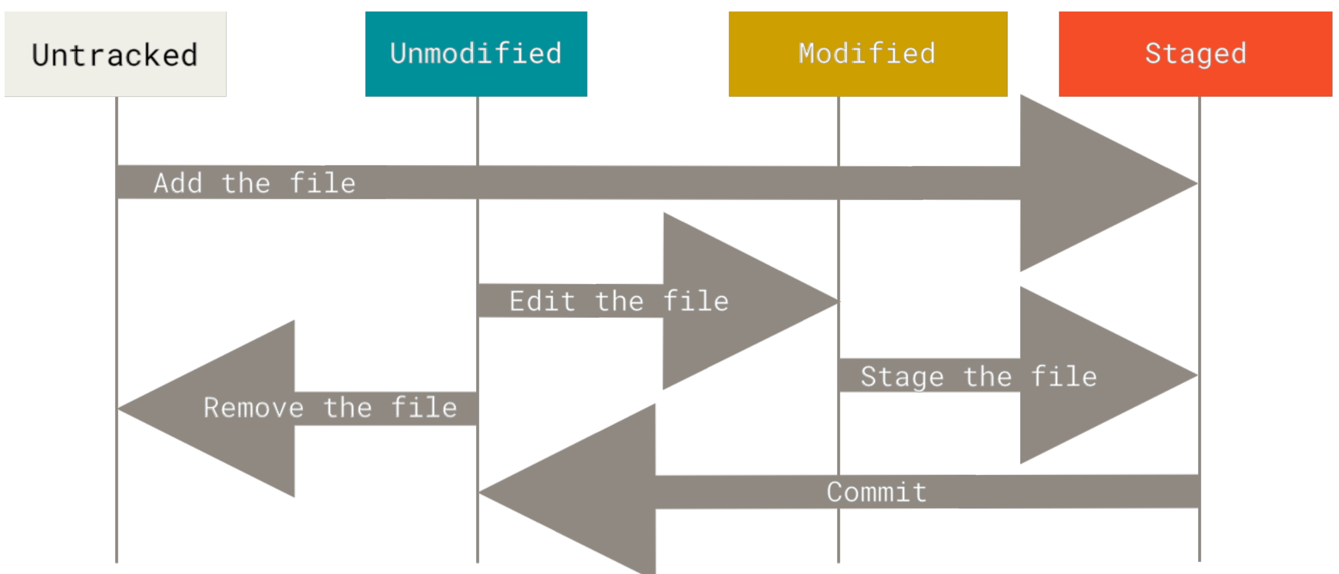


Figure 8. Промяната в статуса на вашите файлове

Проверка на статуса на файловете

Основният инструмент, с който се проверява състоянието на файловете ви е командата `git status`. Ако я изпълните директно след клониране, когато не сте правили промени все още,

ще видите следното:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Това означава, че имате чиста работна директория - с други думи, никой от следените ви файлове не е променян. Git също така не вижда никакви непроследени файлове, иначе щеше да ги покаже тук. Накрая, командата ви казва в кой клон (branch) на проекта се намирате и че не се отклонявате от същия клон на сървъра. Засега, този клон е винаги `master`, както е по подразбиране, към момента това не ви интересува. Клонове в Git ще разгледа клоновете и референциите в подробности.

Нека сега добавим нов файл в проекта, прост `README` файл. Ако файлът не е съществувал преди и изпълните `git status`, ще видите untracked файла си така:

```
$ echo 'My Project' > README
$ 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)

    README

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

Можете да видите, че новият `README` файл е непроследен, защото е в секцията "Untracked files" на изхода от командата. Untracked в общи линии означава, че Git вижда файл, който не е присъствал в предишния snapshot (commit) и Git няма сам да започне да го прибавя към следващите commits докато вие не укажете това изрично. Това е умишлено поведение и ви предпазва от ситуации, в които бихте могли автоматично да добавяте файлове, които не желаете, например генерирани binary файлове. Вие обаче искате да включите `README` файла, така че нека го направим.

Проследяване на нови файлове

За да започнете да следите нов файл, използвайте командата `git add`. За вашия `README` файл, изпълнете това:

```
$ git add README
```

Ако след това изпълните отново статус командата, ще видите че `README` файлът вече се следи и е индексиран за включване в следващия къмит:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

Разбирате, че файлът е индексиран, защото се намира в секцията със заглавие “Changes to be committed”. Ако къмитнете в този момент, файлът който ще попадне в следващия snapshot-а ще е в състоянието, в което е бил, когато сте изпълнили `git add` командата за него. Може да си спомните, че когато по-рано изпълнихте `git init`, след това изпълнихте и `git add <files>` — това беше за да започнете да следите файлове във вашата директория. Командата `git add` приема име на път за файл или директория, ако е директория - тя добавя всички файлове в нея рекурсивно.

Индексиране на променени файлове

Нека променим файл, който вече се проследява. Ако промените вече проследен файл с име `CONTRIBUTING.md` и след това изпълните `git status` отново, ще видите нещо подобно:

```
$ 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:   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
```

Файлът `CONTRIBUTING.md` се появява под секцията “Changes not staged for commit” — което значи, че проследеният файл е бил променен в работната директория, но все още не е индексиран за къмитване. За да го индексирате - изпълнете командата `git add`. Както вече виждате, `git add` е многоцелева команда — използвате я както за да започнете да следите файлове, така и за да ги индексирате в staging областта и дори да правите по-различни неща, като например да маркирате отбелязани като конфликтни по време на сливане файлове като коректни такива. Би могло да ви е от полза да приемате значението ѝ повече като “добави това съдържание в следващия къмит” вместо като “добави този файл към проекта”. Нека сега изпълним `git add` за да индексираме файла `CONTRIBUTING.md`, след което да пуснем `git status` отново:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

И двата файла сега са индексирани и ще попаднат в следващия къмит. В този момент, представете си, че сте забравили да направите една дребна промяна по `CONTRIBUTING.md` преди да го публикувате. Вие го отваряте отново, правите промяната и сте готови на къмитнете. Обаче, нека пуснем `git status` още един път:

```
$ vim CONTRIBUTING.md
$ 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:   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
```

И какво виждаме? Сега `CONTRIBUTING.md` се показва *едновременно* като staged и unstaged. Как е възможно това? Оказва се, че Git индексира файла точно както е бил, когато сте изпълнили `git add`. Ако къмитнете сега, версията на `CONTRIBUTING.md`, която ще отиде в snapshot-а ще е тази, след която сте изпълнили `git add` - а не тази в която е, когато изпълните `git commit`. С други думи - вашата малка промяна няма да бъде включена и публикувана. Ако промените файл след като сте пуснали командата `git add`, трябва да изпълните `git add` отново, ако желаете да индексирате новата промяна:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

Кратък статус

`git status` информацията е доста изчерпателна, но и многословна. Git поддържа и флаг за кратък статус, така че да виждате промените си в по-компактна форма. Ако изпълните `git status -s` или `git status --short`, получавате по-опростен изход:

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

Новите, непроследени файлове са със знака `??`, новите индексирани файлове с `A`, променените файлове с `M` и т.н. Изходът е в две колони — лявата показва статуса на staging областта, а дясната статуса на работната директория. Така в горния пример, `README` файлът е променен в работната област, но не е индексиран, докато файлът `lib/simplegit.rb` е променен и индексиран. Файлът `Rakefile` е променен, индексиран и след това променен отново, така че по него има промени които са индексирани и такива, които не са.

Игнориране на файлове

Често, ще имате класове от файлове, за които няма да искате Git да ги добавя автоматично и дори да ви ги показва като непроследени. Това обикновено са автоматично генерирани файлове - лог-файлове или такива създадени от компилиращата ви система. В подобни случаи, можете да създадете файл с име `.gitignore`, в който да ги опишете с подходяща маска за имената им. Ето един примерен такъв файл:

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

Първият ред в него указва на Git да пропуска всички файлове завършващи на “.o” или “.a” — обектни и архивни файлове, които може да са създадени от компилатора. Вторият ред указва да се пропускат всички файлове, чиито имена завършват с тилда (`~`), които се ползват в много текстови редактори като Emacs за маркиране на временни файлове.

Можете също да включвате log, tmp, или pid директории, автоматично генерирана документация и т.н. Добра идея е да си направите `.gitignore` файла преди да започнете работа, така че да не къмитнете без да искате нежелани файлове.

Правилата за маските, които можете да включвате в `.gitignore` файла са както следва:

- Празните редове и редовете започващи с `#` се игнорират.
- Работят стандартните глобални правила за маски и те ще бъдат приложени рекурсивно по цялото работно дърво.
- Можете да започвате маските с (`/`) за да избегнете рекурсия.
- Можете да завършвате маските с (`/`) за да указвате директория.
- Можете да обърнете логиката на маската като я започнете с (`!`).

Глобалните правила са подобни на опростени регулярни изрази, които шеловете използват. Звездичката (`*`) търси за нула или повече символа; `[abc]` търси за кой да е символ в скобите (в този случай a, b, или c); въпросителният знак (`?`) търси единичен символ; символи в скоби с тире между тях (`[0-9]`) търсят за произволен символ в обхвата между символите (в този случай от 0 до 9). Можете да използвате две звездички за да търсите в под-директории; `a/**/z` ще открие `a/z`, `a/b/z`, `a/b/c/z`, и т.н.

Ето друг примерен `.gitignore` файл:

```
# без .a файлове
*.a

# но lib.a се включва, въпреки че игнорирате всички .a файлове отгоре
!lib.a

# игнорирай само TODO файла в текущата директория, не и под-директориите
съдържащи TODO
/TODO

# игнорира всички файлове в коя да е директория с име build
build/

# игнорира doc/notes.txt, но не и doc/server/arch.txt
doc/*.txt

# игнорира всички .pdf файлове в директорията doc/ и всички нейни под-
директории
doc/**/*.pdf
```

GitHub поддържа сравнително подробен списък от добри `.gitignore` примери за стотици проекти и езици на адрес https://github.com/github/gitignore, ако искате отправна точка за проекта си.

В общия случай, едно хранилище би могло да има единичен `.gitignore` файл в най-горната директория, който се прилага върху всички други рекурсивно. Обаче, възможно е да имате и допълнителни `.gitignore` файлове в поддиректориите. Правилата в тези вложени `.gitignore` файлове ще се прилагат само към файловете, намиращи се в директорията, в която се пазят. Linux kernel хранилището например има 206 `.gitignore` файла.

Извън темата на тази книга е да се впускаме в детайли за множеството `.gitignore` файлове; погледнете `man gitignore`, ако желаете повече информация.

Преглед на индексираните и неиндексирани промени

Ако командата `git status` е твърде неясна за вас (понеже може да искате да знаете точно какво сте променили, а не само имената на файловете), можете да ползвате командата `git diff`. Ще разгледаме по-подробно `git diff` по-късно, вие вероятно най-често ще я ползвате за отговор на два въпроса: Какво сте променили, но не сте индексирали все още? Какво сте индексирали и предстои да къмитнете? Въпреки, че `git status` в общи линии отговаря показвайки ви имената на файловете, `git diff` показва точните редове код добавени и премахнати — пачът какъвто точно е бил.

Да кажем, че редактирате и индексирате `README` файла отново и след това редактирате `CONTRIBUTING.md` без да го индексирате. Ако пуснете `git status` командата, вие виждате нещо такова:

```
$ 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)

    modified:   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
```

За да видите какво сте променили, но не индексирали - напишете `git diff` без аргументи:

```
$ 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 your 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
```

Командата сравнява наличното в работната директория с това в индексната област. Резултатът ви показва промените, които са направени, но не са индексирани.

Ако желаете да видите какво сте индексирали и ще отиде в следващия къмит, можете да използвате `git diff --staged`. Това сравнява индексираните промени с това, което е било в последния къмит:

```
$ 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
```

Важно е да запомните, че `git diff` сама по себе си не показва всички промени от последния къмит — а само тези, които все още не са индексирани. Това може да е смущаващо, защото значи, че ако сте индексирали всичките си промени, `git diff` няма да покаже нищо.

Друг пример, ако индексирате файла `CONTRIBUTING.md` и след това го промените, можете да ползвате `git diff` за да видите промените във файла, които са индексирани и тези които не са. Ако състоянието ни изглежда така:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ 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)

    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
```

Сега можете да ползвате `git diff` за да видите какво все още не е индексирано:

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

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

и `git diff --cached` за да видите файла в индексираното му състояние (`--staged` и `--cached` са синоними):

```
$ 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 your 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 във външен инструмент

Изпълнете `git difftool --tool-help` за да видите какви diff инструменти са налични за вашата система. Ние ще продължим да ползваме `git diff` командата по различни начини в книгата. Има и друг начин за преглед на промените, ако предпочитате графичен или друг способ. Ако изпълните `git difftool` вместо `git diff`, можете да гледате всяко от сравненията в софтуери като emerge, vimdiff и много други подобни, вкл. комерсиални такива.

Публикуване на промените (commit)

Сега, след като индексната област е в състоянието, което искате, можете да публикувате (къмитнете) вашите промени. Помнете, че всичко, което все още не е индексирано — всякакви файлове, които сте създали или редактирали след последната `git add` команда — няма да отидат в това публикуване. Те ще останат като променени файлове на вашия диск. Нека кажем, че последния път когато сте пуснали `git status`, вие сте видели, че всичко е индексирано и сте готови да къмитнете промените. Най-простият начин да запишете е изпълнявайки командата `git commit`:

```
$ git commit
```

Правейки това, Git ще стартира вашия текстов редактор.

(Това се определя от `EDITOR` environment променливата на вашия шел — обикновено vim или emacs, въпреки че можете да конфигурирате редактора по подразбиране с `git config --global core.editor` командата както видяхме в Начало.

Редакторът показва следното (в случая екранът е от 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
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#	new file:   README
#	modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Можете да видите, че подразбиращото се къмит съобщение съдържа последния изход от командата `git status` в коментиран вид и един празен ред над него. Можете да изтриете тези коментари и да напишете собствено съобщение или да ги оставите там за да ви припомнят по-късно какво точно сте публикували.

Ако искате още по-подробно напомняне за това какво сте модифицирали, можете да изпълните командата с параметър `git commit -v`. Това ще включи в съобщението и diff на самите промени, така че да можете точно да проследите какво сте къмитнали.

Когато излезете от редактора запазвайки промените, Git ще публикува промените заедно със съобщението (коментарите и diff информацията се премахват)

Вместо да пускате текстовия редактор, можете да подадете къмит съобщението директно като параметър на командата с флага `-m`:

```
$ 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
```

Сега вече къмитнахте за пръв път промените си! Можете да видите, че това действие ви дава и допълнителна информация за себе си: към кой клон сте къмитнали (`master`), каква е SHA-1 чексумата на къмита (`463dc4f`), колко на брой файлове са променени и статистика за добавените и премахнати редове код.

Помнете, че къмитът съдържа моментна снимка на това, което е имало в индексната област (staging area). Всичко, което не е било там няма да присъства в къмита и файловете ще си стоят като променени. За да ги добавите - трябва да направите следващ къмит. Всеки път, когато къмитвате промени, вие правите snapshot на състоянието на вашия проект и по-късно можете да го възстановите или да го ползвате за сравнение.

Прескачане на Staging областта

Въпреки, че може да е много полезна за фина настройка на вашите промени, понякога индексната област може да се прескочи в процеса на работа. Ако искате директно да къмитнете променени файлове без да ги добавяте в нея, Git осигурява средство за това. Опцията `-a` към командата `git commit` прави така, че Git автоматично ще индексира всеки следящ се файл преди да направи къмита и така можете да пропуснете понякога досадната необходимост да изпълнявате `git add`:

```
$ 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:   CONTRIBUTING.md

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

Забележете, че сега не беше нужно да изпълнявате `git add` за файла `CONTRIBUTING.md` преди да го къмитнете. Това е, защото `-a` параметърът включва всички променени файлове. Това е удобно, наистина, но бъдете внимателни, понякога този флаг може да включи в къмита нежелани промени.

Изваждане на файлове

За да извадите файл от Git, вие трябва да го изключите от списъка със следящи се файлове (по-прецизно казано, да го премахнете от индексната област) и след това да публикувате промяната. Командата `git rm` прави това и също така изтрива файла от работната директория, така че да не го виждате като непроследен файл следващия път.

Ако просто изтриете файла от работната си директория, той се показва под “Changes not staged for commit” (тоест, *unstaged*) секцията от изхода на `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")
```

След това, ако изпълните `git rm`, системата индексира това изтриване на файла:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ 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)

    deleted:    PROJECTS.md
```

При следващия къмит файлът ще изчезне и повече няма да се следи. Ако сте променили файла и вече сте го добавили към индекса, трябва да форсирате изтриването с параметъра `f`. Това е предпазна опция за да предотвратите случайно изтриване на данни, които не са били публикувани и не могат да се възстановят от Git.

Друго полезно действие, което вероятно ще искате да можете да правите, е да пазите файла в работната директория, но да го извадите от индекса. С други думи, да пазите файла на диска си, но Git да не го следи повече. Това е особено полезно, когато сте забравили да добавите нещо в `.gitignore` файла си и без да искате сте го индексирали - например голям лог-файл или купчина от `.a` файлове създадени от компилатора. За да се справите с това, ползвайте опцията `--cached`:

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

Командата `git rm` може да се ползва с имена на файлове, директории и цели маски за имена. Това означава, че можете да правите подобни неща:

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

Отбележете обратния слеш (`\`) преди звездичката `*`. Това е нужно, защото Git прави своя собствена развивка на имената на файлове в допълнение към развивката, която прави

шела. Тази команда премахва всички файлове с разширение `.log` намиращи се в директорията `log/`. Можете да направите и следното:

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

Тази команда премахва всички файлове, имената на които завършват със символа `~`.

Преименуване на файлове

За разлика от много други VCS системи, Git не следи експлицитно преименуването на файлове. Ако преименувате файл в Git, никакви мета данни няма да се съхранят в Git базата, така че да му указва, че сте преименували файла. Обаче, Git е достатъчно интелигентен за да усети това — ще се занимаем с детекцията на преименуваните файлове малко по-късно.

Затова е малко смущаващо, че Git всъщност има `mv` команда. Ако искате да преименувате файл в Git, можете да изпълните това:

```
$ git mv file_from file_to
```

и то си работи. На практика, ако изпълните командата и погледнете в статуса, ще видите че Git гледа на файла като на преименуван:

```
$ git mv README.md README
$ 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)

    renamed:    README.md -> README
```

Обаче, това е еквивалентно на следното:

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

Git установява, че това е безусловно преименуване, така че няма значение дали сте променили файла по този начин или с `mv` командата. Единствената реална разлика е, че `git mv` е една команда вместо три — така че това е команда за удобство. По-важното е, че можете да използвате произволни средства за преименуване на файлове и да се занимавате с add/rm действията по-късно, преди да къмитнете промените.

Преглед на историята на действията

След като сте създали няколко къмита, или ако сте клонирали хранилище с налични такива - може да пожелаете да погледнете назад за да видите как се е развивал проекта. Най-простият, но и мощен инструмент за това е командата `git log`.

Тези примери използват много прост проект наречен “simplegit”. За да го изтеглите, изпълнете:

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

След като пуснете `git log` в този проект, трябва да видите изход подобен на следния:

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

    Change version number

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

    Remove unnecessary test

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

    Initial commit
```

По подразбиране, без аргументи, `git log` показва промените направени в хранилището в обратен хронологичен ред, най-новите се показват най-горе. Както можете да видите, командата показва всеки къмит с неговата SHA-1 чексума, името и имейла на автора, датата и съобщението на къмита.

Самата команда `git log` разполага с голям брой различни опции, които да ви помогнат да намерите точно необходимата информация. Ще покажем някои от най-популярните.

Един от най-полезните аргументи е `-p` или `--patch`, който показва разликите (*patch* изхода) настъпили с всеки къмит. Можете да използвате също и `-2`, което ще ограничи изхода само до последните два къмита:

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

    Change 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

    Remove 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
```

Тази команда показва същата информация, но с разликите директно след всеки елемент от историята. Това е много полезно за преглед на код или за набързо разглеждане на промените настъпили в серия къмити от даден сътрудник. Можете да използвате и серия от статистически параметри с `git log`. Например, ако искате да видите съкратена статистика за всеки къмит, ползвайте параметъра `--stat`:

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

    Change 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

    Remove 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

    Initial commit

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

Както се вижда, параметърът `--stat` отпечатва след всеки къмит списък на модифицираните файлове, колко от тях са променени и колко редове в тях са добавени и изтрити. Отпечатва се и сумарна информация в края.

Друга наистина полезна опция е `--pretty`. Това форматира изхода по начин различен от подразбиращия се. Разполагате с няколко избора за ползване. Стойността `oneline` печата всеки къмит на единичен ред, което е полезно ако търсите в множество къмити. В допълнение, стойностите на аргумента `short`, `full`, и `fuller` показват изхода в почти същия формат, но с по-малко или повече информация съответно:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 Change version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit
```

Най-интересната стойност на параметъра е `format`, която ви позволява сами да укажете формата на изхода. Това е особено полезно, ако се налага да генерирате изход за последваща машинна обработка, защото можете изрично да укажете формата и да сте сигурни, че той

ще си остане същия докато версиите на Git се обновяват:

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

Полезни опции за `git log --pretty=format` показва някои от най-популярните флагове, които `format` разбира.

Table 1. Полезни опции за `git log --pretty=format`

Флаг	Описание
`%H`	Хеш на къмита
`%h`	Съкратен хеш на къмита
`%T`	Tree хеш
`%t`	Съкратен Tree хеш
`%P`	Родителски хешове
`%p`	Съкратени родителски хешове
`%an`	Име на автора
`%ae`	Имейл на автора
`%ad`	Дата на author къмит (форматът взема предвид --date=option)
`%ar`	Релативна дата на author къмит
`%cn`	Име на committer
`%ce`	Имейл на committer
`%cd`	Дата на committer-къмит
`%cr`	Релативна дата на committer-къмит
`%s`	Съобщение

Може да се запитате каква е разликата между *author* и *committer*. Авторът е лицето, което първоначално е писало нещо по дадена задача, докато committer е лицето, което последно е допринесло в нея. Така, ако вие изпратите даден пач към проект и някой от сътрудниците го приложи към проекта, и двамата правите принос — вие като автор и въпросния сътрудник като committer. Ще разгледаме по-подробно това разделение в Git в разпределена среда.

Аргументите `oneline` и `format` са особено полезни в съчетание с друг аргумент, `--graph`. Това добавя забавна малка ASCII графика, показваща клона и историята на сливанията:

```
$ 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 Add 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
```

Този тип изход става по-интересен, когато навлезем в материята на клоновете (branching) и сливането (merging) в следващата глава.

Но това са само някои прости опции за форматиране на изхода на `git log` — съществуват и много други. Common options to `git log` изброява опциите, които вече разгледахме плюс някои други полезни такива, променящи изхода на командата log.

Table 2. Common options to `git log`

Опция	Описание
`-p`	Показва пача за всеки къмит.
`--stat`	Показва статистики за файловете променени във всеки къмит.
`--shortstat`	Показва само changed/insertions/deletions реда от --stat варианта.
`--name-only`	Показва списък на променените файлове след информацията за къмита.
`--name-status`	Показва списък на засегнатите файлове ведно с added/modified/deleted детайлите.
`--abbrev-commit`	Показва само първите няколко символа на SHA-1 чексумата, вместо всичките 40.
`--relative-date`	Показва датата в релативен формат (например, “2 weeks ago”) вместо в пълния ѝ формат.
`--graph`	Показва ASCII графика на branch и merge историята до изхода.
`--pretty`	Показва къмитите в алтернативен формат. Стойностите включват oneline, short, full, fuller, и format (където указвате собствен формат на изхода).
`--oneline`	Съкращение за `--pretty=oneline --abbrev-commit` използвани заедно.

Ограничаване на изхода

В допълнение към опциите за формат на изхода, `git log` поддържа и полезни средства за лимитиране, с които да показвате само част от къмитите. Вече видяхте едно от тях под формата на параметър `-2`, показващ само последните два къмита. В действителност, можете да ползвате `-<n>`, където `n` е произволно число, за да покажете колкото от тях желаете. Практически обаче, това рядко се налага, защото Git по подразбиране странира изхода си, така че да виждате само по един екран в даден момент.

Обаче, опциите за ограничаване по време, като `--since` и `--until`, са много полезни. Например, тази команда показва списък от къмитите направени в последните две седмици:

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

Тази команда работи с множество формати — можете да укажете специфична дата като `"2008-01-15"`, или релативен период като `"2 years 1 day 3 minutes ago"`.

Можете също да филтрирате списъка с къмити, които съответстват на определен критерий за търсене. Опцията `--author` позволява да търсите по определен автор и `--grep` опцията ви позволява да търсите по ключови думи в съобщенията на къмитите. (Отбележете, че ако искате да използвате едновременно и двете опции, трябва да добавите `--all-match`, иначе командата ще показва резултат ако дори и само единият критерий съвпада)

Можете да укажете повече от една инстанция за `--author` и `--grep` критериите, което ще ограничи показваните къмити до *всеки*, който отговаря на `--author` и *всеки*, който отговаря на `--grep` маските; обаче, добавянето на `--all-match` опцията допълнително ограничава изхода до само тези, които отговарят на *всички* `--grep` маски.

Друг полезен филтър е опцията `-S` (позната още като “pickaxe” опцията на Git), която приема стринг и показва само тези къмити, които са променили броя на срещанията на този стринг. Ако примерно желаете да намерите последния къмит, който е добавил или премахнал обръщение към специфична функция, можете да изпълните:

```
$ git log -S function_name
```

Последната доста полезна опция, която можете да пратите като филтър към `git log` е път към файл/директория. Ако укажете име на директория или файл, можете да ограничите изхода до къмитите, в които са правени промени по тези специфични файлове. Това винаги трябва да е последен параметър към командата и обикновено се слага префикс от две тирета (`--`) за разделяне на пътищата от другите параметри:

```
$ git log -- path/to/file
```

В таблицата Опции за ограничаване на изхода на `git log` разглеждаме тези и някои други опции.

Table 3. Опции за ограничаване на изхода на `git log`

Опция	**Описание**
`-<n>`	Показва само последните n на брой къмита
`--since`, `--after`	Показва само къмитите направени след указаната дата.
`--until`, `--before`	Показва само къмитите направени преди указаната дата.

Опция	Описание
`--author`	Показва само къмитите, в които авторът съответства на подадения стринг.
`--committer`	Показва само къмитите, в които committer-ът съответства на подадения стринг.
`--grep`	Позволява търсене по стринг в commit-съобщението
`-S`	Позволява търсене по стринг в промените в кода

Например, ако искате да видите кои къмити променили тестови файлове в сорс кода на Git са направени от Junio Hamano през октомври 2008 и това не са merge-къмити, можете да изпълните нещо подобно:

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

Тази команда филтрира близо 40 хиляди къмита от историята на сорс кода на Git и показва само 6-те, които отговарят на критерия.

Пропускане на показването на merge къмити

В зависимост от работната последователност, която се използва във вашето хранилище, възможно е голям процент от къмитите в историята да са просто merge къмити, които обикновено не са много информативни. За да забраните показването им, и да опростите историята, просто подайте аргумента `--no-merges`.

Възстановяване на направени действия

Във всеки един момент може да се окаже, че искате да отмените дадена промяна по проекта. Тук ще разгледаме някои основни средства за отмяна на промени. Бъдете внимателни, защото не винаги може да отмените отмяна! Това е една от малките области в Git, където можете да загубите част от данните си, ако не действате прецизно.

Едно от най-честите действия по отмяна е да направите къмит твърде рано и да сте забравили да добавите няколко файла или да сте объркали commit съобщението. Ако искате да опитате този къмит отново, направете забравените промени, индексирайте ги и можете да къмитнете отново с параметъра `--amend`:

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

Тази команда взема съдържанието на индекса и го използва за къмита. Ако не сте правили промени от последния къмит (например, пускате командата веднага след предишния къмит), тогава вашият snapshot ще изглежда по идентичен начин и единственото нещо, което ще промените е commit съобщението.

Ще се отвори същия редактор, но със заредено съобщението от последния къмит. Можете да промените съобщението както обикновено, но то ще се презапише върху предишния ви къмит.

Като пример, ако къмитнете и веднага след това се сетите, че не сте индексирали промените във файл, който искате да влиза в къмита, можете да направите следното:

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

Ще си останете с един къмит — вторият замества резултатите от първия.

Важно е да се запомни, че когато коригирате последния къмит, го *заменяте* с изцяло нов, подобрен такъв, който изцяло замества стария. По същество историята ще изглежда така, сякаш предишния къмит никога не се е случвал и няма да се показва в нея.

Очевидната полза от amending къмитите е, че можете да правите малки промени по последния къмит, без да трябва да задръствате историята със съобщения от сорта на “упс, забравих да добавя файл” или “поправям грешка в последния къмит”.

Отменяйте къмити само ако те са все още локални и не са изпращани никъде отдалечено. Отмяна на предишно изпратени към сървъра къмити и форсираното изпращане на клона ще предизвика проблеми за сътрудниците ви. За повече информация какво се случва, когато се прави това и как да възстановите пораженията, ако сте от приемащата страна, прочетете Опасности при пребазиране.

Изваждане на файл от индекса

Следващите две секции демонстрират как се работи с индексната област и промените в работната директория. Хубавото е, че командата, която използвате за да определите статуса на тези две области, също така ви подсказва и как да отменяте направени в тях промени. За пример, нека кажем, че сте променили два файла и искате да ги къмитнете като две отделни промени, но неволно сте изпълнили `git add *` и сте ги индексирали и двата. Как да извадите от индекса единия от двата? Командата `git status` ви подсказва начина:

```
$ 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
```

Точно под текста “Changes to be committed”, пише `git reset HEAD <file>...` to unstage. Да ползваме този съвет за да де-индексираме файла `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
```

Командата е леко странна, но работи. Файлът `CONTRIBUTING.md` сега си е променен, но вече не е в индекса

Вярно е, че `git reset` може да се окаже опасна команда, особено ако ѝ подадете флага `--hard`. Обаче, в горния случай, файлът в работната ви директория е недокоснат, така че тя е сравнително безопасна.

Към момента, това мистериозно извикване е всичко, което трябва да знаете за `git reset` командата. Ще навлезем в много по-дълбоки подробности за това какво прави `reset` и как да я ползваме за да правим наистина интересни неща в Мистерията на командата Reset.

Отмяна на промените в променен файл

Какво се случва, ако установите, че не искате да пазите промените си във файла `CONTRIBUTING.md`? Как можем лесно да го "де-модифицираме" — тоест да го превъртим назад до съдържанието, което е имал при последния къмит (или както е бил при първоначалното клониране в работната ви директория)? За късмет, `git status` ни подсказва и това. В последния ни пример, работната област изглеждаше така:

```
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
```

Това ви казва доста недвусмислено как да отмените промените, които сте направили. Нека да го изпълним:

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

    renamed:    README.md -> README
```

Можете да се уверите, че промените изчезнаха.

Важно е да се помни, че `git checkout -- <file>` е опасна команда. Всички локални промени, които сте правили по този файл са изчезнали необратимо — Git просто заменя файла с най-скоро къмитнатата му версия. Никога не ползвайте тази команда, освен ако не сте абсолютно сигурни, че не желаете промените във файла.

Ако желаете да запазите промените си по файла, но все още държите да пазите този файл настрани от проекта към дадения момент, има по-добри начини да го направите, ще ги разгледаме в материала за скриване (stashing) и клонове код (branchingКлонове в Git).

Помнете, всичко което е *къмитнато* в Git може почти винаги да бъде възстановено по-късно. Дори къмити, които са били в изтрити клонове или комити презаписани с къмит от тип `--amend`, могат да бъдат възстановени (вижте Възстановяване на данни за повече информация). Обаче, всичко което загубите и не е било къмитното - най-вероятно няма да може да се възстанови.

Възстановяване с git restore

Git 2.25.0 има нова команда: `git restore`. Тя по същество е алтернатива на `git reset`, която разгледахме. От версия 2.25.0 натама, Git ще използва `git restore` вместо `git reset` за много undo операции.

Нека повторим стъпките и да възстановяваме данните с `git restore` вместо с `git reset`.

Изваждане от индекса на файл с git restore

Следващите 2 секции демонстрират как се работи с индексната област и работната директория посредством `git restore`. Добрата страна е, че командата, която използвате за да

установите статуса на тези две области също така ви подсеща и как да отменяте промени в тях. Например, да кажем, че сте променили два файла и искате да ги къмитнете като две отделни промени, но без да искате сте изпълнили `git add *` и сте ги индексирали и двата. Как да извадие от индекса единия? Командата `git status` ви напомня:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   CONTRIBUTING.md
    renamed:    README.md -> README
```

Веднага след реда "Changes to be committed", виждате: `git restore --staged <file>...` to unstage. Нека използваме този съвет, за да извадим от индекса файла `CONTRIBUTING.md`:

```
$ git restore --staged CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    README.md -> README

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

Сега `CONTRIBUTING.md` е модифициран, но вече не е индексиран.

Отмяна на промените по променен файл с git restore

Какво да направим, ако установим, че не искаме промените, които сме направили по `CONTRIBUTING.md`? Как лесно да ги отменим — да го върнем до състоянието му, в което е бил преди последния къмит (или в което е бил при клонирането)? За щастие, `git status` подсказва и това. В последния ни пример, работната директори изглеждаше така:

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

Командата доста недвусмислено показва как да отменим промените. Нека го направим:

```
$ git restore CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    README.md -> README
```

Важно е да помним, че `git restore <file>` е опасна команда. Всички локални промени, които сме направили по даден файл изчезват — Git просто подменя файла с последната му къмитната версия. Не я ползвайте, освен ако не сте абсолютно сигурни, че не искате тези локални несъхранени промени.

Работа с отдалечени хранилища

За да можете да сътрудничите в Git проекти, трябва да знаете как да управлявате отдалечените си хранилища. Отдалечените хранилища са версии на вашия проект, които се съхраняват някъде в корпоративната мрежа или в Интернет. Може да имате много от тях, като всяко от хранилищата може да е с права за вас само за четене или за четене/писане. Сътрудничеството с други разработчици изисква управление на тези отдалечени хранилища и издърпване/изпращане на данни от/към тях когато трябва да споделите работата си. Управлението на отдалечени хранилища включва умения за това как да добавяте такива, да премахвате хранилища, които вече не са валидни, управление на разнообразни отдалечени клонове код и дефинирането им като следени (tracked) или не и т.н. В тази секция разглеждаме някои от тези неща.

Отдалечени хранилища могат да присъстват на локалната ви машина.

Напълно възможно е да работите с "отдалечено" хранилище, което в действителност се намира на същия хост. Думата "отдалечено" не означава непременно, че хранилището е някъде другаде в мрежата или в Интернет, тя означава че то просто е другаде. Работата с подобно хранилище все още си изисква всичките стандартни push/pull/fetch операции.

Показване на отдалечените хранилища

За да видите кои отдалечени сървъри имате конфигурирани, използвайте командата `git remote`. Тя отпечатва съкратените имена на всяко отдалечено хранилище, което сте указали. Ако сте клонирали отдалечено хранилище, трябва да видите в списъка поне елемента `origin` — това е подразбиращото се име, което Git дава на сървъра, от който сте клонирали:

```
$ 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
```

Можете да ползвате и флага `-v`, който показва пълния URL, който Git пази за съответното кратко име на отдалеченото хранилище:

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

Ако имате повече от едно отдалечено хранилище, командата показва всички тях. Например, хранилище с много отдалечени копия за работа с няколко сътрудника, би могло да изглежда така.

```
$ 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)
```

Това означава, че можем да издърпваме работата от всеки от тези потребители доста лесно. Може освен това да имаме права да изпращаме наш код към едно или повече от копията, въпреки че това не става ясно от тук.

Отбележете също, че тези отдалечени копия използват различни протоколи, ще обърнем повече внимание на това в Достъп до Git на сървъра.

Добавяне на отдалечени хранилища

Вече показахме как командата `git clone` самичка създава `origin` отдалеченото хранилище за вас. Ето как да си добавите изрично отдалечено хранилище. Командата `git remote add <shortname> <url>` добавя отдалечено хранилище със съкратено име, което впоследствие да

можете да ползвате лесно в обръщенията към него:

```
$ 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)
```

Сега вече можете да ползвате краткото име pb, вместо целия отдалечен URL. Например, ако искате да издърпате цялата информация, която Paul има, но все още не е в локалното ви хранилище, можете да използвате командата `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
```

Сега `master` клонът на Paul е достъпен локално за вас като `pb/master` — можете да го слеете в някой от вашите клонове код или да превключите към него, ако желаете да го прегледате. Ще разгледаме подробно какво са клоновете код и как да ги използваме в Клонове в Git.

Fetching и Pulling на данни от отдалечени хранилища

Както току що видяхте, извличането на данни от отдалечен проект се прави с:

```
$ git fetch <remote>
```

Командата се свързва с отдалечения сървър и изтегля всички данни за него, които все още нямате локално. След като направите това, ще имате указатели към всички клонове код на това отдалечено хранилище, така че да можете да ги инспектирате и слеете с локалните си клонове по всяко време.

Ако клонирате хранилище, командата `git clone` автоматично го добавя като отдалечено под съкратеното име "origin". Така че, `git fetch origin` изтегля всички нови данни от това хранилище, които са били добавени след като сте го клонирали (или последно актуализирали). Важно е да се отбележи, че `git fetch` само изтегля информацията в локалната ви база данни — тя не прави автоматично сливане с каквато и да било част от локалната ви работа и по никакъв начин не модифицира локалните файлове. Вие сами трябва да направите това ръчно, когато сте готови.

Ако текущият ви клон код (branch) е настроен да проследява отдалечен клон (вижте следващата секция и Клонове в Git за повече информация), можете да използвате командата `git pull` за автоматично изтегляне и сливане на данните от отдалечения клон в локалния. Това може да е по-лесно и по-удобно за вас като работна последователност и по подразбиране, `git clone` автоматично настройва локалния ви `master` да следи отдалечения `master` клон от сървъра, от който сте клонирали (или както се казва клонът по подразбиране на него). Така че `git pull` изтегля данните от сървъра, от който сте клонирали и автоматично се опитва да ги слее с кода, върху който работите в момента, спестявайки ви една ръчна стъпка по сливането.

From git version 2.27 onward, `git pull` will give a warning if the `pull.rebase` variable is not set. Git will keep warning you until you set the variable.

If you want the default behavior of git (fast-forward if possible, else create a merge commit): `git config --global pull.rebase "false"`

If you want to rebase when pulling: `git config --global pull.rebase "true"`

Изпращане на данни към отдалечено хранилище

Когато работата ви стигне до статус, в който искате да я публикувате с другите разработчици от даден проект, можете да изпратите промените си към главното хранилище. Командата за това е проста: `git push <remote> <branch>`. Ако искате да публикувате промените си от локалния `master` клон към `origin` сървъра (да кажем отново, клонирането обикновено настройва и двете кратки имена за вас автоматично), можете да изпълните следното:

```
$ git push origin master
```

Тази команда работи само ако, сте клонирали хранилището от отдалечен сървър, ако имате права за писане в него и ако никой междувременно не е изпратил към него нещо преди вас. Ако вие и някой друг сте клонирали едно и също хранилище и ако другият е изпратил обратно данни преди вас, то вашето изпращане правилно ще бъде отказано. Ще трябва първо да изтеглите работата на другия колега, да я слеете в локалното си копие и след това ще можете да изпратите към отдалечения сървър. Вижте Клонове в Git за повече подробности относно това как да изпращате към отдалечен сървър.

Преглед на отдалечено хранилище

Ако желаете повече информация за отдалечено хранилище, ползвайте командата `git remote show <remote>`. Ако я изпълните с определено кратко име като `origin` например, ще видите следното:

```
$ 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)
```

Показва се адреса на отдалеченото хранилище, както и информация за проследяването на клоновете. Командата подсказва, че ако сте в локалния `master` клон и изпълните `git pull`, това автоматично ще го слее с промените в отдалеченото хранилище след изтеглянето на всички отдалечени референции. Тя също така отпечатва и всички отдалечени референции, които са издърпани.

Това е прост пример, който е вероятно да срещнете. Когато ползвате Git по-интензивно обаче, може да видите и доста повече данни от `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)
```

Тази команда показва към кой отдалечен клон ще се изпращат вашите промени, когато изпълните `git push` докато сте в определен локален клон. Тя още ви показва отдалечените

клонове на сървъра, които вие все още нямате локално при вас, локално съхранените от преди клонове, които вече не съществуват на сървъра, и множество локални клонове, които могат да се слеят автоматично със съответните си отдалечени такива (които следят) при изпълнение на `git pull`.

Премахване и преименуване на отдалечени хранилища

Използвайте `git remote rename` за смяна на краткото име на отдалечено хранилище. Например, ако искате да смените името на `pb` с `paul`:

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

Това също променя и имената, под които се показват отдалечените проследявани клонове. Този, който преди се казваше `pb/master` сега е `paul/master`.

Ако по някаква причина искате да премахнете отдалечено хранилище, например сменили сте сървъра или пък някое от огледалата или пък определен сътрудник не участва повече в проекта — можете да използвате командите `git remote remove` или `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

Веднъж след като премахнете референция към отдалечено хранилище по този начин, всички remote-tracking клонове и конфигурационни настройки асоциирани с него, също се премахват.

Тагове в Git

Подобно на повечето VCS системи, Git позволява да маркирате (тагвате) специфични точки от историята на хранилището като важни. Обикновено това се използва за маркиране на различни версии на проекта (v1.0, v2.0 и т.н.). В тази секция, ще научите как да показвате наличните тагове, да създавате и премахвате тагове и да ги различавате по тип.

Показване на таговете

Показването на налични тагове в Git е просто. Просто напишете `git tag` (с опционални параметри `-l` или `--list`):

```
$ git tag
v1.0
v2.0
```

Тази команда отпечатва таговете по азбучен ред, редът по който са изобразени няма реално значение.

Можете също да търсите тагове по определен стринг. Хранилището на Git например, съдържа повече от 500 тага. Ако се интересувате само от версиите 1.8.5, можете да изпълните следното:

```
$ 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
```

Използването на wildcards изисква параметъра `-l` или `--list`

Ако просто искате целия списък тагове, изпълнението на командата `git tag` без параметри изрично подразбира, че желаете списък и го показва, в този случай използването на `-l` или `--list` е по желание

Ако обаче подадете wildcard маска за търсене на имена на тагове, тогава `-l` или `--list` са задължителни.

Създаване на тагове

Git поддържа два основни типа тагове: *lightweight* и *annotated*.

Lightweight тагът прилича на branch, който не се променя — това е просто указател към специфичен къмит.

Annotated таговете обаче, се съхраняват като пълни обекти в базата данни - те съдържат имейла и името на тагващия, дата, описателно съобщение и дори могат да се подписват и проверяват с GNU Privacy Guard (GPG). Хубаво е да се създават annotated тагове, защото тогава съхранявате всичката описана информация за тях, но ако искате временен такъв или по някаква причина не искате да пазите подробните описания, lightweight таговете също са вариант.

Annotated тагове

Създаването на анотиран таг в Git е лесно. Най-лесният начин е да подадете флага `-a`, когато пускате командата `tag`:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

Флагът `-m` указва съобщението за тага, което ще се съхранява заедно с него. Ако не укажете такова, Git ще стартира редактора ви, така че да можете да го напишете, точно както при къмитите.

Можете да разгледате данните за тага с къмита, който е бил тагнат с командата `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

    Change version number
```

Тя показва информация за тагващия разработчик, датата на която къмитът е бил тагнат и съобщението на тага преди да покаже информацията за самия къмит.

Lightweight тагове

Другият начин да тагвате даден къмит е с lightweight таг. Това най-просто казано е чексумата на къмита записана във файл - не се съхранява друга информация. За да създадете такъв олекотен таг, не подавайте флаговете `-a`, `-s`, или `-m` - просто укажете името на тага:

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

Този път, ако пуснете `git show` за тага, няма да виждате допълнителна информация за него. Командата просто показва къмита:

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

    Change version number
```

Тагване на предишни къмити

Можете да тагвате къмити и след като сте приключили с тях. Представете си, че историята на вашите къмити изглежда така:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Да предположим, че сте забравили да тагнете проекта като версия v1.2, която е трябвало да бъде маркирана в къмита регистриран като “Updated rakefile”. Можете да го направите и в по-късен момент. За да тагнете този стар къмит, подайте чексумата му (или част от нея) в края на командата:

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

Сега можете да проверите дали сте тагнали къмита успешно:

```
$ 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

    Update rakefile
...
```

Споделяне на тагове

По подразбиране, `git push` не изпраща таговете към отдалечените сървъри. Ще трябва ръчно да ги изпратите след като сте ги създали. Този процес е точно като споделяне на отдалечени клонове — можете да изпълните `git push origin <tagname>`.

```
$ 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
```

Ако имате много тагове и искате да ги изпратите наведнъж, подайте на командата флага `--tags`. Това ще трансферира всички ваши тагове, които не са били налични на сървъра наведнъж.

```
$ 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
```

Сега, ако някой клонира или издърпва от вашето хранилище, ще получи и таговете ви.

> `git push` *публикува и двата вида тагове*
>
> Публикуването на тагове с `git push <remote> --tags` изпраща и lightweight и annotated таговете. В момента няма начин за изпращане само на lightweight тагове, но ако използвате командата `git push <remote> --follow-tags`, тогава към отдалеченото хранилище ще бъдат изпратени само annotated таговете.

Изтриване на тагове

За да изтриете таг от локалното си хранилище, може да използвате командата `git tag -d <tagname>`. Например, можем да изтрием lightweight тага отгоре така:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

Това обаче не изтрива тага от никой отдалечен сървър. Съществуват два начина за изтриване на таг от отдалечен сървър.

Първият е да използвате `git push <remote> :refs/tags/<tagname>`:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]         v1.4-lw
```

Начинът да изтълкувате горното странно изписване е да го възприемете като изпращане на нулева стойност преди двуеточието към името на отдалечения таг, което ефективно го изтрива.

Вторият (и по-интуитивен) начин е с команда като тази:

```
$ git push origin --delete <tagname>
```

Извличане по тагове

Ако искате да видите версиите на файловете, към които сочи даден таг, можете да направите git checkout на този таг, въпреки че това ще постави хранилището в статус

“detached HEAD”, което има някои неприятни странични ефекти:

```
$ git checkout v2.0.0
Note: switching to 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout v2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-
final
HEAD is now at df3f601... Add atlas.json and cover image
```

В режим “detached HEAD”, ако направите промени и след това създадете къмит, тагът ще остане същия, но новия ви къмит няма да принадлежи към никой клон и няма да бъде достъпен освен по точния хеш на къмита. Затова, ако трябва да правите промени, например да поправите грешка в стара версия например — вероятно ще искате да създадете клон:

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

Разбира се, ако направите това и направите къмит, вашият `version2` клон ще бъде леко различен от тага `v2.0.0`, защото ще се премести напред с вашите промени, така че бъдете внимателни.

Псевдоними в Git

Преди да приключим с тази глава за основите на Git, има още една малка секция, която може да направи работата ви по-лесна и удобна: псевдонимите (aliases). По-късно в книгата няма да се обръщаме към тях или да считаме, че сте ги овладяли, но е хубаво да знаете как да ги ползвате.

Git не допълва автоматично командите ви докато ги пишете. Ако не искате да пишете целия текст за дадена команда, можете лесно да си създадете съкратен псевдоним за нея с помощта на `git config`. Ето няколко примера, които може да намерите за полезни:

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

Съгласно този пример, вместо да пишете `git commit`, можете да напишете `git ci`. Колкото повече започнете да ползвате Git, толкова по-вероятно е да искате нещо подобно, така че - не се притеснявайте да си създавате нови псевдоними.

Тази техника също така ви позволява да си създавате команда, която не съществува, а ви се иска да я имате под ръка. Например, за коригиране на не много удобния похват за де-индексиране на файл:

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

Това ще направи следните две команди еквивалентни:

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

Което изглежда една идея по-чисто. Често потребителите добавят команда `last`, нещо подобно на това:

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

По този начин лесно можете да видите последния къмит:

```
$ 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>
```

Както се вижда, Git просто заменя новата команда със съдържанието на реалната такава. Обаче, може да искате да изпълните външна команда, вместо подкоманда на Git. В този случай, стартирате командата със символа `!`. Това е полезно, ако пишете собствени инструменти за работа с Git хранилища. Можем да демонстрираме като асоциираме `git`

`visual` към псевдонима `gitk`:

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

Обобщение

В този момент, вие можете да правите всички базисни локални Git операции — създаване и клониране на хранилища, промени, подготвяне и публикуване на тези промени, и разглеждане на историята на промените в проекта във времето. Следва да разгледаме най-полезния аспект на Git - branching модела.

Клонове в Git

Почти всички VCS системи разполагат с поддръжка на разклонения на версиите на кода под някаква форма. Разклоняването на кода означава, че вие се отделяте от основната линия на разработка (във ваш собствен клон, branch) и продължавате да работите без да се намесвате в тази основна линия. В множеството VCS системи това е процес, който изисква много ресурси и често сте принудени да копирате цялата си директория със сорс код, което може да е бавно при големи проекти.

Някои хора определят branching модела на Git като неговата най-силна черта и в действителност това е едно от нещата, които помагат на Git да изпъква сред другите VCS системи. Кое е толкова специално? Начинът, по който Git имплементира клоновете код е изключително олекотен, което прави branching операциите почти мигновени - това важи със същата сила и за превключването напред и назад по различните клонове код без оглед на мащаба на проекта. За разлика от другите VCS системи, Git окуражава работните процеси съдържащи чести разклонявания и сливания - дори по няколко пъти на ден. Ако успеете да овладеете тази страна на Git, ще разполагате с един мощен и уникален инструмент, който значително ще подобри и улесни методите ви на разработка.

Накратко за разклоненията

(Под клон, разклонение и branch ще имаме предвид едно и също нещо.) За да разберем как в действителност Git реализира разклоненията, трябва да се върнем стъпка назад и да си припомним как Git съхранява своите данни.

Както може би помните от Какво е Git, Git не съхранява информацията си като серии от промени или разлики, а вместо това пази серии от моментното състояние на проекта - *snapshots*.

Когато правите къмит, Git съхранява един commit обект, който съдържа указател към snapshot-а на индексираното съдържание (това, което е в индексната област). Този обект също така съдържа името и имейла на автора, къмит съобщението и също така - указатели към къмита или къмитите, които са правени директно преди текущия къмит (тоест, неговите родител/родители): първоначалният къмит няма родители, нормалният къмит има един родител, а къмитът създаден в резултат от сливане на няколко клона има множество родители.

За да илюстрираме това, нека допуснем, че имате директория с три файла и сте ги индексирали и къмитнали. Процесът по индексирането на файловете (staging) изчислява чексума за всеки от файловете (това е SHA-1 хеш стрингът за който говорихме по-рано в Какво е Git), записва версията на всеки файл в хранилището (Git третира файловете като blob-обекти) и добавя чексумите в индексната област (staging area):

```
$ git add README test.rb LICENSE
$ git commit -m 'Initial commit'
```

След като изпълните `git commit`, Git изчислява чексума за всяка поддиректория (в този

случай само основната директория на проекта) и ги съхранява като дървовиден обект в Git хранилището. След това Git създава commit-обект, който съдържа метаданните и указател към root-дървото на проекта, така че да може да пресъздаде snapshot-а (тоест йерархията от файлове и директории) по-късно, когато е необходимо.

Вашето Git хранилище сега съдържа 5 обекта: по един *blob* за всеки от трите файла, едно *дърво* описващо съдържанието на директорията и указващо кой файл под формата на кой blob се съхранява, и един *къмит* с указател към това основно дърво и всички метаданни за къмита.

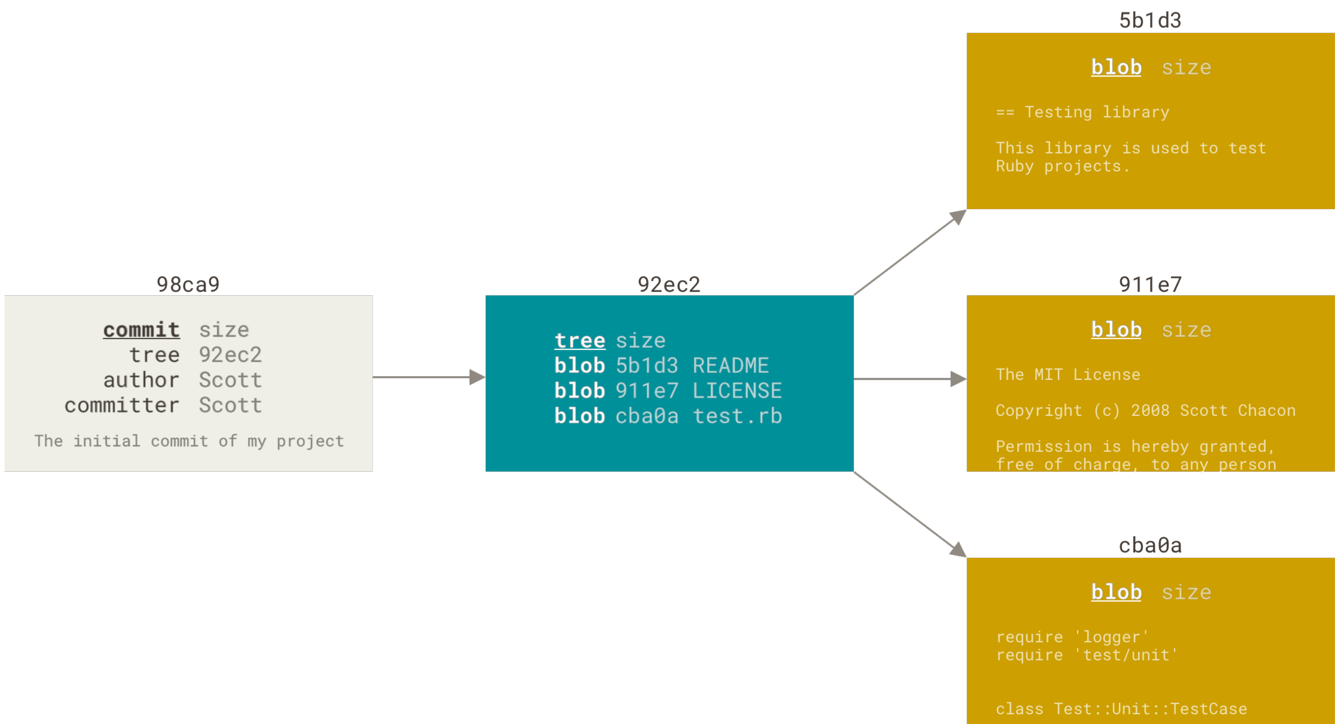


Figure 9. Един къмит и неговото дърво

Ако направите някакви промени и къмитнете отново, следващият къмит ще съхранява указател към къмита направен веднага преди него.

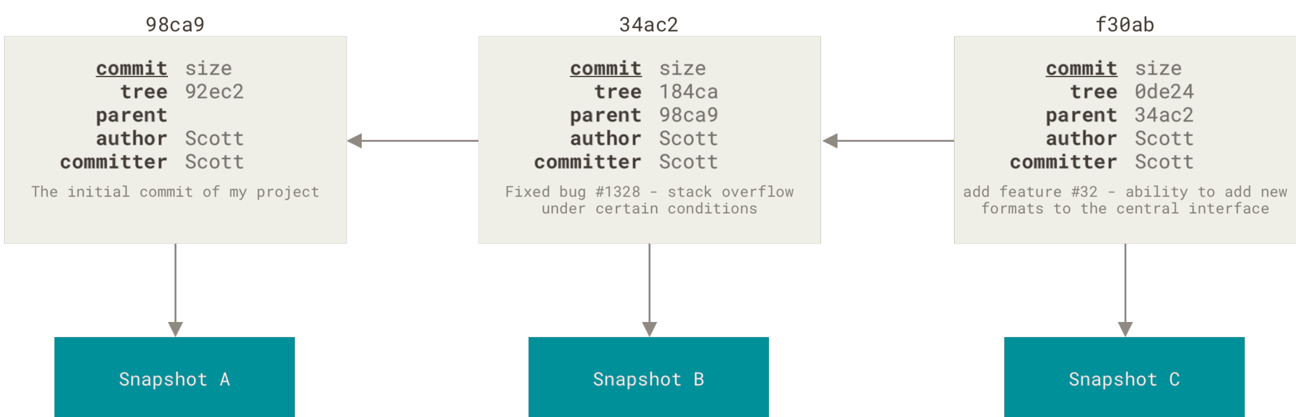


Figure 10. Къмити и родителски къмити

Разклонението код (branch) в Git е просто олекотен, променлив указател към един от тези къмити. Името на разклонението по подразбиране за Git е `master`. Когато започнете да правите къмити, вие разполагате с `master` branch, който сочи към последния къмит, който

сте направили. Всеки път, когато къмитвате, той автоматично се премества и сочи към последния къмит.

“master” клонът в Git не бива да се разглежда като специален такъв. Той е подобен на всички останали клонове. Единствената причина почти всяко хранилище да има master клон е, че командата `git init` го създава по подразбиране и повечето хора не си правят труда да му сменят името.

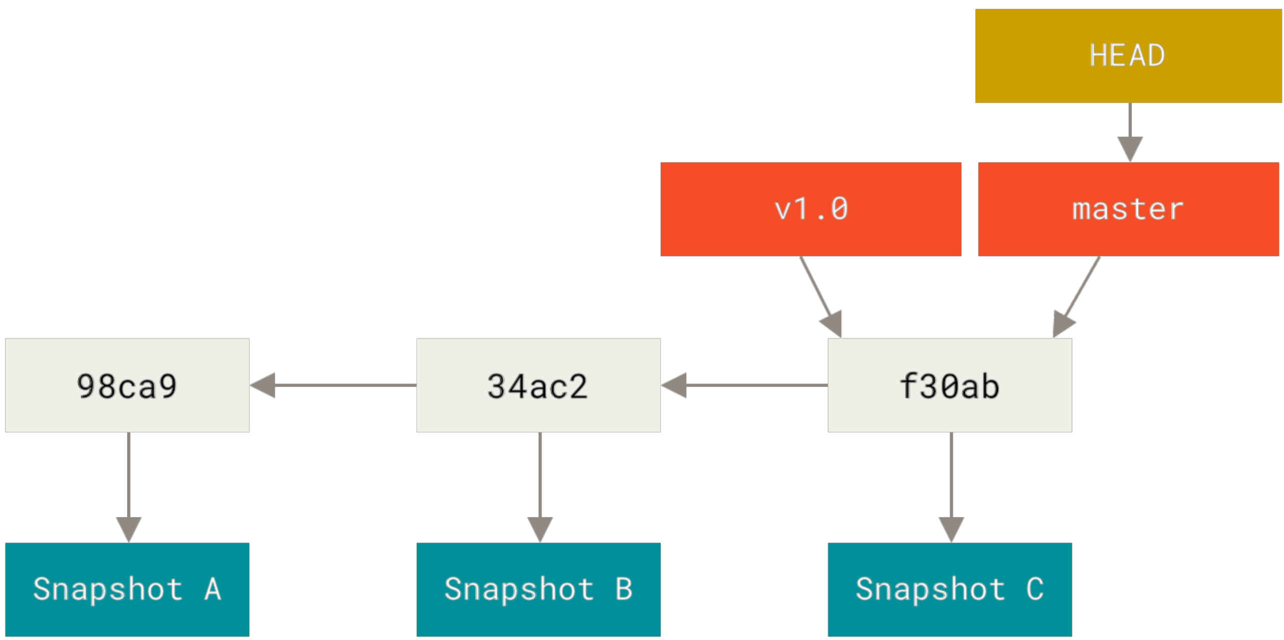


Figure 11. branch и неговата commit история

Създаване на ново разклонение

Какво се случва, когато създадете нов клон? Git просто създава нов указател за вас, който да може да се премества. Да кажем, че създавате клон с име `testing`. Това се прави с командата `git branch`:

```
$ git branch testing
```

Това създава нов указател към същия къмит, на който сте в момента.

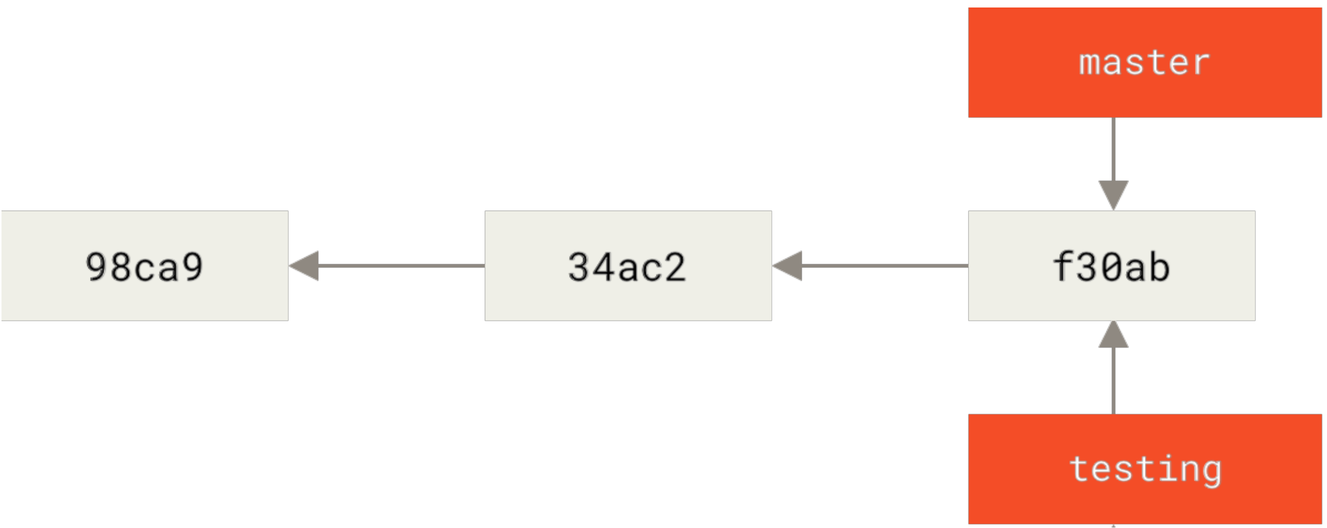


Figure 12. Два клона сочещи към една и съща серия къмити

Как Git знае в кой клон сте в даден момент? Системата си пази специален указател, който се нарича `HEAD`. Отбележете, че това е съвсем различно от `HEAD` концепциите в други VCS от рода на Subversion или CVS. В Git, това е указател към текущия локален клон от хранилището ви. В този случай, вие още сте в `master` клона. Това е така, защото `git branch` командата само *създаде* новия клон, но не превключи към него.

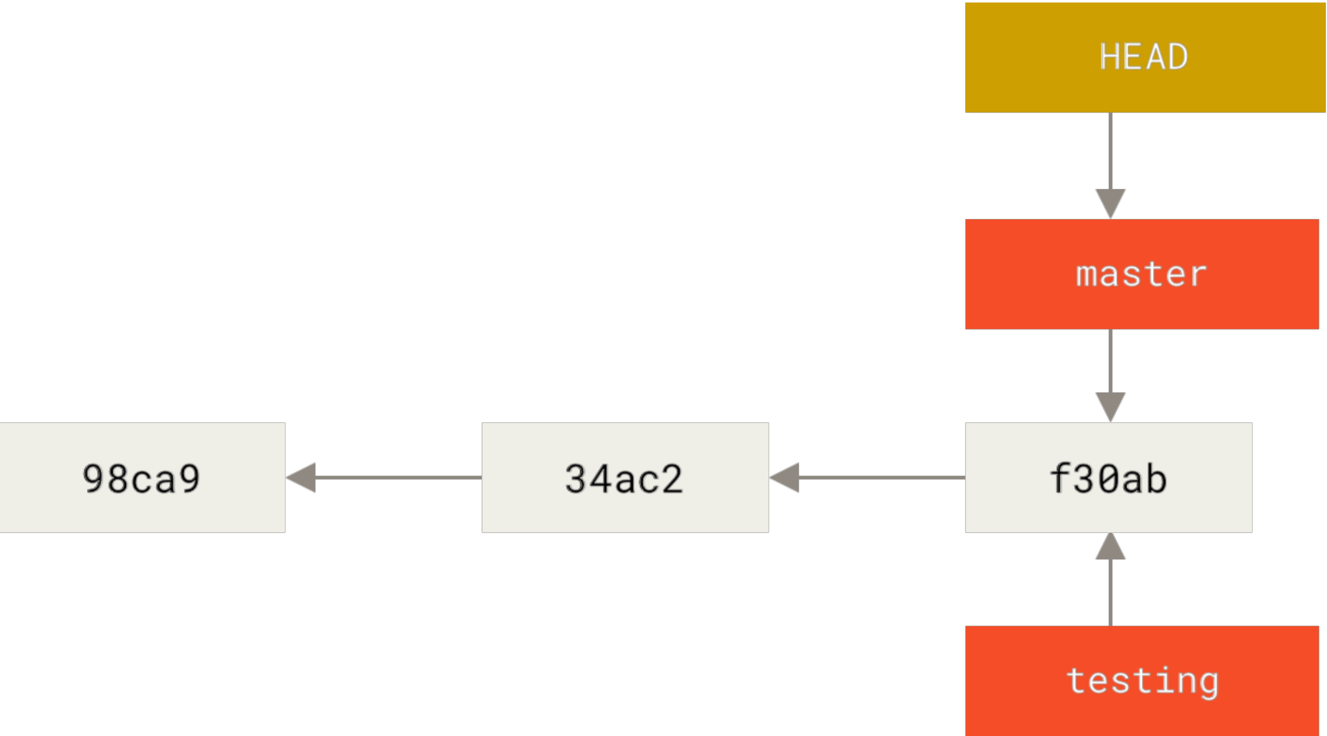


Figure 13. HEAD указател сочещ към текущия branch

Можете лесно да видите това изпълнявайки командата `git log --decorate`, която отпечатва накъде сочат указателите на разклоненията.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the
central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

Виждате `master` и `testing` клоновете веднага до `f30ab`-къмита.

Превключване на разклонения

За да превключите към съществуващ клон, изпълнете командата `git checkout`. Нека превключим към `testing` клона:

```
$ git checkout testing
```

Това премества `HEAD` указателя и сега той сочи към `testing` клона.

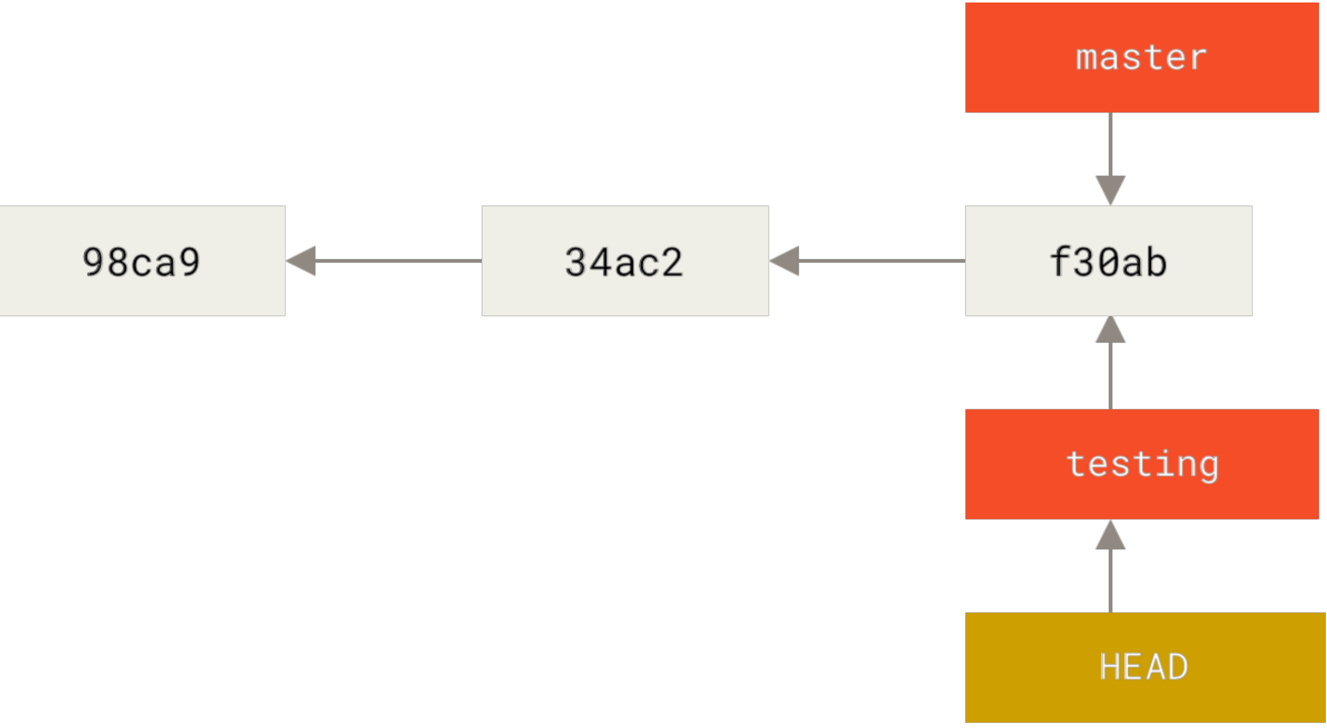


Figure 14. HEAD сочи към текущия клон

Какво означава това? Нека направим още един къмит:

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

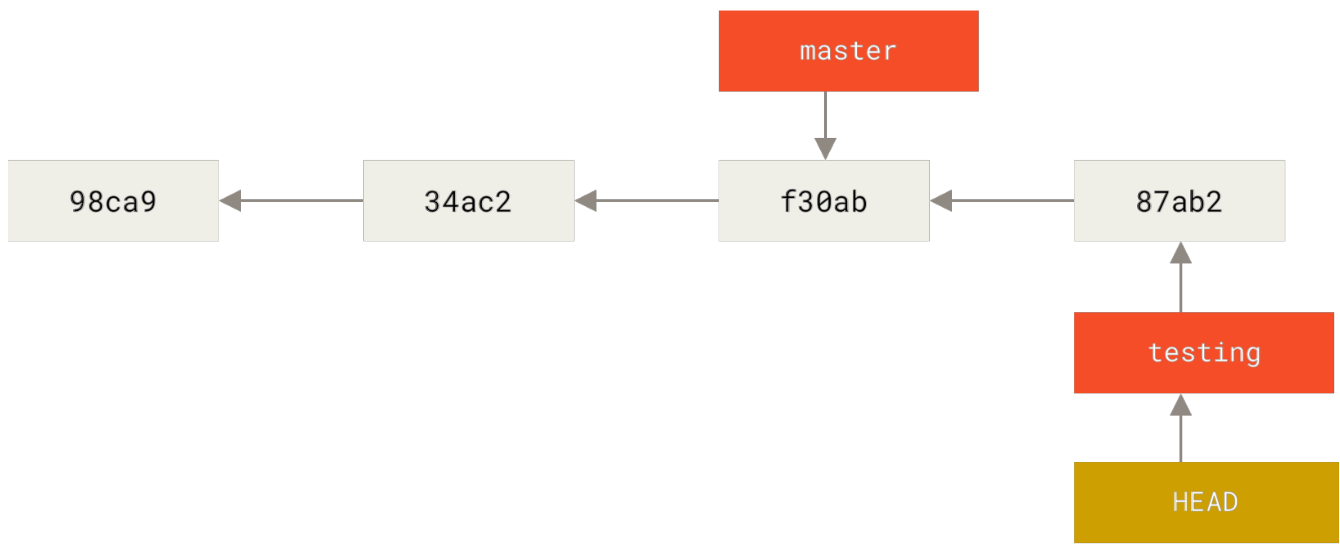


Figure 15. HEAD клонът се премества напред при направен къмит

Това е интересно, защото сега вашият `testing` клон се премести напред, но `master` клонът все още сочи към къмита, в който бяхте когато изпълнихте `git checkout` за да превключите разклоненията. Нека се върнем отново на `master` клона:

```
$ git checkout master
```

> *`git log` не показва всички клонове постоянно*
>
> Ако изпълните `git log` сега, може да се зачудите къде е изчезнал току що създадения "testing" клон, понеже той няма да се покаже в изхода.
>
> Клонът не е изчезнал, Git просто не знае, че се интересувате от него и се опитва да ви покаже това, което той мисли, че търсите. С други думи, по подразбиране, `git log` ще показва само историята на къмитите в клона, който е активен в момента.
>
> Ако желаете историята на конкретен клон, ще трябва да го укажете изрично `git log testing`. За да покажете всички клонове, използвайте `git log --all`.

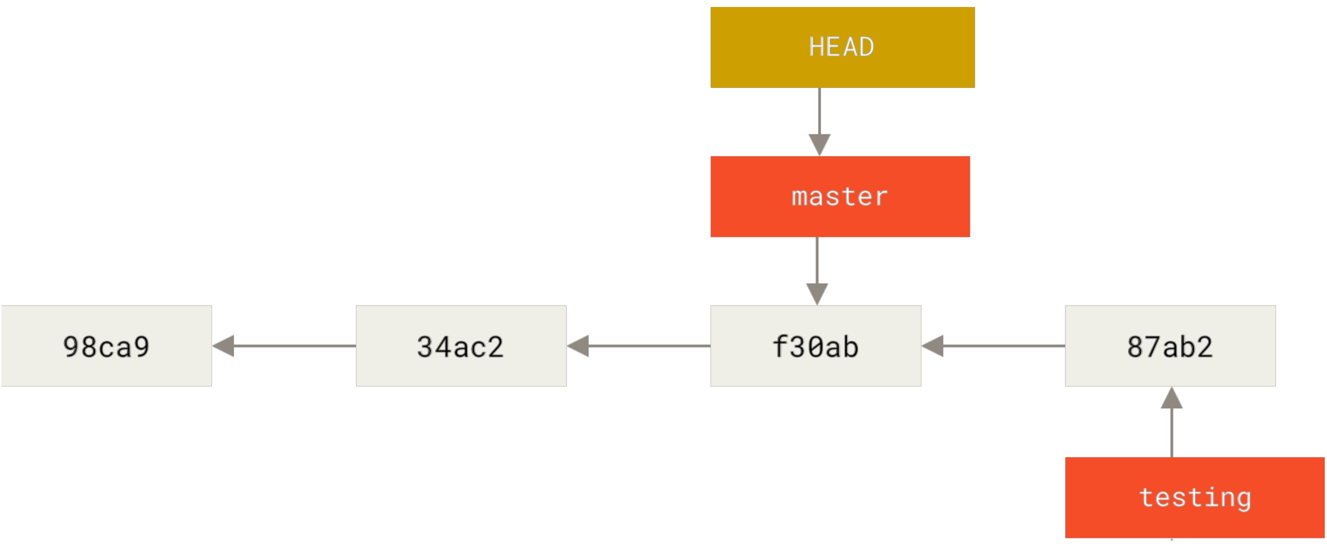


Figure 16. HEAD се премества когато превключвате

Тази команда направи две неща. Тя премести HEAD указателя обратно към точката на `master` клона - и също така върна обратно статуса на всички файлове в работната ви директория така че те сега съдържат това, което са съдържали в момента на последния къмит в `master` клона. Това също означава, че промените които предстои да правите занапред от тази точка на проекта, ще произлизат от по-стара негова версия. Практически - връщането в `master` клона заличи всички промени от `testing` клона в работната директория и сега можете да тръгнете в различна посока.

Превключването между клоновете променя файловете в работната директория

Важно е да се посочи, че когато превключвате клонове в Git, файловете в работната директория ще се променят. Ако превключите към по-ранен клон, вашата работна директория ще се превърти назад във времето и ще съдържа това, което е имала последния път когато сте направили къмит в този клон. Ако Git не може да направи това безпроблемно - то превключването няма да бъде позволено въобще.

Нека направим няколко промени и да къмитнем отново:

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

Сега историята на проекта се отклони (виж Разклонена история). Вие създадохте и превключихте към нов клон, направихте промени по кода в него, превключихте към основния клон и направихте други промени. И двата вида промени са изолирани в отделни разклонения: можете да превключвате между тези разклонения и да ги слеете в едно, когато сте готови. Направихте всичко това с прости команди като `branch`, `checkout`, и `commit`.

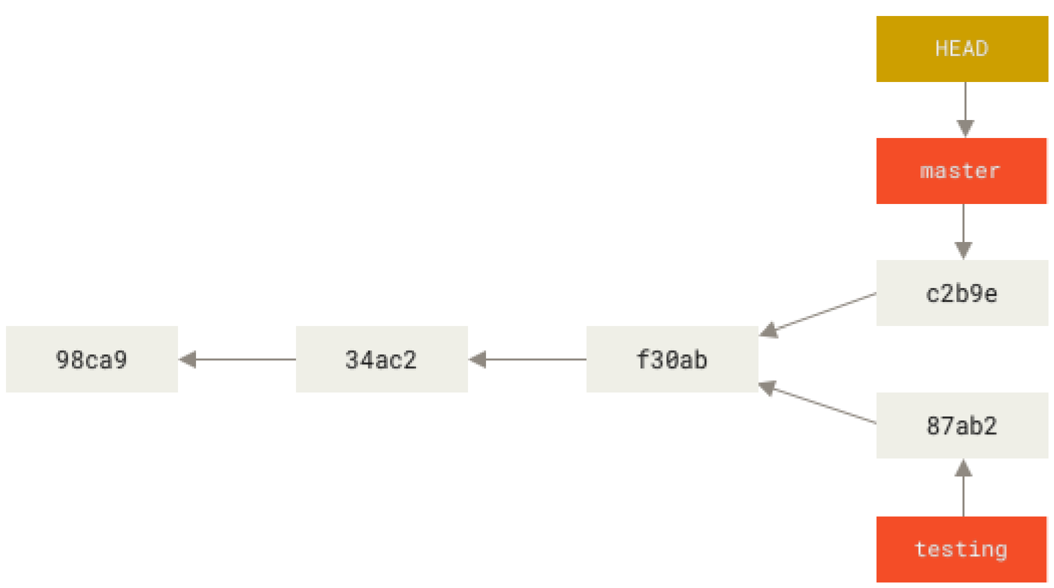


Figure 17. Разклонена история

Можете лесно да видите това и с командата `git log`. Ако изпълните `git log --oneline --decorate --graph --all`, това ще отпечата историята на вашите къмити, показвайки къде са вашите branch указатели и как се е разклонила историята на проекта.

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

Понеже клонът в Git на практика е обикновен файл съдържащ 40-символна SHA-1 чексума на къмита, към който клонът сочи, създаването и изтриването на разклонения в Git почти не изисква ресурси. Създаването на нов клон е толкова бързо и просто колкото е записа на 41 байта във файл (40-те символа от чексумата и символ за нов ред).

Това рязко контрастира с начина, по който повечето стари VCS системи управляват разклоненията, защото те копират всички файлове на проекта ви в отделна директория. При тях това може да отнеме много секунди и дори минути, според размера на проекта, докато при Git се прави почти мигновено. Също така, понеже записваме родителите когато къмитваме, намирането на правилната базова точка за сливане се осъществява автоматично за нас и е много лесно. Тези функции окуражават разработчиците да създават и използват разклонения колкото може по-често.

Нека видим защо е добре да го правим.

Създаване на нов клон и превключване към него автоматично

Често се случва да искате да превключите веднага към новосъздаден клон — това може да стане на една стъпка с командата `git checkout -b <newbranchname>`.

From Git version 2.23 onwards you can use `git switch` instead of `git checkout` to:

- Switch to an existing branch: `git switch testing-branch`.
- Create a new branch and switch to it: `git switch -c new-branch`. The `-c` flag stands for create, you can also use the full flag: `--create`.
- Return to your previously checked out branch: `git switch -`.

Основи на клоновете код и сливането

Нека илюстрираме разклоняването и сливането с малък пример, какъвто може да срещнете в реалния живот. Ще следваме следните стъпки:

1. Работите по уеб сайт.
2. Създавате нов клон за нова статия, по която работите.
3. Извършвате някакви дейности по този клон.

В този момент, получавате обаждане за внезапно възникнал критичен проблем в друга част от сайта, който трябва да решите бързо. Ще направите следното:

1. Превключвате към работния (production) клон.
2. Създавате нов клон и решавате проблема в него.
3. След тест, че всичко в поправката е наред, сливате hotfix клона обратно в работния клон.
4. Превключвате отново към клона с новата статия и продължавате работа.

Основи на разклоняването

Първо, нека приемем, че работите по проекта си и вече имате няколко къмита в клона `master`.

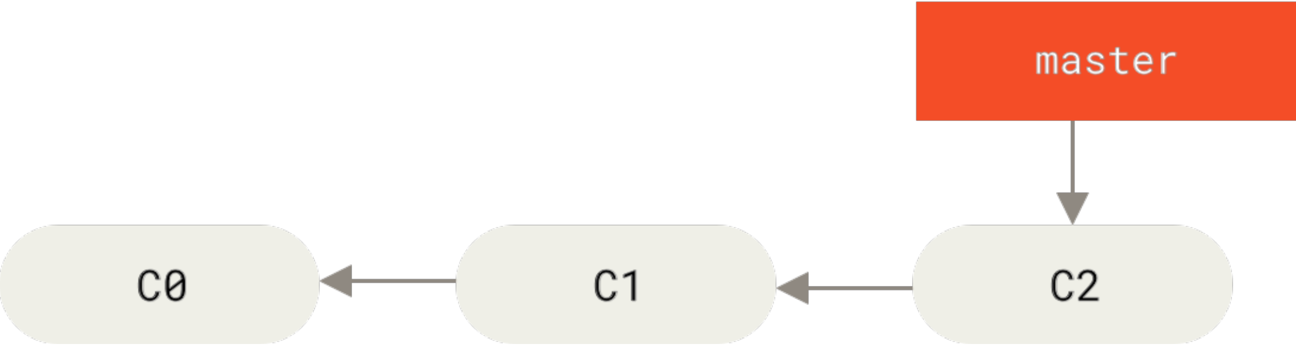


Figure 18. Проста история на къмитите

Решили сте, че трябва да работите по проблем #53 в issue-tracking системата, която ползва вашата компания. За да създадете клон и превключите към него в същия момент, изпълнете командата `git checkout` с параметър `-b`:

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

Това е съкратена версия на командите:

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

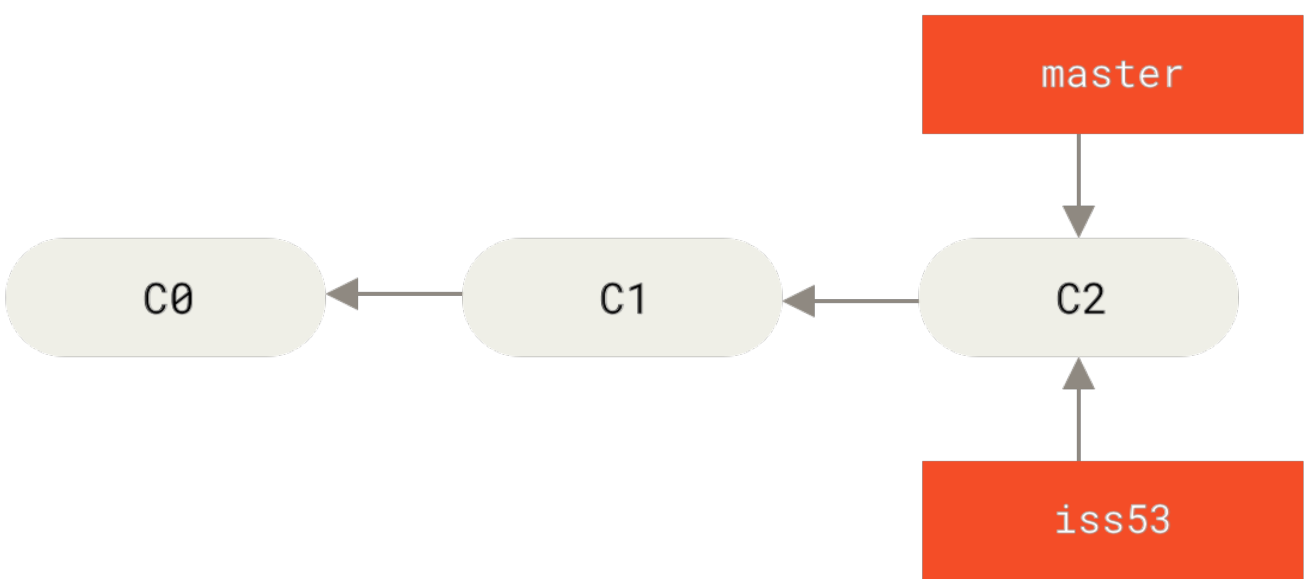


Figure 19. Създаване на нов указател към branch

Вие си работите по сайта и правите няколко къмита. По време на този процес, клонът `iss53` се премества напред, защото е текущ (това означава, че `HEAD` указателят сочи към него):

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

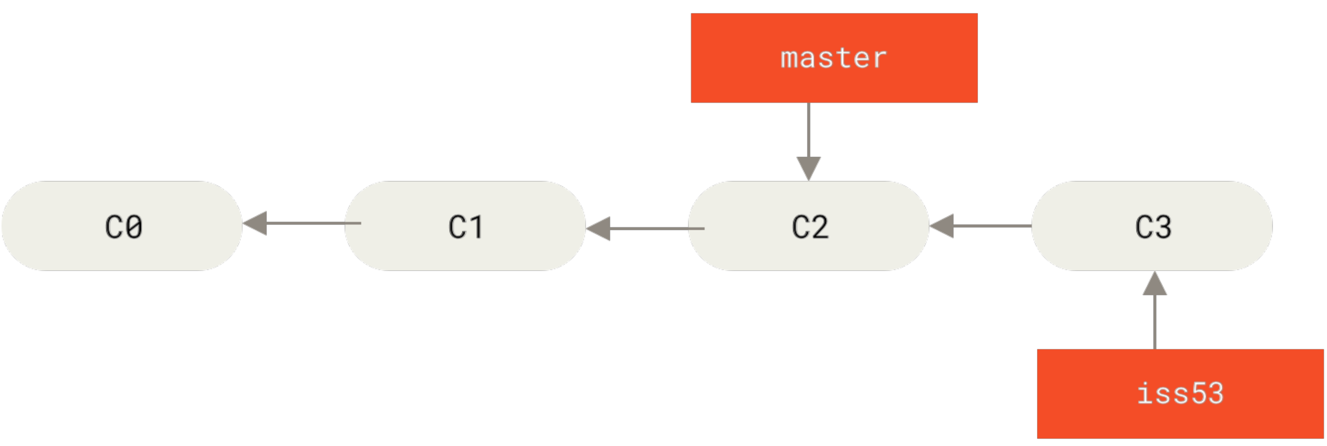


Figure 20. Клонът `iss53` се е преместил напред в процеса на работа

Сега получавате обаждане, че нещо не е наред с уебсайта и трябва да го оправите незабавно. С Git, не е нужно да прилагате поправката заедно с промените, които се съдържат в `iss53` клона и не е нужно да влагате усилия по отмяната на тези промени преди да можете да приложите спешната поправка в production версията на сайта. Всичко, което трябва да направите е да превключите обратно към `master` клона.

Обаче преди да направите това, отбележете, че ако работната ви директория или индексната област съдържат некъмитнати промени, които влизат в конфликт с клона, към който превключвате, Git няма да позволи превключването на клоновете. Най-добре е да имате чист работен статус преди превключването. Съществуват начини да заобиколите това (известно като stashing и commit amending), които ще разгледаме по-късно в Stashing и Cleaning. Засега, нека приемем, че сте къмитнали промените си, така че може да се върнете в `master` клона:

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

В този момент, работната ви директория ще се превърти обратно до съдържанието, което е имала преди да започнете работа по проблем 53 и можете да се концентрирате в спешната поправка, която е необходимо да въведете. Това е важен момент за запомняне: когато превключвате между клоновете, Git връща работната директория до статуса, в който е била последния път, когато сте къмитнали в този клон. Системата добавя, изтрива и променя файловете автоматично, за да ви предостави работното копие на обектите в момента на последния ви къмит.

След това, имате да правите спешната поправка. Нека създадем един `hotfix` клон, по който да работим докато тя стане готова:

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

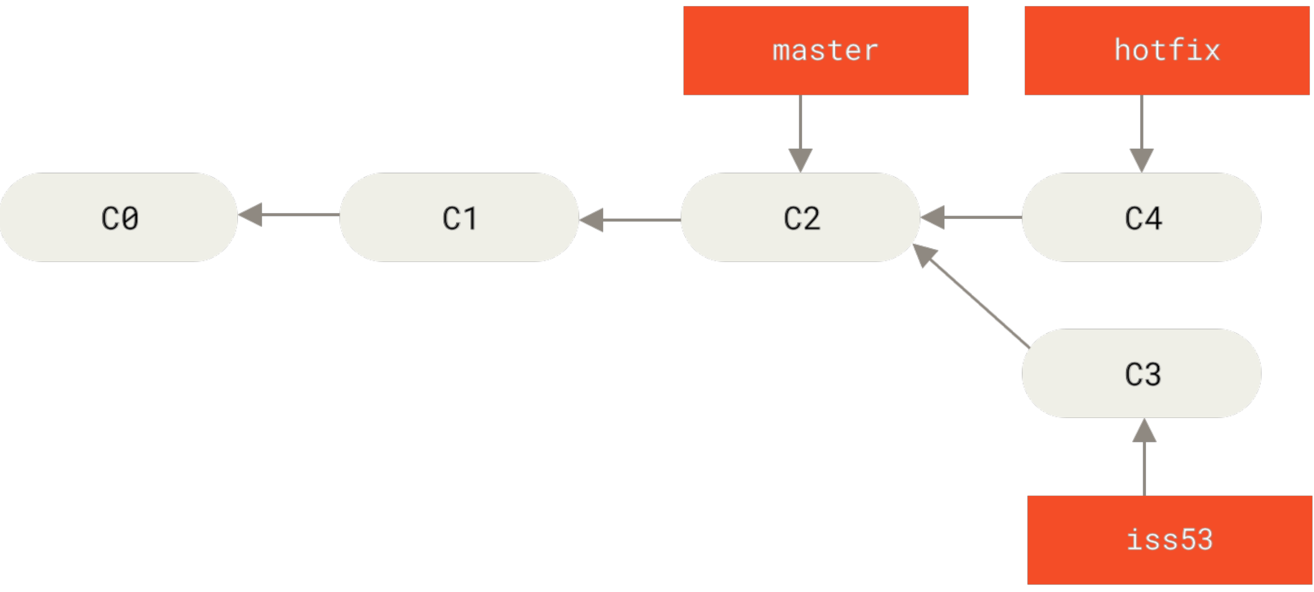


Figure 21. Hotfix клон произлизащ от `master` клона

Можете да пускате тестовете си, да се уверите, че поправката работи както се очаква и да слеете обратно вашия `hotfix` клон в `master` клона за да го пуснете в работния вариант. Това се прави с командата `git merge`:

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

В съобщението от сливането ще забележите фразата “fast-forward”. Понеже къмитът `C4`, към който сочи клона `hotfix`, който сляхте, беше директно след къмита `C2`, Git просто премества указателя напред. Казано по друг начин, когато се опитвате да слеете един къмит с друг такъв, който може да бъде достигнат следвайки историята на първия, Git опростява нещата премествайки указателя напред, защото не се налага да се върши работа по сливане на разклонен код. Това се нарича “fast-forward.”

Сега промяната ви е в snapshot-а на къмита сочен от `master` клона и можете да пуснете промяната в реалния сайт.

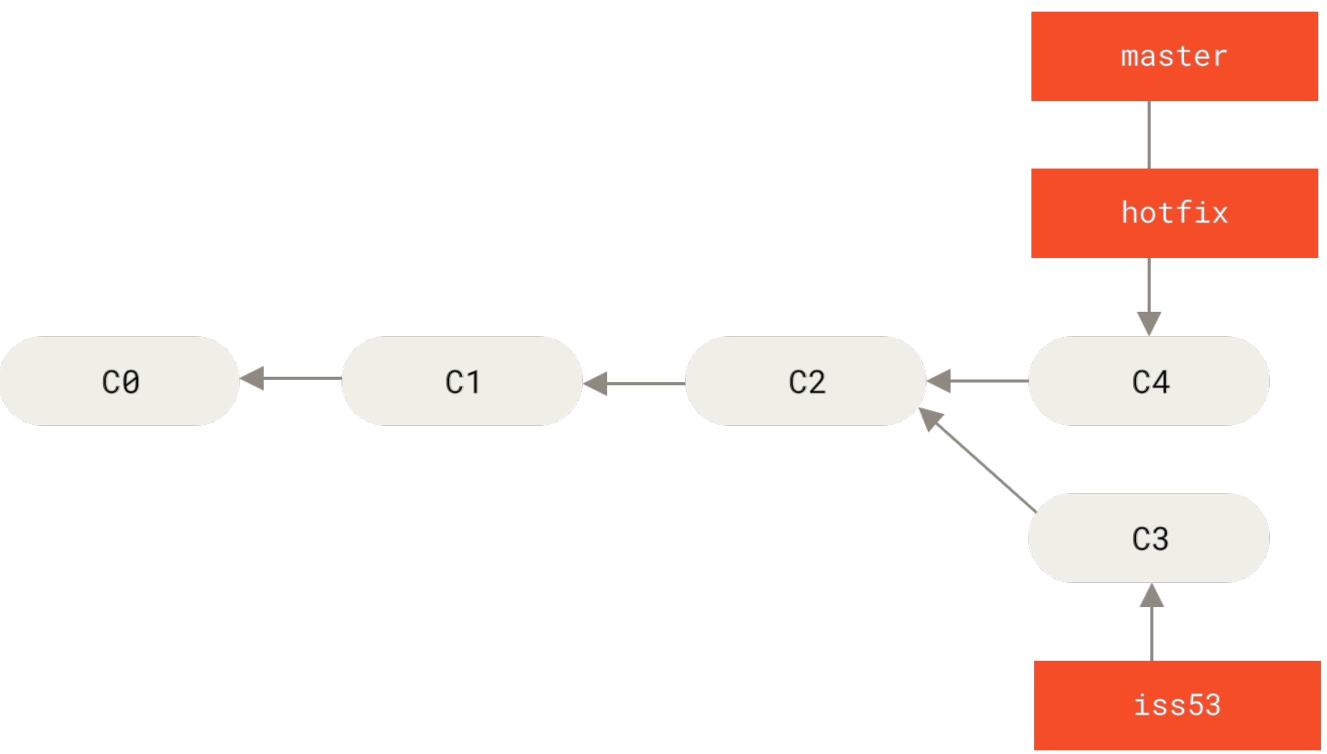


Figure 22. `master` е превъртян (fast-forwarded) към `hotfix`

След като суперважната промяна е въведена, можете да се върнете обратно към работата, която вършехте преди обаждането. Обаче, първо ще изтриете клона `hotfix`, понеже вече не ви е нужен — `master` клонът сочи към същото място. Изтриването се прави с параметъра `-d` на командата `git branch`:

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

Сега можете да се прехвърлите към клона, в който си вършите обичайната работа по проблем 53 и да си я продължите.

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

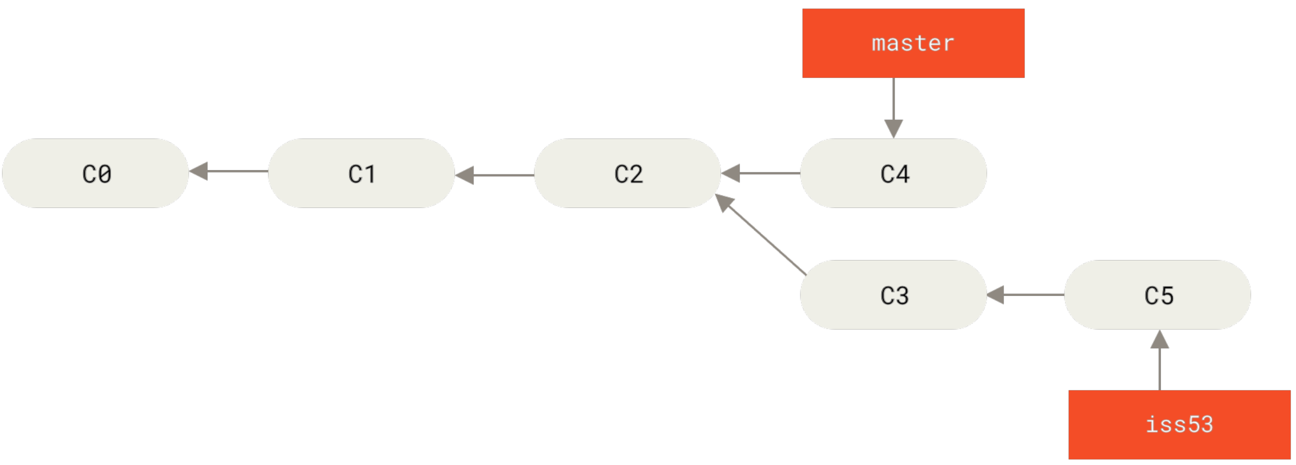


Figure 23. Работата продължава в клона `iss53`

Тук си струва да отбележим, че промените, които направихте в клона `hotfix` не се съдържат във файловете на клона `iss53`. Ако искате да ги имате, можете да слеете `master` клона в `iss53` изпълнявайки `git merge master`, или пък можете да изчакате с интегрирането на тези промени докато дойде момента, в който решите че е време да слеете `iss53` клона обратно в `master`.

Сливане

Да кажем, че сте решили, че работата по проблем 53 е свършена и сте готови да я слеете в `master` клона. За да направите това, ще действате по същия начин, по който го направихте с клона `hotfix` по-рано. Всичко, което трябва да направите е да превключите към клона, в който искате да сливате и да изпълните `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(+)
```

Това обаче изглежда по-различно от `hotfix` сливането. В този случай, историята на разработката се е отклонила от по-ранна точка. Понеже къмитът на клона, в който сте (C4), не е директен предшественик на клона, който сливате, Git ще има малко работа за вършене. В този случай, Git прави просто трипосочно сливане използвайки двата snapshot-а на клоновете и общия им предшественик (C2).

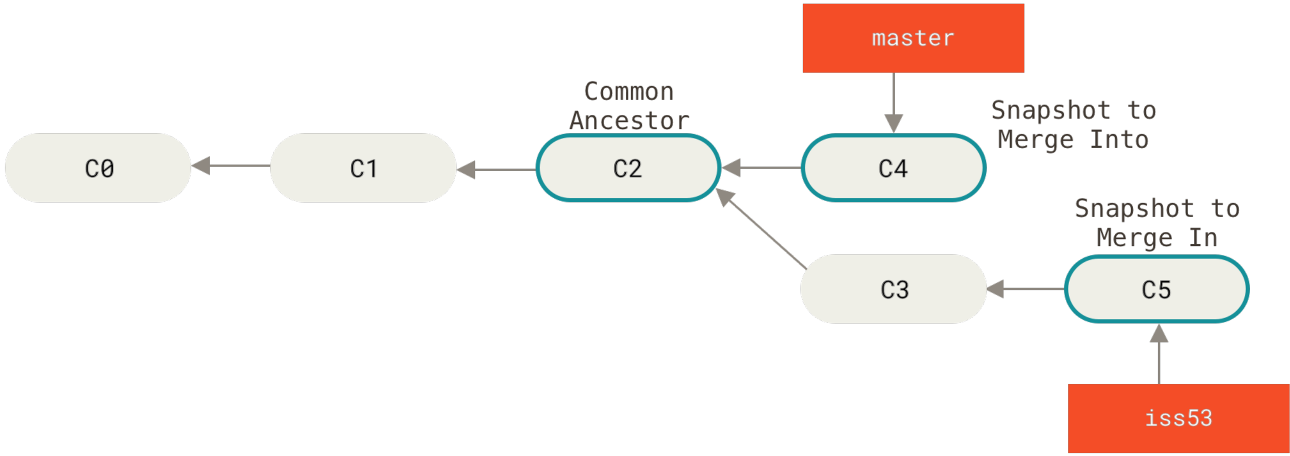


Figure 24. Три snapshot-а използвани в типично сливане

Това се нарича сливащ къмит (merge commit) и е специален заради това, че има повече от един родител. Вместо просто да премести указателя на клона напред, Git създава нов snapshot, който е резултат от това трипосочно сливане и автоматично създава нов къмит, който да сочи към него.

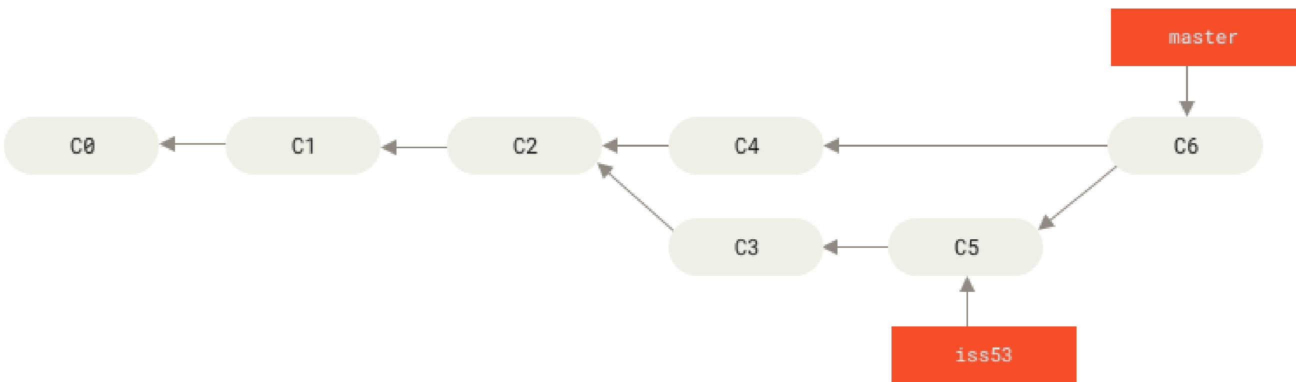


Figure 25. Сливащ commit

Сега, когато работата ви е слята, вече не се нуждаете от клона `iss53`. Можете да затворите проблема в issue-tracking системата и да изтриете клона:

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

Конфликти при сливане

Понякога обаче, сливането не минава гладко. Ако сте променили една и съща част от един и същи файл в различни клонове, които искате да слеете, Git няма да може да направи това чисто. Ако промените ви по проблем #53 са модифицирали една и съща част от файл с тези от клона `hotfix`, ще се получи конфликт при сливането, съобщението за който може да

изглежда по подобен начин:

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

Git не е извършил merge къмита. Процесът е прекъснат докато не разрешите конфликта Ако искате да видите кои файлове не са слети вследствие на конфликта, можете да изпълните `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")
```

Всичко, по което има конфликти при сливането се показва като unmerged. Git добавя стандартни маркери за разрешаване на конфликта към файловете, в които има такъв, така че можете да ги отворите ръчно и да ги коригирате. Файлът ви ще съдържа секция подобна на тази:

```
<<<<<<< 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
```

Това означава, че версията в `HEAD` (вашият `master` клон, понеже той е текущия, в който сте изпълнили командата по сливането) е горната част на този блок (всичко преди `=======`), докато версията в клона `iss53` е в долната част. За да решите конфликта, трябва да изберете една от двете или да слеете съдържанието сами. Например, можете да замените целия блок с това:

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

Това решение съдържа по малко от всяка секция и редовете `<<<<<<<`, `=======`, `>>>>>>>` се изтриват напълно. След като направите това за всяка секция и всеки файл в който има конфликти, изпълнете `git add` за всеки файл, за да го маркирате като коригиран. Индексирането на файл в Git го маркира като коректен, без конфликти.

Ако желаете да използвате графичен инструмент за решаването на конфликти, можете да изпълните `git mergetool`, което ще стартира подходящия визуален инструмент и ще ви води през конфликтите подред:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

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

Ако желаете да използвате инструмент различен от подразбиращия се (в случая Git е избрал `opendiff`, защото командата е пусната на Mac), можете да видите всички поддържани такива в горната част на изхода от командата след надписа “one of the following tools.” Просто напишете името на инструмента, който предпочитате.

Ако се нуждаете от по-модерни инструменти за разрешаване на заплетени конфликти, обръщаме повече внимание на това в Сливане за напреднали.

След като затворите инструмента, Git ви пита дали сливането е успешно. Ако кажете това на скрипта, системата ще индексира файла и ще го маркира като коректен. Можете да пуснете `git status` отново за да проверите дали всички конфликти са разрешени:

```
$ 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
```

Ако това ви устройва и сте проверили, че всичко по което е имало конфликти е в индекса, можете да изпълните `git commit` за да завършите сливащия къмит. Къмит съобщението по

подразбиране изглежда подобно на това:

```
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
#
```

Ако мислите, че за другите ви колеги гледащи кода по-късно ще е полезно да разберат как сте разрешили конфликта, можете да промените къмит съобщението с обяснение за това.

Управление на клонове

След като създадохме, сляхме и изтрихме няколко разклонения код, нека разгледаме някои инструменти, които ще бъдат от полза, когато започнете да ползвате разклоненията постоянно.

Командата `git branch` може да прави повече от създаване или изтриване на клонове. Ако я пуснете без аргументи, ще получите прост списък на наличните клонове код:

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

Звездичката преди името на клона означава, че той е текущо активния в момента (клонът към който сочи `HEAD` указателя). Това означава, че ако къмитнете в този момент, `master` клонът ще се придвижи напред отразявайки резултатите от вашата работа. За да видите последния къмит за всеки клон, изпълнете `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 readme
```

Полезните опции `--merged` и `--no-merged` могат да филтрират списъка до клоновете, които сте слели или все още не сте слели с текущия клон. За да видите кои клонове сте слели, изпълнете `git branch --merged`:

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

Понеже вече сляхте `iss53` по-рано, можете да го видите в списъка. Клоновете в този списък, които нямат звездичка пред името си могат безопасно да бъдат изтрити с `git branch -d`, защото вече сте интегрирали промените им в текущия клон и няма опасност да загубите нищо.

Обратно, ако искате да видите клоновете, които все още не сте слели с текущия, изпълнете `git branch --no-merged`:

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

Това показва другия ви клон. Понеже той съдържа работа, която все още не е слята, не можете да го изтриете с `git branch -d`:

```
$ 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'.
```

Ако все пак искате да изтриете клона заедно с промените по него, можете да форсирате изтриването с параметъра `D`, както подсказва съобщението.

Ако не сте посочили даден къмит или име на клон като аргументи, то опциите `--merged` и `--no-merged` ще ви покажат съответно какво е и какво не е слято в *текущия* клон.

Можете винаги да подадете допълнителен параметър за да попитате за merge статуса по отношение на друг клон без първо да превключате към него, например - какво не е слято в `master` клона.

```
$ git checkout testing
$ git branch --no-merged master
  topicA
  featureB
```

Преименуване на клон

Не преименувайте клонове, които все още се ползват от други сътрудници. Не преименувайте клонове като master/main/mainline без да сте прочели секцията "Преименуване на master клон".

Да кажем, че имате клон с име *bad-branch-name* и искате да го смените на *corrected-branch-name*, запазвайки цялата история. Искате също така да смените името на клона и в отдалечените хранилища (GitHub, GitLab или друг сървър). Как се прави това?

Преименувайте клона локално с командата `git branch --move`:

```
$ git branch --move bad-branch-name corrected-branch-name
```

Това променя името от bad-branch-name на corrected-branch-name, но промяната е само локална засега. За да го направите видим за колегите, изпратете го към сървъра така:

```
$ git push --set-upstream origin corrected-branch-name
```

Да видим къде сме сега:

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

Отбележете, че сте в клона corrected-branch-name. Коригираният клон е наличен на сървъра. Но предишния клон също е там. Можете да го изтриете така:

```
$ git push origin --delete bad-branch-name
```

Сега bad branch name е напълно заменен с corrected branch name.

Смяна на името на master клона

Смяната на имената на клонове като master/main/mainline/default ще счупи интеграциите, услугите, помощните инструменти и build/release скриптовете, които вашето хранилище ползва. Преди да го направите, уверете се, че сте се консултирали с колегите си. Също така, проверете внимателно хранилището и се уверете, че сте коригирали всички референции към старото име на клона в кода и скриптовете ви.

Преименувайте локалния *master* клон на *main* с командата:

```
$ git branch --move master main
```

Сега локално няма *master* клон, понеже той е преименуван на *main*.

За да могат и другите да видят това, трябва да изпратите новия *main* клон към сървъра. Така той става видим за колегите ви.

```
$ git push --set-upstream origin main
```

Сега клоновете изглеждата така:

```
git branch --all
* main
  remotes/origin/HEAD -> origin/master
  remotes/origin/main
  remotes/origin/master
```

Сега локалният *master* клон го няма и е заменен с *main*. Клонът *main* е също така наличен на сървъра. Сървърът обаче все още има *master* клон. Сътрудниците ще продължат да ползват *master* клона като база за своята работа, докато не направите следващи промени.

Сега имате още малко задачи преди да завършите промяната:

- Всички проекти, които зависят от този, ще трябва да обновят кода и конфигурациите си.
- Обновяване на всички test-runner конфигурционни файлове.
- Корекция на build и release скриптовете.
- Корекция на настройките на хоста на хранилището за неща като клон по подразбиране, правила за сливане и други подобни, които имат отношение към именуването на клоновете.

- Обновяване на обръщенията към стария клон в документацията.
- Затваряне или сливане на всички pull requests, насочени към стария клон.

След всико това, и след като се уверите, че main клонът функционира точно като *master* клона, можете да изтриете *master* от сървъра:

```
$ git push origin --delete master
```

Стратегии за работа с клонове код

След като вече имате основните познания за разклоняване и сливане на код, нека видим какво можете и следва да правите с тях. Тук ще разгледаме някои стандартни стратегии за работа с клонове код, които стават възможни благодарение на лекотата, с която Git управлява разклоненията. Може да решите да използвате някои от тях във вашия цикъл на разработки.

Дълго използвани клонове

Понеже Git използва просто трипосочно сливане, сливането на един клон с друг много пъти в продължение на дълъг период от време, е много лесно. Това означава, че можете да имате множество клонове, които са винаги отворени и които можете да ползвате за различни етапи на вашия цикъл от разработки и редовно да правите сливания на промените от един клон в друг.

Много разработчици, които ползват Git, следват подобна тактика - в `master` клона държат само напълно стабилния код, който е вече публикуван или ще бъде публикуван. Те имат отделен паралелен клон наречен `develop` или `next`, от който работят или тестват за стабилност — този код не е непременно стабилен, но щом стане такъв, може да се слее в `master` клона. Той се използва за създаването на допълнителни topic branches (клонове с кратък живот, подобни на вашия `iss53`) когато са готови, за да е сигурно, че те преминават всички тестове и не предизвикват грешки.

В действителност, говорим за указатели, които се преместват по линията на къмитите, които правим. Стабилните клонове са в долния край на линията на историята на къмитите, а най-новите - в горния край.

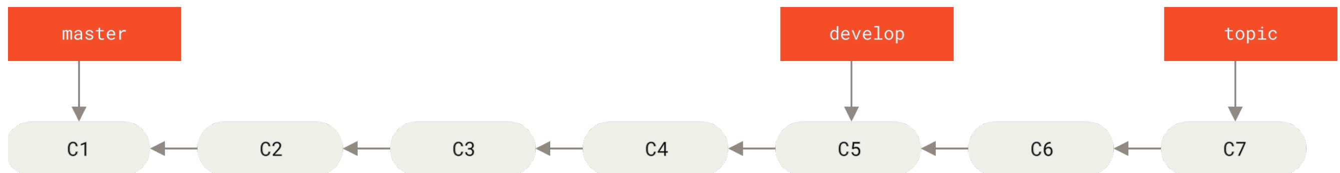


Figure 26. Линеен изглед на progressive-stability разклоняване

За улеснение, можем да мислим за тях като за работни помещения, където множествата къмити преминават към "по-стабилно" помещение, когато се изтестват напълно.

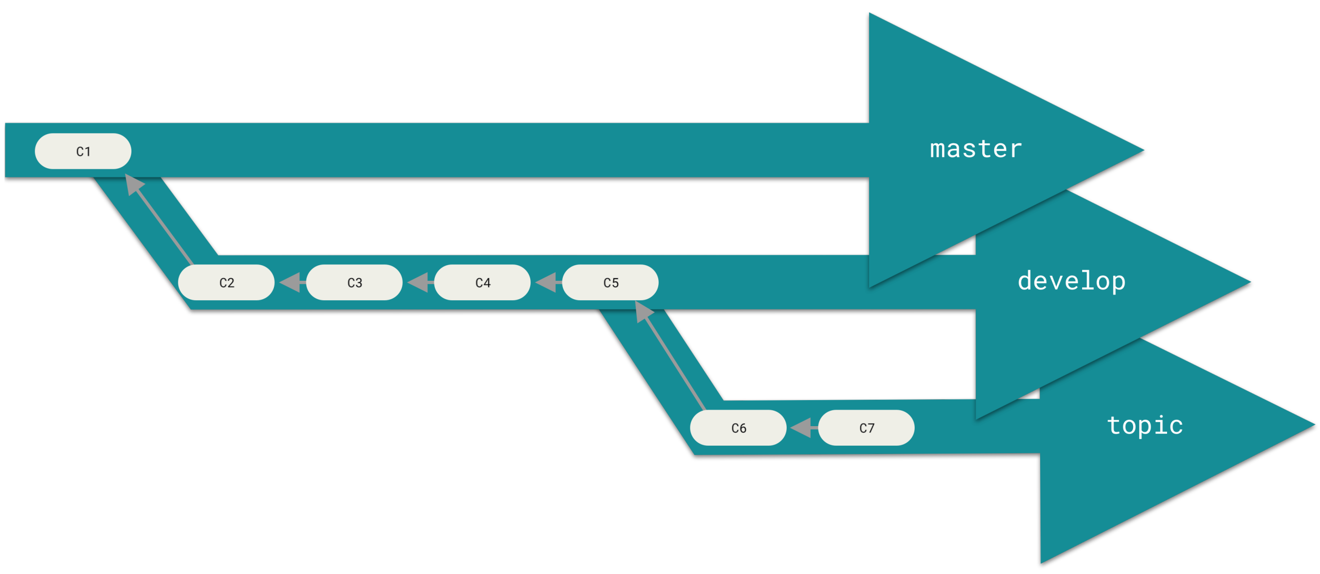


Figure 27. Дървовиден изглед на progressive-stability разклоняване

Можете да правите това за различни нива на стабилност. Някои по-големи проекти също така имат `proposed` или `pu` (proposed updates) клон интегриращ други такива, които може да не са готови да отидат в `next` или `master` клона. Идеята е, че клоновете код са с различни нива на стабилност и когато достигнат по-стабилен статус - се сливат с клона над тях. Да припомним отново - поддържането на long-running клонове не е необходимо, но често е полезно, особено когато си имате работа с много големи и сложни проекти.

Topic клонове

Topic клоновете обаче, са полезни за проекти от всякакъв мащаб. Topic клонът е клон с кратък живот, който създавате за въвеждането на единична функционалност или свързана с нея работа. Това е нещо, което най-вероятно никога не сте правили с други VCS преди Git, защото в общи линии е скъп процес с тях. Но в Git е много лесно да създавате, работите, сливате и изтривате клонове по няколко пъти на ден.

Видяхте това в последната секция с клоновете `iss53` и `hotfix`. Направихте няколко къмита в тях и ги изтрихте директно след като ги сляхте с `master` клона. Тази техника позволява да превключвате към различни контексти в проекта ви бързо и изцяло — понеже работата ви е изолирана в собствени клонове фокусирани в специфични задачи е по-лесно да преглеждате какво е ставало по време на специфичната разработка по даден проблем на проекта. Можете да си пазите промените там за минути, дни или месеци и да ги слеете, когато сте готови — без значение на последователността, в която са създавани или променяни.

Представете си пример - работите по проект в основния клон, създавате клон за решаване на възникнал проблем (`iss91`), работите в него известно време, създавате още един клон за да изпробвате втори начин за решаване на възникналия проблем (`iss91v2`), връщате се отново в `master` клона и работите по него дълго време. В един момент ви хрумва идея, за която не сте сигурни, че е толкова добра, но за да я изпробвате, създавате клон за нея (да го наречем `dumbidea`). Историята на къмитите ви би могла да изглежда така:

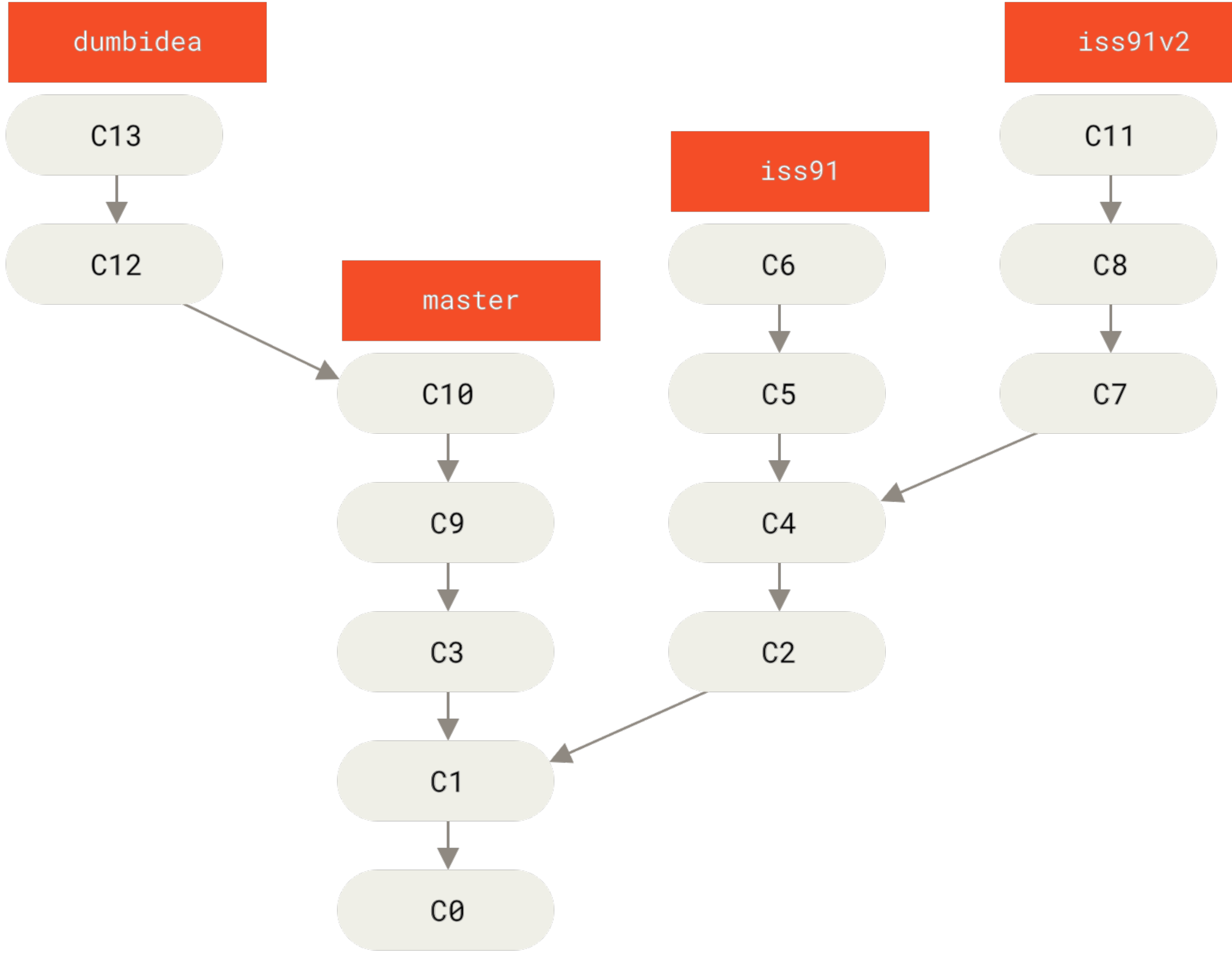


Figure 28. Множество topic клонове

Сега нека кажем, че решавате да изберете второто решение на проблема като най-добро (`iss91v2`) и същевременно сте показали `dumbidea` клона на вашите колеги и той се е оказал гениално хрумване. Можете да захвърлите клона `iss91` (губейки къмитите `C5` и `C6`) и да слеете в `master` останалите два. Тогава историята ще изглежда така:

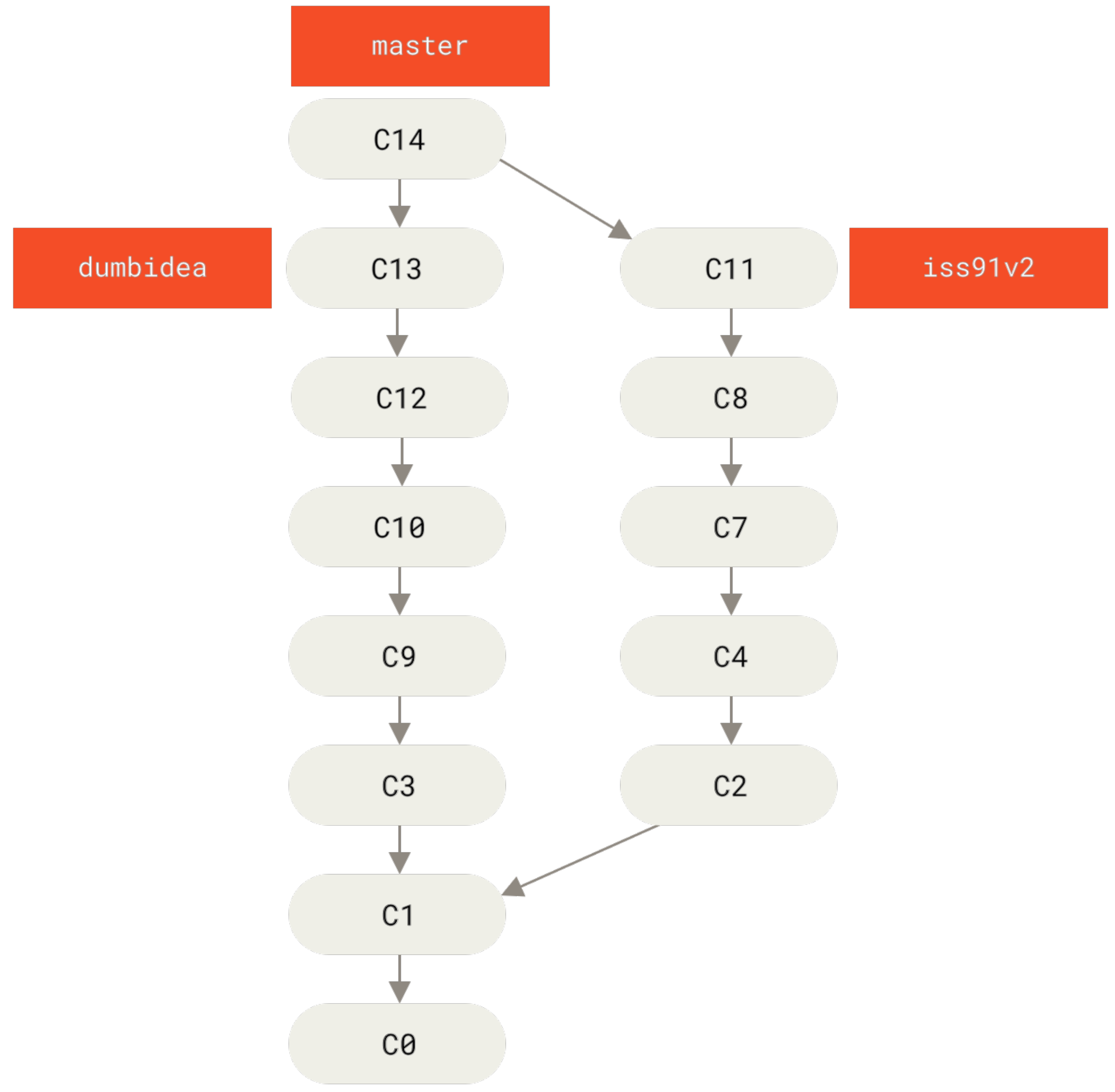


Figure 29. Историята след сливането на `dumbidea` *и* `iss91v2`

Ще обърнем повече внимание на различните работни стратегии в главата Git в разпределена среда, така че преди да се спрете на конкретна схема за разклоняване за следващия ви проект, уверете се че сте я прочели.

Важно е просто да помните, че когато правите всичко това - тези клонове са изцяло локални. Когато разклонявате и сливате, всичко се прави единствено във вашето Git хранилище - не се извършва никаква мрежова комуникация.

Отдалечени клонове

Отдалечените референции са указатели към вашите отдалечени хранилища, вкл. клонове, тагове и др. Можете да получите списък на всички отдалечени указатели изрично с командата `git ls-remote <remote>`, или `git remote show <remote>` за отдалечени клонове и друга информация. Най-използваната функционалност от разработчиците е да се възползват от

предимствата на remote-tracking клоновете.

Remote-tracking клоновете са указатели към състоянието на отдалечените клонове код. Това са локални референции, които не можете да местите, те се преместват автоматично за вас в резултат от някакви мрежови комуникации, така че да е сигурно, че те акуратно отразяват статуса на отдалеченото хранилище. Служат като отметки за да ви напомнят в какво състояние са били отдалечените клонове код последния път, когато сте се свързвали с тях.

Те следват конвенцията `<remote>/<branch>`. Например, ако искате да видите в какво състояние е бил `master` клона на вашето отдалечено хранилище `origin` последния път, когато сте комуникирали с него, можете да го направите посредством клона `origin/master`. Ако съвместно с ваш колега работите по даден проблем и той е публикувал нещо в клона `iss53`, вие можете да имате локален такъв със същото име, но клонът на сървъра ще е достъпен през референцията `origin/iss53`.

Това може да е объркващо, така че нека го разгледаме с пример. Нека кажем, че имате Git сървър в мрежата ви на адрес `git.ourcompany.com`. Ако клонирате хранилище от него, `clone` командата на Git автоматично ще го именува с името `origin`, ще издърпва всички данни от него, ще създаде указател към мястото където е `master` клона на това хранилище и ще го съхрани като `origin/master` локално при вас. Git също така създава локален `master` клон, който сочи към същото място в проекта, така че да имате от къде да започнете локалната си работа.

“origin” не е специална дума

Точно както и `master` - думата `origin` няма никакво специално значение в Git. Докато “master” е името по подразбиране за началния клон когато изпълните `git init`, то “origin” се ползва по подразбиране, когато изпълнявате `git clone`. Това е единствената причина за толкова разпространеното ползване на тези две думи. Ако изпълните `git clone -o booyah` вместо просто `git clone`, тогава ще имате `booyah/master` като ваш отдалечен клон по подразбиране.

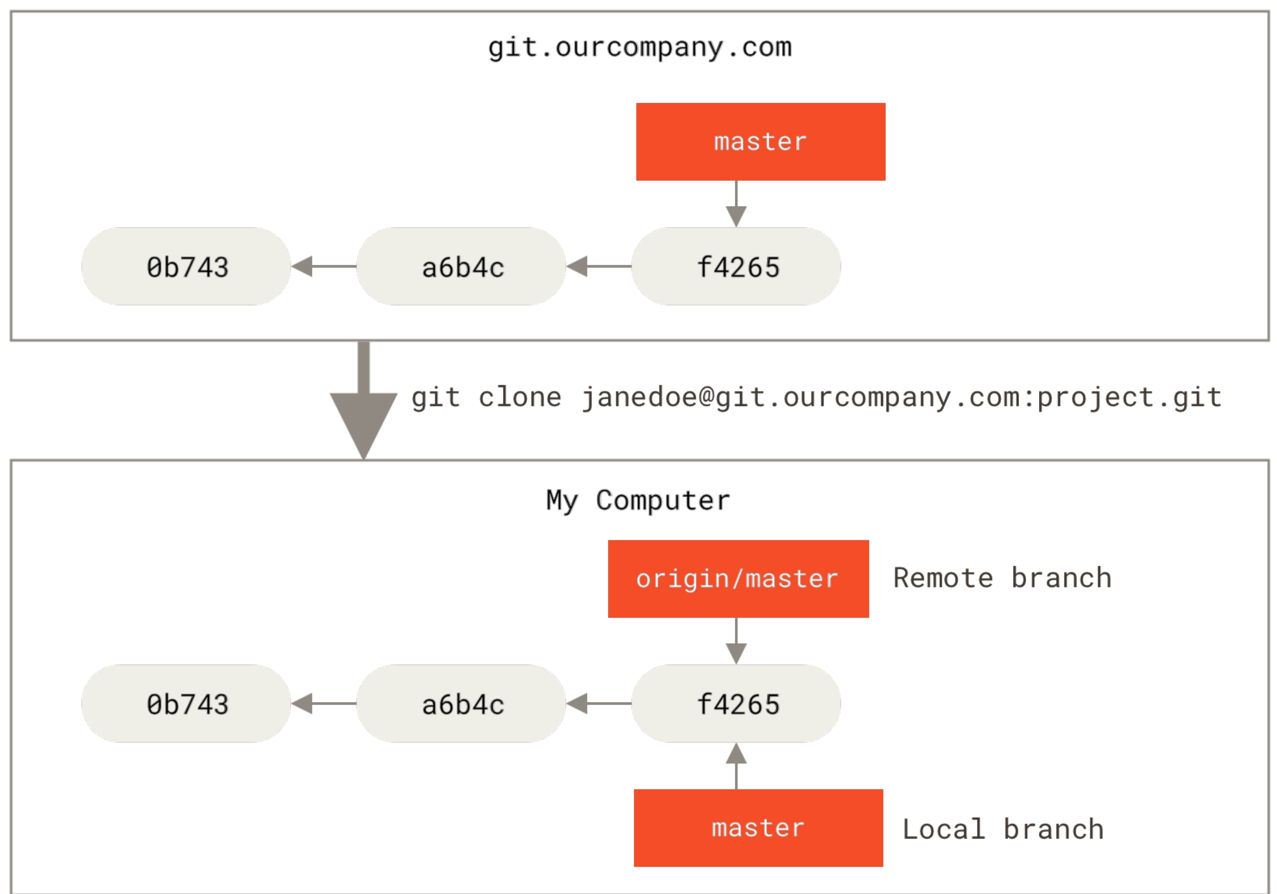
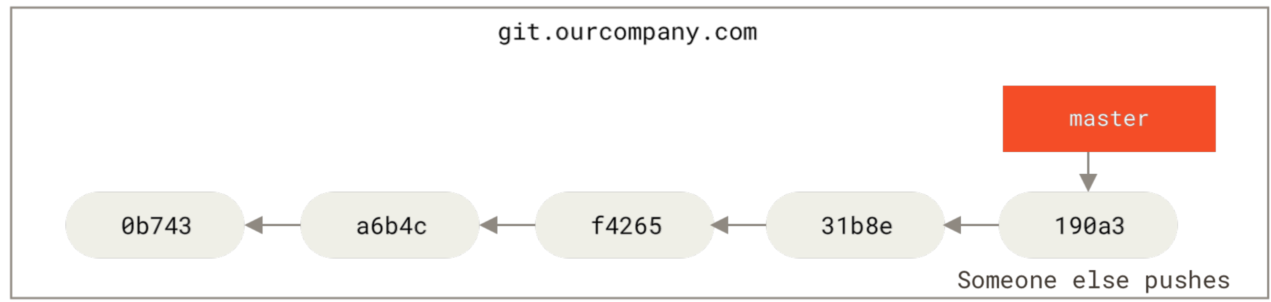


Figure 30. Сървърното и локалното хранилище след клониране

Ако вие вършите някаква работа в локалния си `master` клон и междувременно някой друг изпрати нещо към `git.ourcompany.com` и промени `master` клона там, тогава вашите истории на промените ще се движат напред по различни начини. Също така, докато не контакувате с вашия `origin` сървър, указателят `origin/master` не се премества самичък.



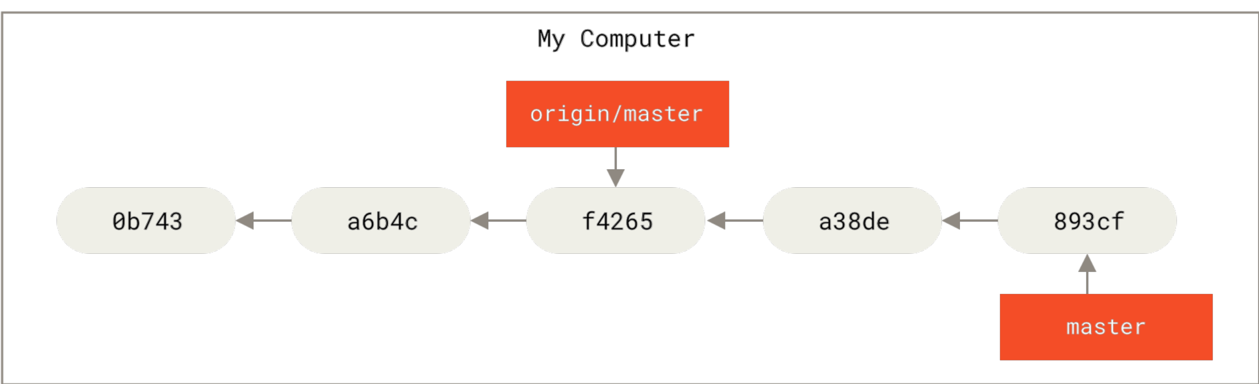


Figure 31. Локалната и отдалечената работа могат да се разделят

За да синхронизирате работата си така, че да отразява промените на сървъра, използвайте командата `git fetch <remote>` (в нашия случай `git fetch origin`). Тази команда установява кой е адреса на сървъра "origin" (в случая `git.ourcompany.com`), издърпва всички данни, които все още нямате локално и обновява локалната база данни, така че указателят `origin/master` вече да сочи към нова, по-актуална позиция от историята.

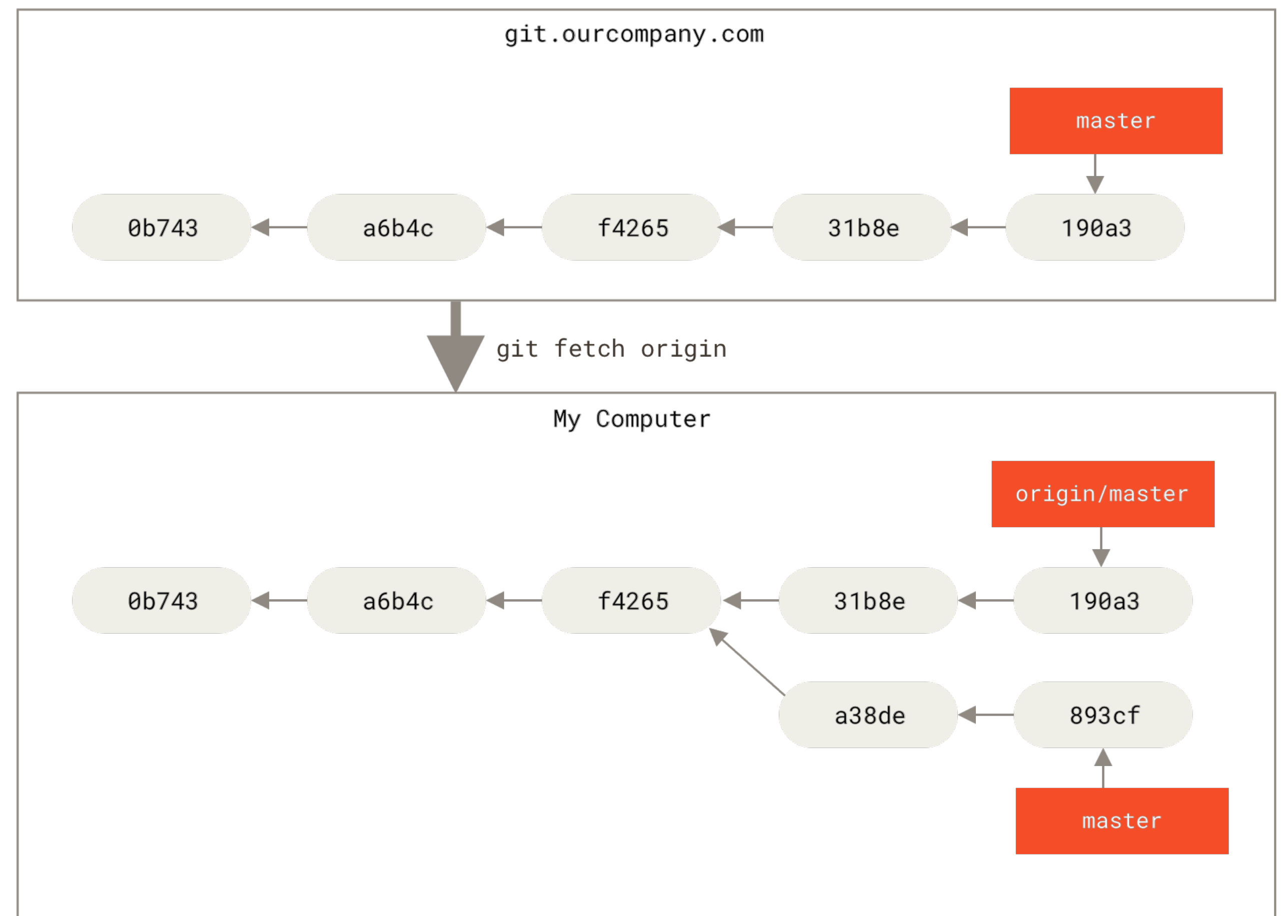
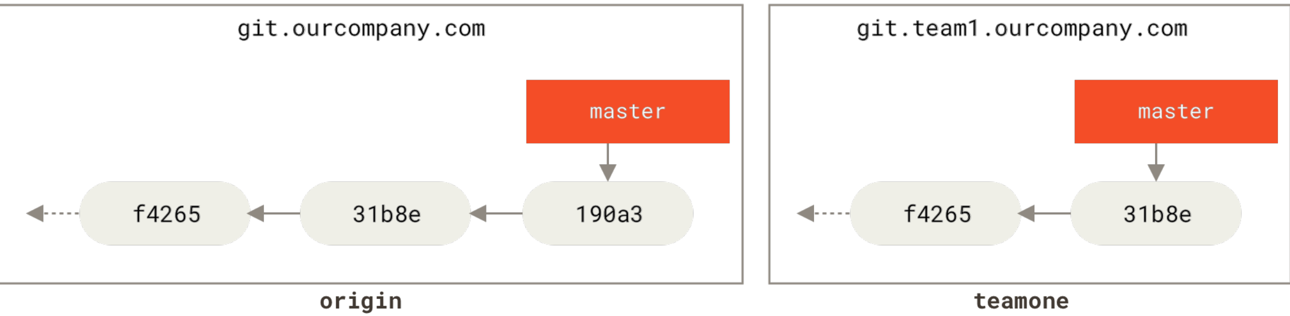


Figure 32. `git fetch` обновява отдалечените ви референции

За да демонстрираме какво е да имате много отдалечени сървъри и как изглеждат клоновете на съответните отдалечени проекти, нека приемем, че имате още един вътрешнофирмен Git сървър, който се използва само за разработка от някой от вашите sprint екипи. Сървърът е на адрес `git.team1.ourcompany.com`. Можете да го добавите като нова отдалечена референция към проекта, в който работите, с командата `git remote add`, която разгледахме в Основи на Git. Наречете това отдалечено хранилище `teamone`, което ще е краткото име за целия URL.



git remote add teamone git://git.team1.ourcompany.com

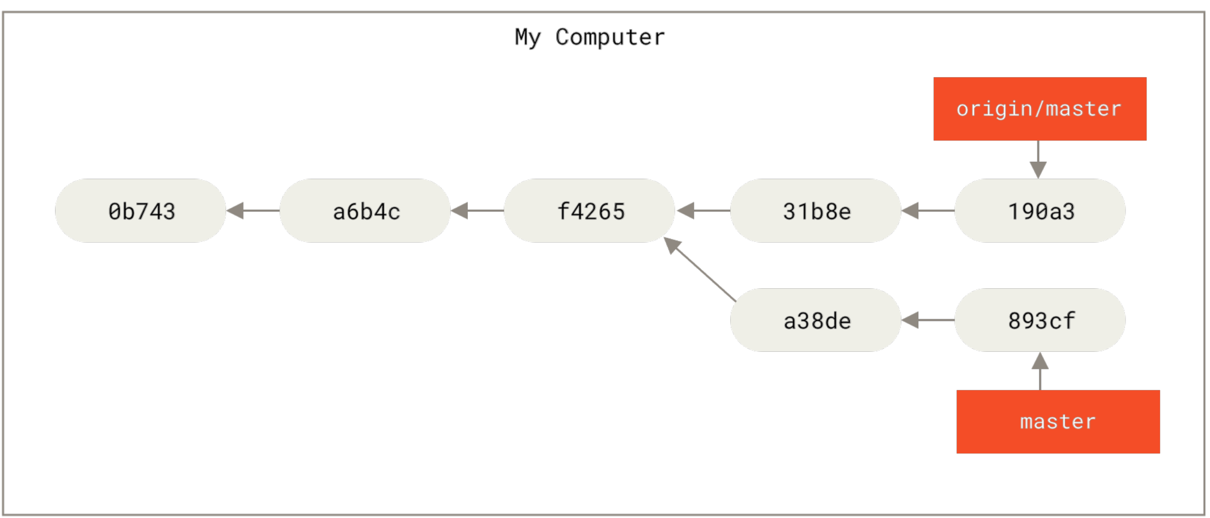


Figure 33. Добавяне на нов remote

Сега можете да изпълните `git fetch teamone` за да изтеглите всичко, което този сървър има, а вие все още нямате локално. Понеже този сървър съхранява подмножество от данните на вашия `origin` сървър, Git всъщност не тегли от него нищо, а просто създава remote-tracking клон наречен `teamone/master`, който сочи към последния къмит, който `teamone` има за своя `master` клон.

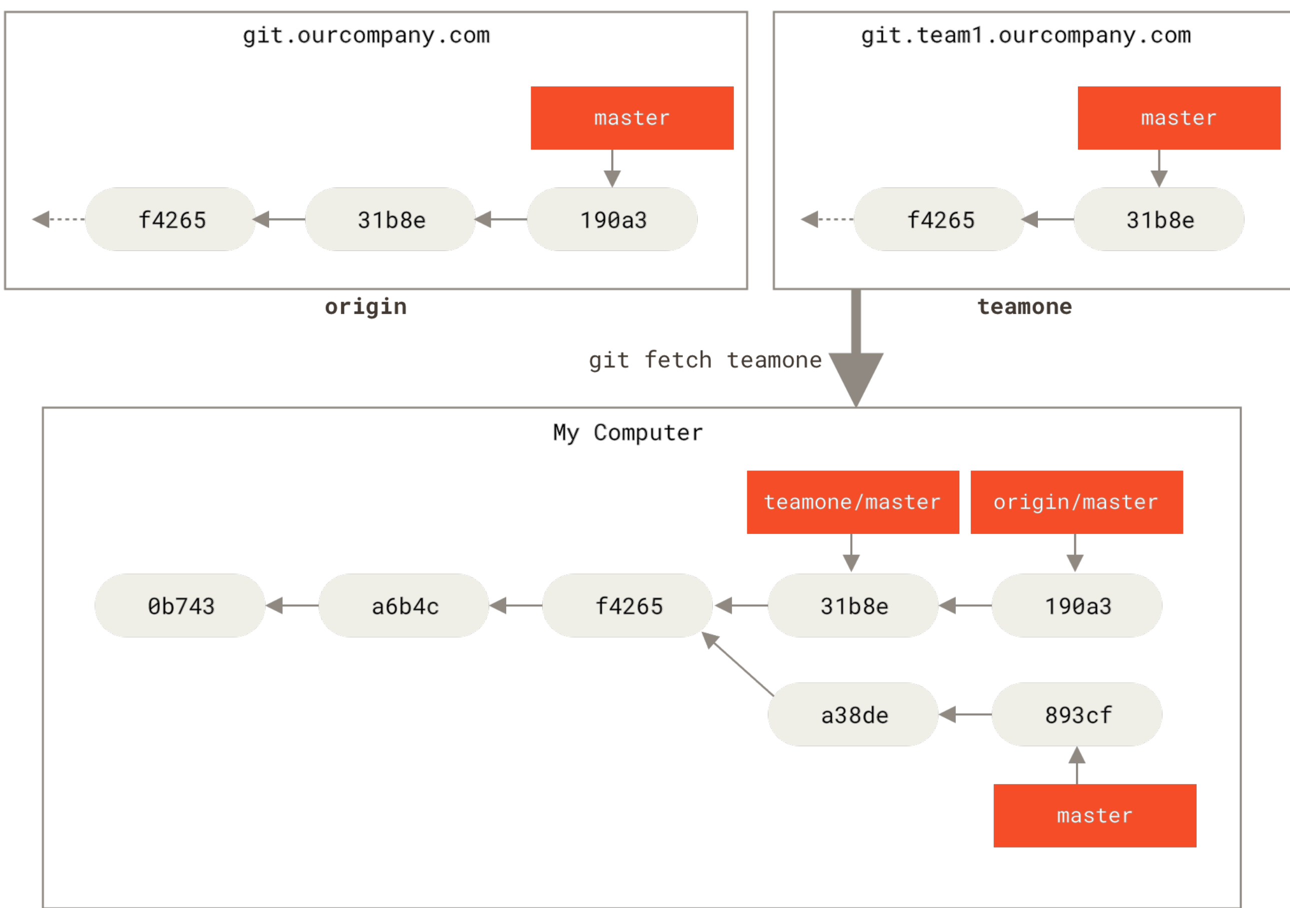


Figure 34. Remote tracking клон за `teamone/master`

Изпращане към сървъра (pushing)

Когато искате да споделите работата от ваш локален клон с другите си колеги, трябва да го изпратите към отдалечено хранилище, към което имате права за писане. Вашите локални клонове не се синхронизират автоматично с регистрираните отдалечени хранилища — вие трябва изрично да ги изпратите към тях. По този начин, можете да си имате частни клонове код само за вас и които не желаете да споделяте, а да споделяте само topic клоновете, към които допринасяте.

Ако имате клон наречен `serverfix`, който искате да споделите с другите, можете да го изпратите по същия начин, по който изпратихте първия си клон. Изпълнете `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
```

Това е един вид съкращение. Git автоматично разширява името на клона `serverfix` до

refs/heads/serverfix:refs/heads/serverfix, което означава, “Вземи локалния ми клон serverfix и го изпрати към отдалеченото хранилище, обновявайки отдалечения клон serverfix”. Ще разгледаме частта refs/heads/ в подробности в Git на ниско ниво, засега не ѝ обръщайте внимание. Можете също да изпълните git push origin serverfix:serverfix, което върши същата работа, тази команда означава “Вземи локалния ми клон serverfix и го слей с отдалечения със същото име”. Можете да използвате този формат, за да слеете локален клон с отдалечен, който се казва по различен начин. Ако не желаете отдалеченият да се казва serverfix в сървърното хранилище, можете да изпълните например git push origin serverfix:awesomebranch и отдалечения клон ще се казва awesomebranch.

Не пишете паролата си всеки път

Ако използвате HTTPS URL за изпращане, Git сървърът ще ви пита за име и парола всеки път когато изпращате към него. По подразбиране ще получите запитване в терминала, така че сървърът да знае дали можете да записвате на него

Ако не желаете това, можете да направите т. нар. “credential cache”. Най-лесно е да запомните паролата в кеша за известно време, което можете да направите с командата git config --global credential.helper cache.

За повече информация за различните опции за този кеш, вижте Credential Storage система.

Следващият път, когато колегите ви теглят от сървъра, ще получат референция към точката в която сочи сървърната версия на клона serverfix под формата на отдалечен за тях клон 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
```

Важно е да се запомни, че когато изпълните git fetch за да изтеглите новосъздадени remote-tracking клонове, вие не получавате автоматично техни редактируеми копия! С други думи, в този случай няма да имате действителното съдържание на клона serverfix, а само указателят origin/serverfix, който не можете да променяте.

За да стане това, трябва да го слеете в текущия си клон с git merge origin/serverfix. Ако не желаете това да стане в текущия клон, а в нов локален такъв с име serverfix, можете да го базирате на remote-tracking клона:

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

Това ви дава локален клон, в който да работите и който стартира от точката, в която е `origin/serverfix`.

Проследяване на клонове

Проследяващите клонове са локални такива, които имат директна връзка с отдалечен клон. Ако сте в такъв проследяващ клон и изпълните `git pull`, Git автоматично знае кой сървър да ползва за изтегляне и в кой клон да слее разликите.

Когато клонирате хранилище, системата създава автоматично `master` клон, който проследява `origin/master`. Обаче, ако желаете, можете да създадете и други проследяващи клонове - такива, които следят клонове от други отдалечени хранилища или пък не следят точно `master` клона от сървъра. Прост случай е примерът, който току що видяхте, изпълнявайки `git checkout -b <branch> <remote>/<branch>`. Това е често случваща се операция, за която Git осигурява `--track` съкращение:

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

В действителност, това е толкова често срещано, че съществува съкращение на съкратената версия. Ако клонът с името, към който се опитвате да превключите, не съществува локално, но съвпада по име с такъв от точно едно отдалечено хранилище, Git ще създаде проследяващ клон за вас:

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

Ако искате да създадете клон с различно име от това, което е в отдалеченото хранилище, можете лесно да ползвате първата версия с различно име за локалния клон:

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

Сега, локалният ви клон `sf` автоматично ще тегли от `origin/serverfix`.

Ако ли пък имате локален клон и искате да го накарате да следи клона, който току що изтеглихте, или пък искате да смените upstream клона, можете да използвате `-u` или `--set-upstream-to` опциите на `git branch` за да укажете изрично името.

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

Upstream съкращение

Когато имате настроен проследяващ клон, можете да се обръщате към неговия upstream клон със съкращенията `@{upstream}` или `@{u}`. Така, ако сте на `master` клона и той следи `origin/master`, можете да използвате `git merge @{u}` вместо `git merge origin/master`, ако ви е по-удобно.

За да отпечатате какви проследявани клонове имате, използвайте параметъра `-vv` на `git branch`. Това ще изведе списък с локалните клонове с повече информация, вкл. какво следи всеки клон и дали той е напред/назад или и двете в сравнение с локалната версия.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
  master    1ae2a45 [origin/master] Deploy index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it
  testing   5ea463a Try something new
```

Така можем да видим, че нашият `iss53` локален клон проследява `origin/iss53` и че е напред с две, което значи, че локално имаме два къмита, които не са изпратени към сървъра. Можем също да видим, че `master` клона следи `origin/master` и е актуален. След това виждаме, че `serverfix` следи `server-fix-good` клона от отдалеченото хранилище `teamone` и е напред с три и назад с едно, което значи, че на сървъра има един къмит, който още не сме слели локално и три локални къмита, които не са пратени на сървъра. Накрая, виждаме че локалния ни клон `testing` не следи нито един отдалечен клон.

Важно е да запомним, че тези цифри са актуални към момента на последното изпълнение на `git fetch` за всеки сървър. Командата не се свързва със сървърите, а само ни казва какво е кеширала локално от последната комуникация с всеки от тях. Ако желаете актуални ahead/behind статистики, преди нея трябва да обновите кеша с отдалечения статус:

```
$ git fetch --all; git branch -vv
```

Pulling (изтегляне и сливане)

Вече казахме, че командата `git fetch` изтегля от сървъра всички промени, които все още нямате локално, но не променя нищо в работната директория. Тя просто ще изтегли данните и ще ви позволи да ги слеете сами. Обаче, съществува команда `git pull`, която по същество е `git fetch` последвана от автоматично изпълнение на `git merge` в повечето случаи. Ако имате проследяващ клон създаден по начина по-горе, изрично посочен от вас или автоматично създаден като следствие от командите `clone` или `checkout`, то `git pull` последователно ще потърси кое отдалечено хранилище и клон са следени от текущия локален клон, след това ще изтегли информацията от тях и ще се опита да я слее автоматично в локалния клон.

В общия случай е по-добре да използвате `fetch` и `merge`, защото понякога магията на `git pull` може да създаде недоразумения.

Изтриване на отдалечени клонове

Да допуснем, че сте готови с отдалечения клон - да кажем че колегите ви са свършили работа по определена функционалност и са я слели в отдалечения ви `master` клон (или както и да се казва клона, в който е стабилния код). Можете да изтриете отдалечения клон с параметъра `--delete` на `git push`. Ако искате да изтриете `serverfix` клона от сървъра, изпълнете:

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

Това в общи линии изтрива указателя от сървъра. Git сървърът в повечето случаи ще пази данните за известно време докато мине garbage collection системата му, така че случайно изтритите данни често могат лесно да се възстановят.

Пребазиране на клонове

В Git съществуват два основни начина за интегриране на промени от един клон код в друг: сливане (`merge`) и пребазиране (`rebase`). В тази секция ще научите какво е пребазирането, как да го правите, защо е мощен инструмент и кои са случаите, в които не бихте искали да го използвате.

Просто пребазиране

Ако се върнете назад до по-ранния пример в Сливане, можете да си припомните, че работата ви се разклонява и вие правихте къмити в два различни клона.

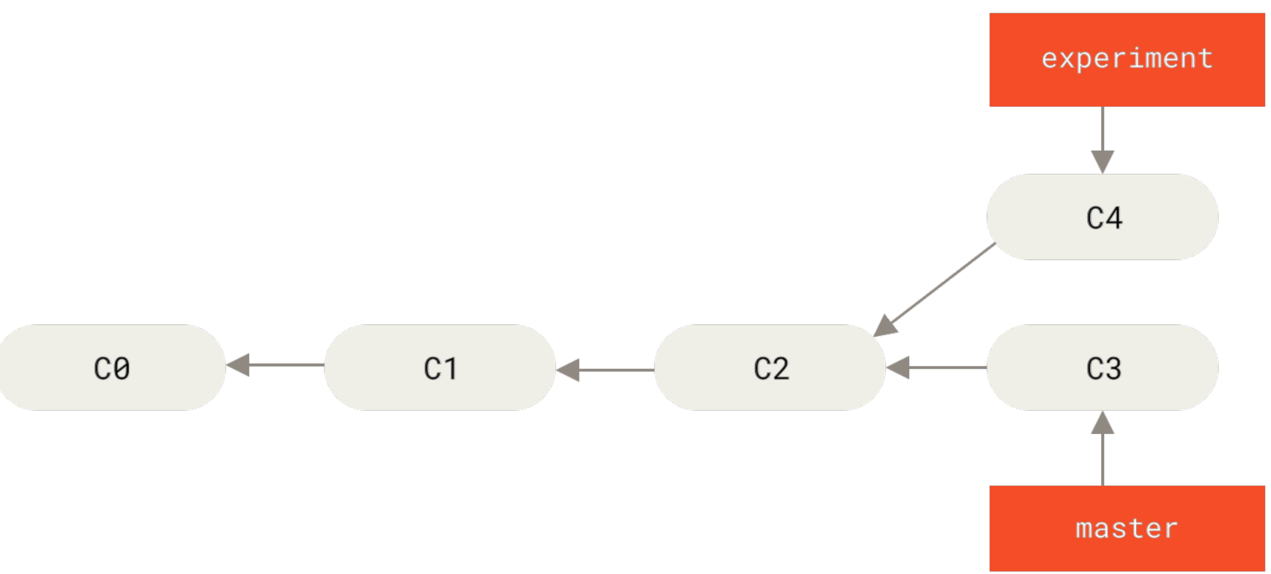


Figure 35. Проста история на разклоняването

Най-лесният начин за интегрирането на клоновете, както вече разгледахме, беше командата `merge`. Тя изпълнява трипосочно сливане между най-новите snapshot-и от клоновете (`C3` и `C4`) и най-близкия им общ предшественик (`C2`) създавайки нов snapshot (и

къмит).

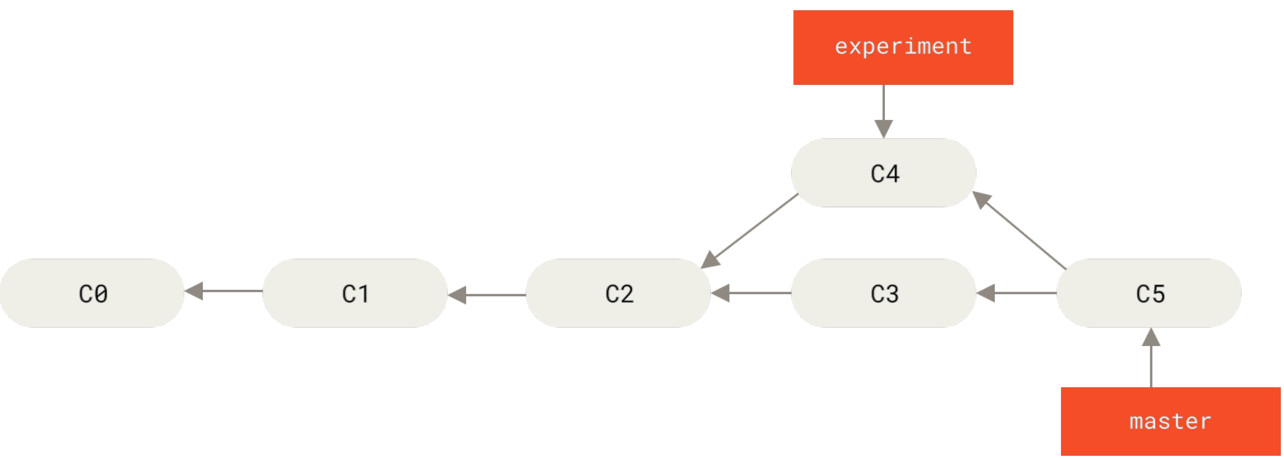


Figure 36. Сливане за интегриране на разклонена работна история

Обаче, съществува и друг начин да направите това: можете да вземете patch на промените, които са въведени с `C4` и да ги приложите върху `C3`. В Git това се нарича пребазиране, *rebasing*. С командата `rebase`, вие вземате всички промени къмитнати в един клон и ги пускате в друг такъв.

В този пример, ще извлечете клона `experiment` и ще го пребазирате върху `master` по следния начин:

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

Това става, като се намери най-близкия общ предшестващ къмит на двата клона (този върху, който сте в момента и този, който ще пребазирате), вземат се разликите въведени от всеки къмит на клона, върху който сте, разликите се записват във временни файлове, текущият клон се ресетва към същия къмит, в който е клона, който ще се пребазира, и накрая се прилага всяка промяна поред.

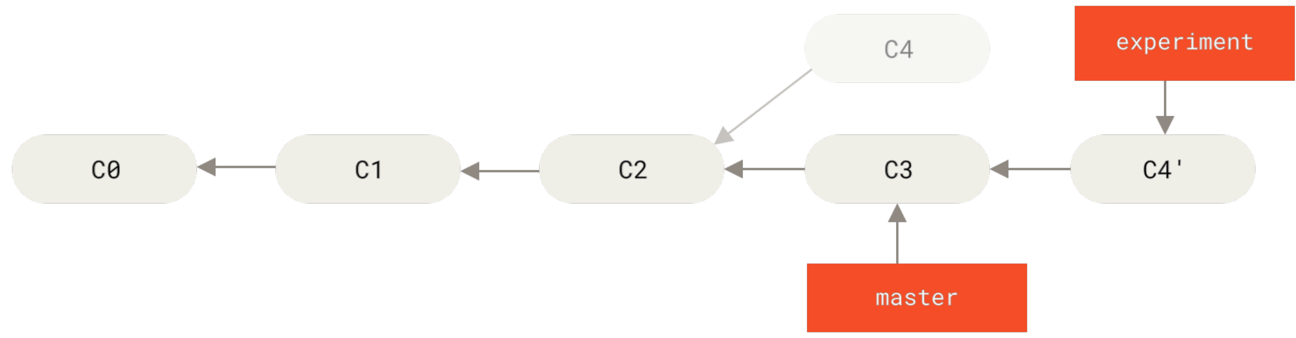


Figure 37. Пребазиране на промяната от `C4` в `C3`

В този момент, можете да се върнете към `master` клона и да направите fast-forward сливане.

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

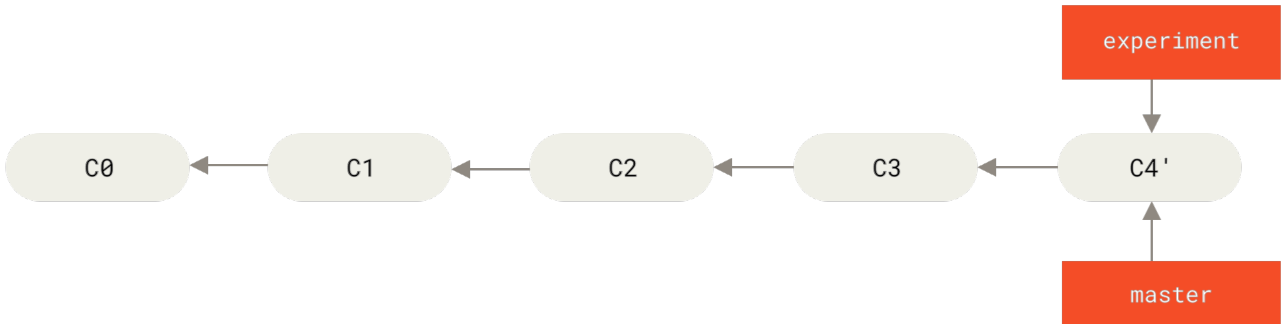


Figure 38. Fast-forwarding на клона `master`

Сега snapshot-ът, към който сочи `C4'` е точно същият като този, към който сочеше `C5` в the merge example. Няма разлика в крайния резултат от интеграцията, но пребазирането прави историята по-чиста. Ако изследвате лога на един пребазиран клон код, той ще прилича на линейна история, цялата работа изглежда като случила се на серии, дори когато в действителност е била паралелна.

Често ще правите това за да се уверите, че вашите къмити се прилагат безпроблемно върху отдалечен клон — вероятно проект, към който се опитвате да допринесете, но който не поддържате като автор. В този случай, вие вършите своята дейност в собствен клон и след това пребазирате работата си върху `origin/master`, когато сте готови да изпратите своите поправки в основния проект. По този начин, поддържащият проекта разработчик не трябва да върши никаква работа по интеграцията на промените ви — просто ще направи fast-forward.

Отбележете отново, че snapshot-ът, към който сочи финалния получил се къмит (бил той последния от пребазираните къмити за rebase или пък новосъздадения в резултат от merge) е един и същи и в двата случая — разликата е само в историята. Пребазирането прилага промените от една линия на разработка в друга по реда, в който те са били направени, докато сливането взема двата края на два клона и ги слива в едно.

Други интересни пребазирания

Едно пребазиране може да бъде приложено и върху друг освен върху целевия му клон. Например, вижте фигурата История с topic клон произлизащ от друг topic клон. Създали сте един topic клон (`server`) за да добавите някаква сървърна функционалност към проекта си и сте направили къмит. След това, ползвайки този клон за отправна точка, сте създали нов такъв (`client`) за да направите някакви промени по клиентската част и сте къмитнали няколко пъти. Накрая, върнали сте се обратно към сървърния клон и сте направили още няколко къмита.

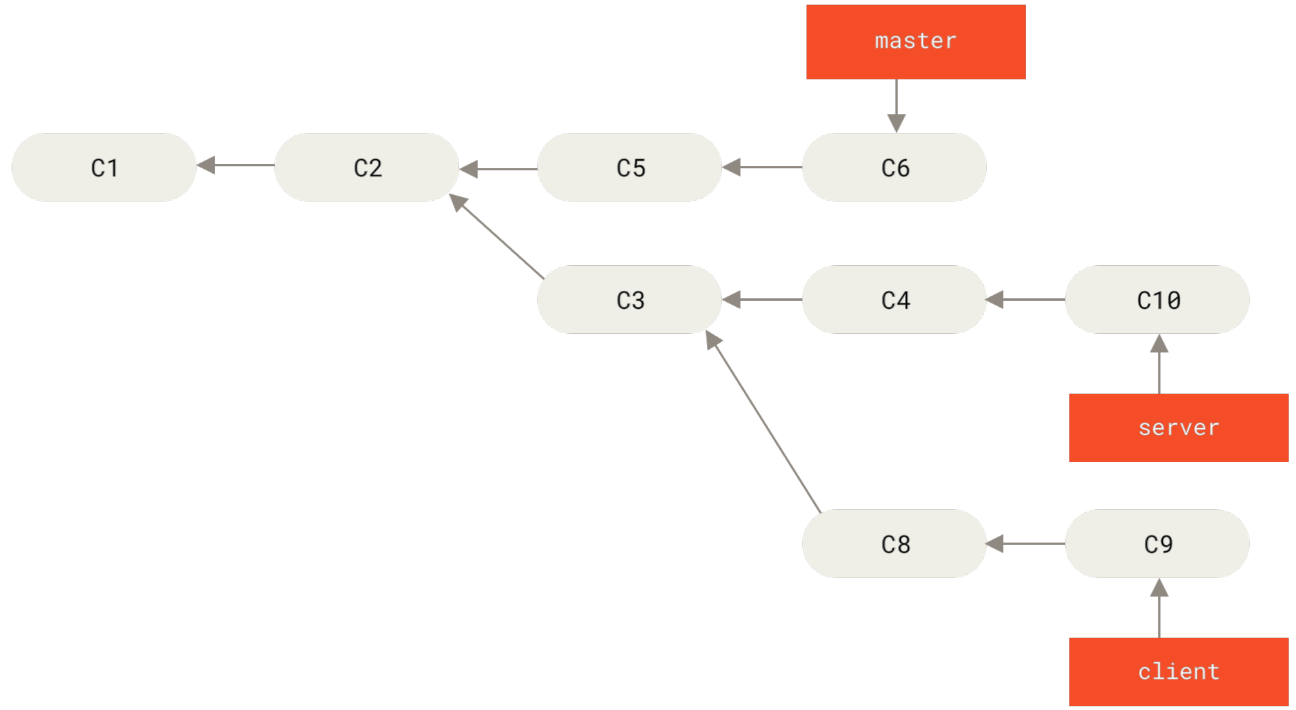


Figure 39. История с topic клон произлизащ от друг topic клон

Да кажем, че решавате да слеете клиентските промени в `master` клона за публикуване, но желаете да си запазите server-side промените за допълнително тестване. Можете да вземете промените в клиентската част, които не присъстват на сървъра (`C8` and `C9`) и да ги приложите върху `master` клона чрез параметъра `--onto` на командата `git rebase`:

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

Това звучи така, “Вземи клона `client`, разбери кои са промените в него след момента, в който този клон се е разклонил от `server` клона и ги приложи отново в него сякаш той е бил базиран първоначално на `master` клона.” Изглежда доста объркано, но резултатът е впечатляващ.

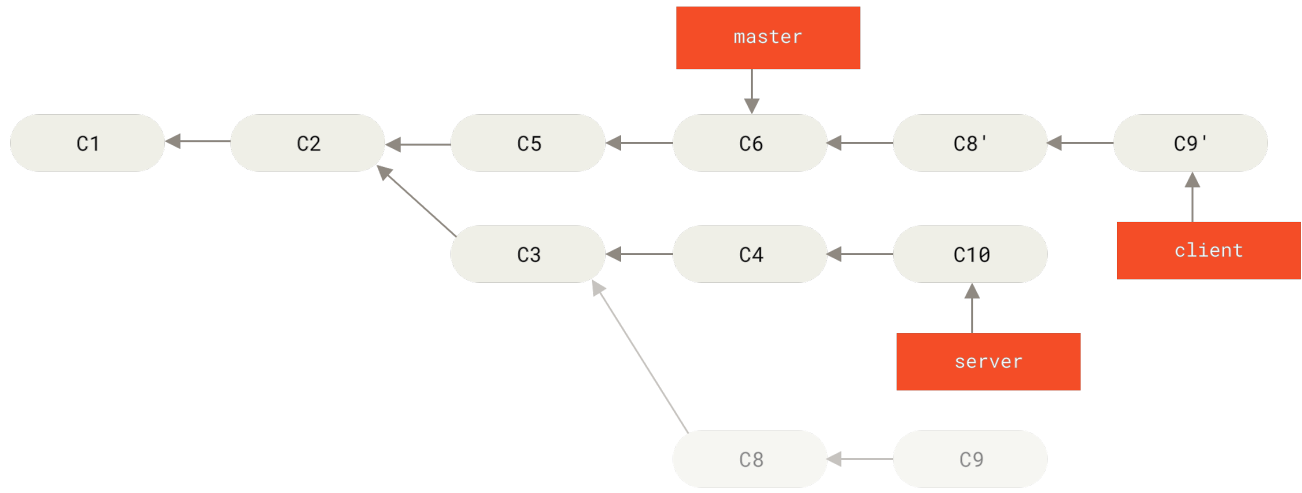


Figure 40. Пребазиране на topic клон от друг topic branch

Сега можете да направите fast-forward на `master` клона (виж Fast-forwarding на `master` клона

за включване на промените от клона client):

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

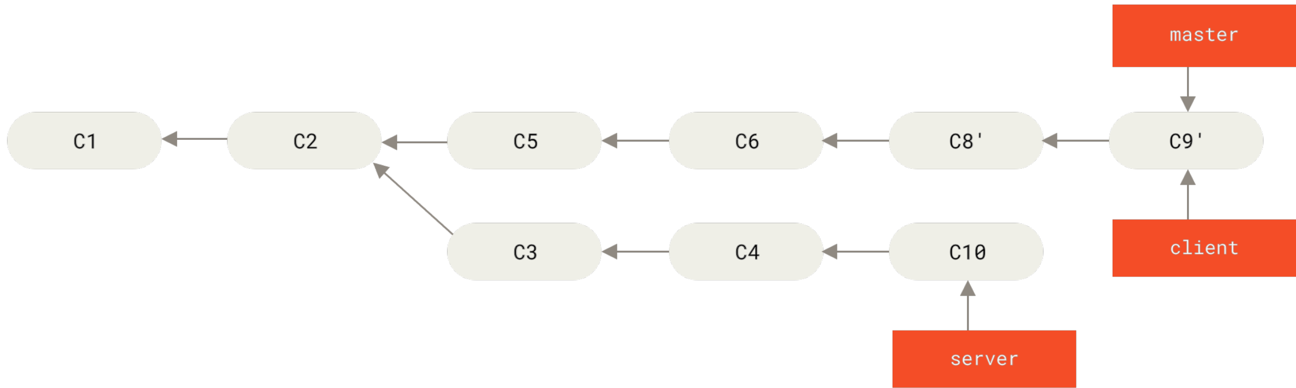


Figure 41. Fast-forwarding на `master` клона за включване на промените от клона client

Нека кажем, че решите да интегрирате и промените от server клона. Можете да пребазирате `server` клона върху `master` без да се налага да превключвате към него изпълнявайки `git rebase <basebranch> <topicbranch>` — което вместо вас ще превключи към topicbranch (в този случай `server`) и ще го приложи върху base branch (в случая `master`):

```
$ git rebase master server
```

Това прилага вашите `server` промени върху `master` клона както е показано в Пребазиране на server клона в `master` клона.

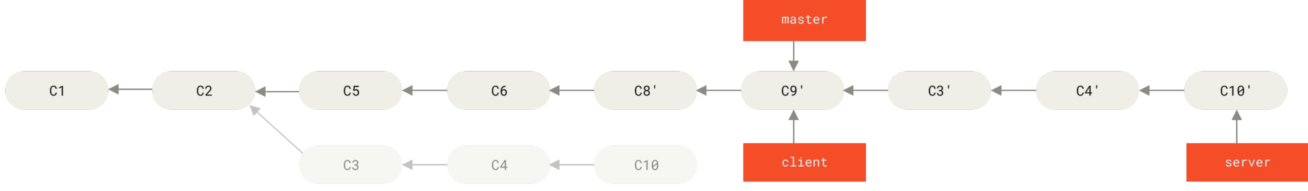


Figure 42. Пребазиране на server клона в `master` клона

След което, можете да превъртите основния клон (`master`):

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

Сега можете да изтриете клоновете `client` и `server`, защото те са интегрирани и не ви трябват повече, което ще направи историята на целия процес да изглежда така Историята на финалния къмит:

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

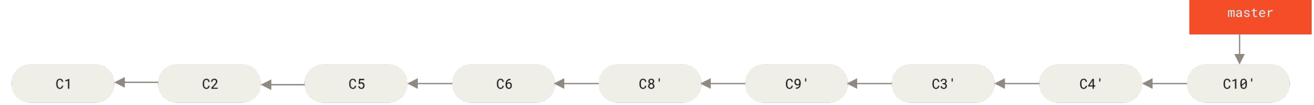


Figure 43. Историята на финалния къмит

Опасности при пребазиране

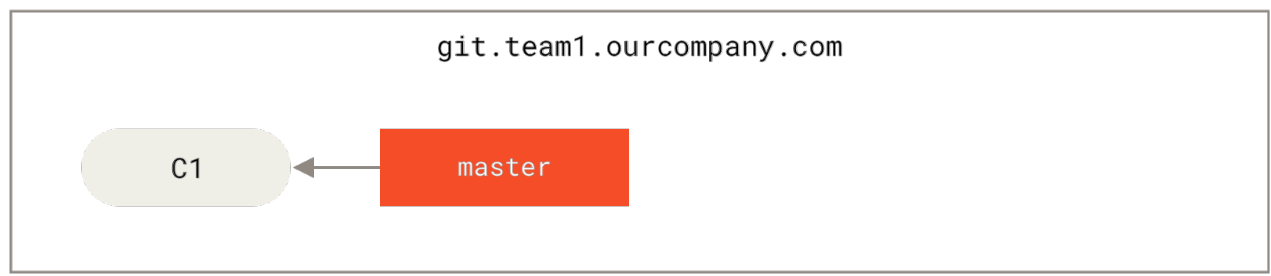
Както можете да се досетите, пребазирането си има и недостатъци, които могат да се обобщят в едно изречение:

Не пребазирайте къмити, които съществуват извън вашето собствено хранилище и върху които други хора може да са базирали работата си.

Послушате ли този съвет, ще сте ОК. Не го ли направите, ще ви намразят.

Когато пребазирате неща, вие зарязвате съществуващи къмити и създавате нови, които са подобни, но не съвсем същите. Ако изпратите къмитите някъде и други ваши колеги ги издърпат и използват като изходна точка за тяхната работа, а след това вие ги презапишете с `git rebase` и ги изпратите отново, то колегите ви ще трябва да слеят отново тяхната работа и нещата бързо ще придобият грозен вид, когато пък вие се опитате да изтеглите тяхната работа обратно за ваше ползване.

Нека видим един пример, как пребазирането на публична работа може да доведе до проблеми. Да допуснем, че клонирате от централен сървър и вършите някаква работа след това. Историята на къмитите ви изглежда по подобен начин:



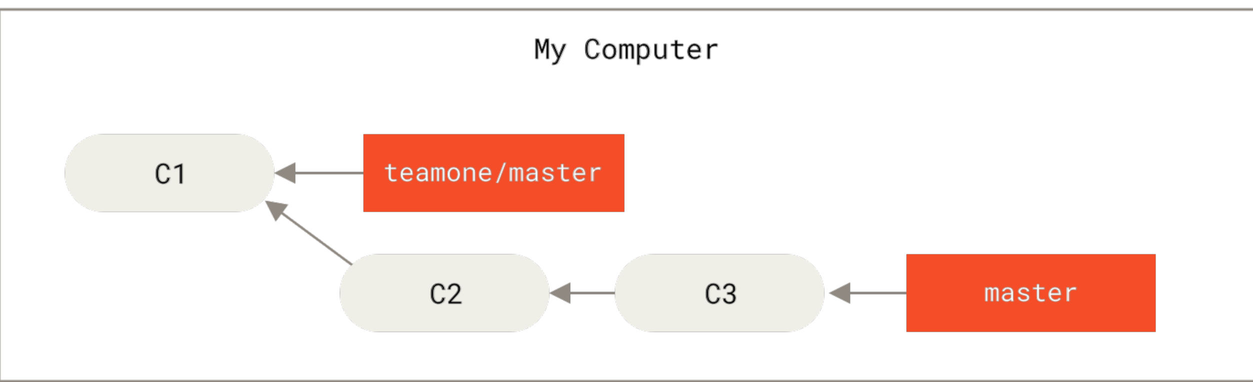
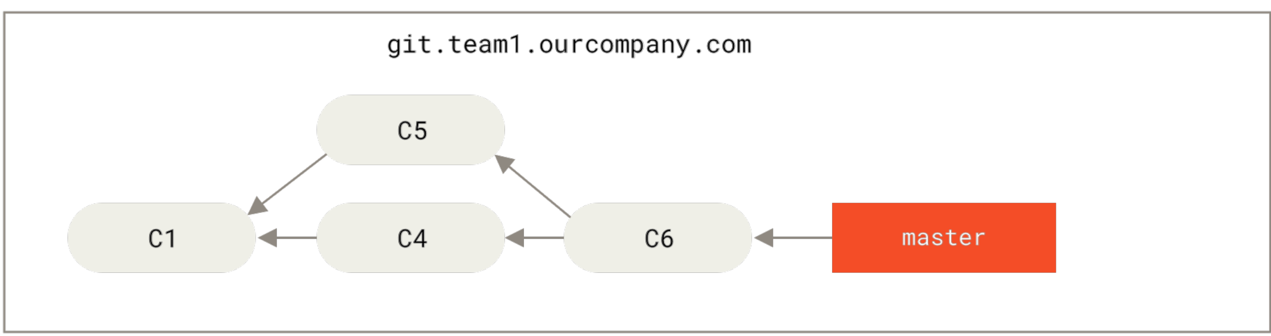


Figure 44. Клонирайте хранилище и направете промени по него

Сега, някой друг върши друга работа, която включва сливане и я изпраща към централния сървър. Вие я издърпвате и сливате новия отдалечен клон във вашата работа, така че историята придобива подобен вид:



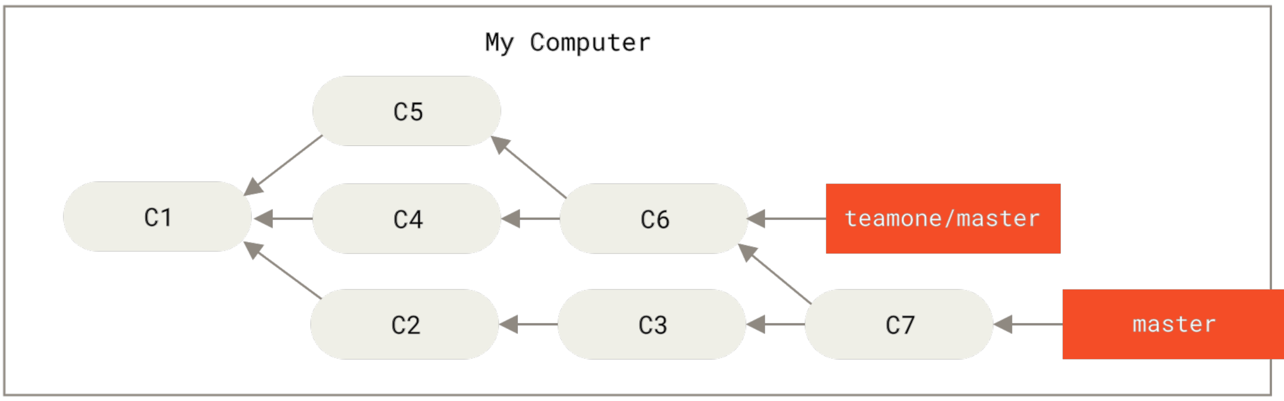
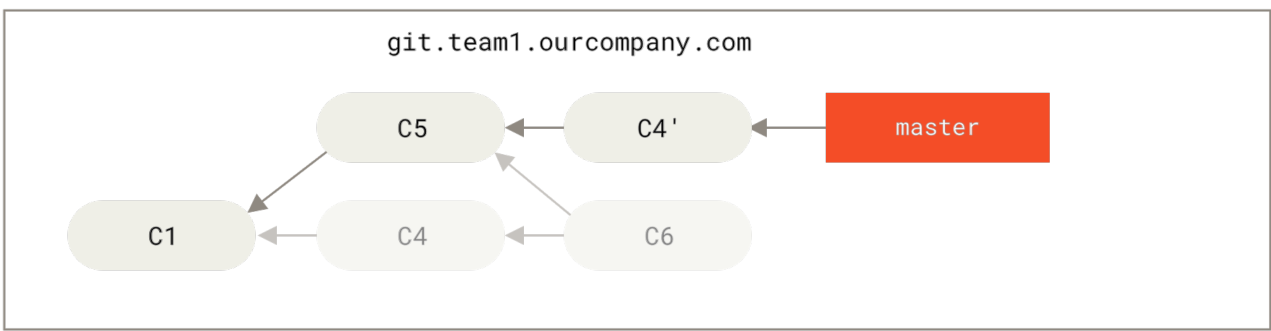


Figure 45. Издърпвате още къмити и ги сливате във вашата работа

След това човекът, който е изпратил слетите си промени, решава да се върне назад и да

пребазира работата си изпълнявайки `git push --force` за да презапише историята на сървъра. Вие от своя страна, издърпвате отново от сървъра получавайки новите къмити.



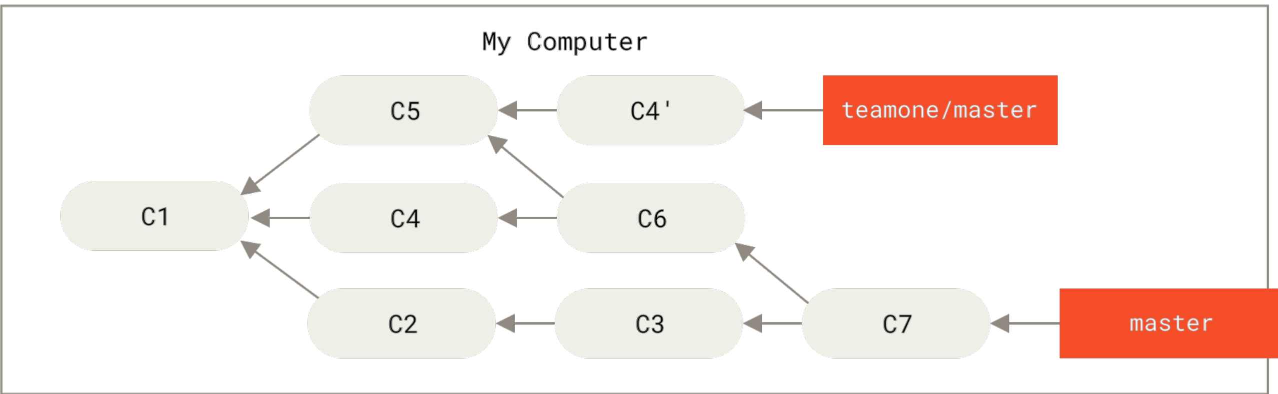


Figure 46. Някой изпраща към сървъра пребазирани къмити, изоставяйки тези, върху които вие сте базирали работата си

Сега и двамата сте в бъркотия. Ако направите `git pull`, ще се създаде merge commit, който включва и двете линни история и хранилището ви ще изглежда така:

Figure 47. Вие сливате в същата работа отново, в нов merge commit

Ако пуснете `git log` в история като тази, ще видите два къмита с един и същи автор, дата и съобщение, което ще е доста смущаващо. По-нататък, ако изпратите тази история обратно към сървъра, ще запишете допълнително в него всички тези пребазирани къмити, което би объркало допълнително хората в бъдеще. Безопасно е да предположите, че другият ви колега не желае `C4` и `C6` да присъстват в историята - в края на краищата това е причината той да направи пребазирането.

Пребазиране по време на пребазиране

Ако **се** окажете в подобна ситуация, Git разполага с допълнителни средства, които да ви помогнат. Ако някой от вашия екип форсирано изпрати промени, които презаписват нещата, които служат за база на вашата работа, трудността за вас ще е да откриете кое е ваше и какво е било презаписано.

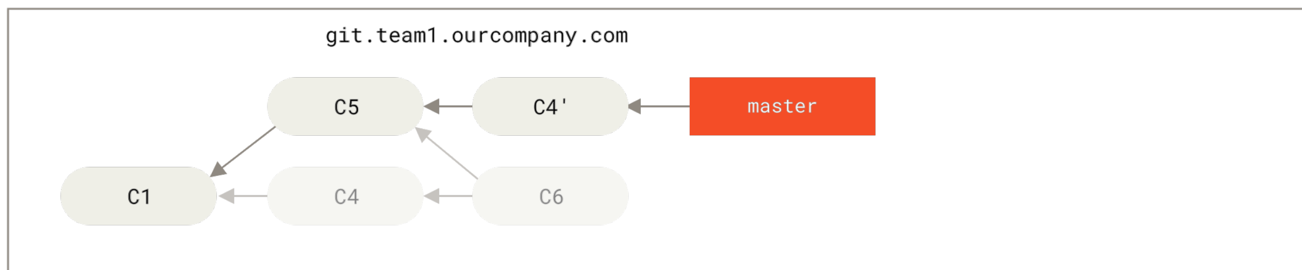
Оказва се, че в допълнение към SHA-1 чексумата на къмита, Git също така калкулира и чексума, която е базирана на поправката (patch-ът) въведена от него. Това се нарича “patch-id”.

Ако издърпате работа, която е била пренаписана и я пребазирате върху новите къмити на вашия колега, Git често може успешно да определи кое е изцяло ваше и да ги приложи обратно върху новия клон.

Например, в предишния сценарий, ако вместо да правим сливане докато сме в момента показан на фигурата Някой изпраща към сървъра пребазирани къмити, изоставяйки тези, върху които вие сте базирали работата си, изпълним `git rebase teamone/master`, Git ще направи следното:

- Ще определи коя работа е уникална за нашия клон (C2, C3, C4, C6, C7)
- Ще определи кои от къмитите не са (C2, C3, C4)
- Ще определи кои къмити не са били презаписани в целевия клон (само C2 и C3, защото C4 е същият patch като C4')
- Ще приложи тези къмити върху `teamone/master`

Така вместо резултатите, които виждаме във Вие сливате в същата работа отново, в нов merge commit, ще получим нещо приличащо повече на Пребазиране върху форсирано изпратена пребазирана работа



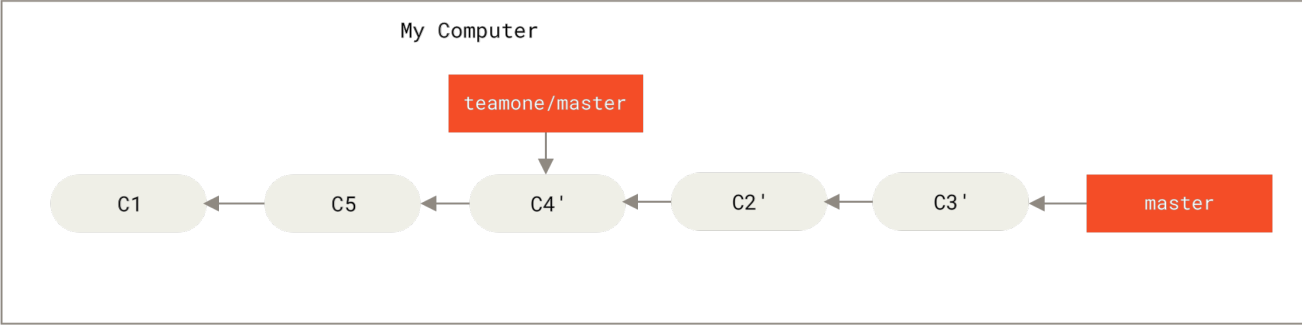


Figure 48. Пребазиране върху форсирано изпратена пребазирана работа

Това работи само, ако `C4` и `C4`, направени от колегата ви са почти един и същи patch. В противен случай, пребазирането няма да може да установи, че това е дублиране и ще добави още един подобен на C4 patch (който вероятно няма да може да се приложи чисто, понеже промените вече ще са поне някъде там).

Можете също така да опростите това изпълнявайки `git pull --rebase`, вместо нормален `git pull`. Или можете да го направите ръчно с `git fetch` последвана от `git rebase teamone/master` в този случай.

Ако използвате `git pull` и искате да включите `--rebase` аргумента по подразбиране, можете

да зададете `pull.rebase` конфигурационната стойност с `git config --global pull.rebase true`.

Ако пребазирате само къмити, които никога не са били публични, а са съществували само в локалния компютър - тогава няма да имате проблеми. Ако пребазирате публикувани вече къмити, за които знаете че никога не са използвани от някой друг - тогава също няма проблем. Ако обаче пребазирате къмити, които вече са изпратени и може би използвани като основа за работата на някой друг в екипа, тогава очаквайте много проблеми и негативно отношение в съвместната ви работа с колегите.

Ако вие или ваш колега установи към даден момент, че това е необходимо - уверете се, че поне всеки знае и пуска `git pull --rebase` за да се намалят неудобствата, когато проблемите дойдат.

Пребазиране или сливане

Сега, когато видяхте в действие пребазирането и сливането, може би се питате кое от двете е по-добро. Преди да отговорим, нека се върнем малко назад и да поговорим за това какво означава историята.

Една възможна дефиниция е, че историята на къмитите във вашето хранилище е **запис на това какво в действителност се е случвало.** Това е исторически документ, ценен сам по себе си, който не бива да се модифицира. Погледнато по този начин, излиза че промяната на историята на къмитите е почти богохулство - вие *лъжете* за това какво се е случвало всъщност. А какво ще е ако имахте разхвърляни серии от merge къмити? Ами - просто така са протекли нещата и хранилището следва да пази този факт за идните поколения.

Обратната гледна точка приема историята като **хронология на това как проектът ви е бил създаден**. Например, не бихте публикували първата чернова на една книга и ръководството за това как да поддържате вашия софтуер заслужава внимателно редактиране. Когато работите върху проект, може да ви трябва запис на всичките ви погрешни стъпки, но когато дойде време да споделите проекта с останалите, вероятно ще искате да покажете съвсем ясно как сте стигнали от точка A до точка B. Това е гледната точка на поддръжниците на инструменти като `rebase` и `filter-branch`, които искат да разкажат историята на проекта не буквално каквато е била, а каквато би била най-полезна за тези, които ще го поемат в бъдеще.

Обратно на въпроса кое е по-добро, пребазирането или сливането? Надяваме се виждате и сами, че отговорът не може да е еднозначен. Git е мощна система, която ви позволява да правите много неща с историята на проекта, но няма еднакви екипи и няма еднакви проекти. Накратко, най-добрият отговор на въпроса е, че трябва сами да решите кой подход е най-добър според конретната специфична ситуация.

Ако се колебаете, но искате да се възползвате от предимствата и на двата подхода, просто помнете това правило - спокойно използвайте пребазиране за локални промени, които сте направили, но не са публични. Така ще си подредите историята в чист вид. Но никога не пребазирайте нещо, което вече сте изпратили за споделено ползване.

Обобщение

Разгледахме основите на разклоненията и сливанията в Git. Би следвало сега да умеете с лекота да създавате и превключвате клонове, както и да сливате локални клонове заедно. Също така, би следвало да можете да споделяте клоновете си код изпращайки ги към общ сървър, да работите с други колеги по споделени клонове код и да пребазирате вашите клонове преди споделянето им. Следва да разгледаме какви ви е необходимо за да си пуснете свой собствен хостинг сървър за Git хранилища.

Git на сървъра

На този етап, би следвало да можете да вършите повечето си ежедневна работа, за която ви е нужен Git. Обаче, за да участвате с вашия труд в какъвто и да било вид съвместна работа с Git, ще ви е нужно отдалечено Git хранилище. Въпреки, че теоретично можете да пращате и издърпвате промени към хранилищата на индивидуални отделни колеги, това не е препоръчително, защото лесно можете да объркате нещата, върху които те работят, ако не сте внимателни. Освен това, вероятно искате вашите колеги да имат достъп до хранилището ви дори когато компютърът ви е без връзка с мрежата. Затова, препоръчителният начин да сътрудничите с други разработчици е да настроите междинно хранилище, към което да имат достъп всички, и да теглите и изпращате код през него.

Да пуснете Git сървър е сравнително лесно. Първо, избирате протоколите, които желаете той да поддържа за комуникация с клиентите. Първата секция от тази глава ще разгледа наличните протоколи в едно с предимствата и недостатъците им. Следващите части ще обяснят някои типични настройки свързани с тези протоколи. Накрая, ще преминем през няколко хоствани опции, в случай че нямате нищо против кода ви да се съхранява на външен сървър и не ви се занимава с настройката и поддръжката на собствен такъв.

Ако е така, можете да преминете директно към последната част на тази глава, за да разгледате какви са опциите за създаване на хостван акаунт и след това да разгледате следващата глава, където дискутираме предимствата и недостатъците на това да работите в distributed source control среда.

Отдалеченото хранилище накратко казано е *bare repository* — Git хранилище без работна директория. Понеже то се ползва само като обща точка за сътрудничество между членовете на екипа, няма необходимост в него да има snapshot от данните разположен на диска, там присъстват само Git данните. Казано възможно най-просто, едно голо (bare) хранилище има само съдържанието на директорията `.git` от вашия проект и нищо повече.

Комуникационни протоколи

Git може да използва 4 основни протокола за трансфер на данни: локален, HTTP, Secure Shell (SSH) и Git. Сега ще погледнем какво представляват те и по какви причини бихте желали да ги използвате (или избягвате).

Локален протокол

Най-простият от четирите е *локалният протокол*, при който отдалеченото хранилище се намира просто в друга директория на диска. Това често се ползва, ако всички в екипа ви имат достъп до споделена файлова система от рода на NFS споделено устройство или пък, в по-редки случаи, когато всички се логват на един и същи компютър. Последното не е препоръчително, защото всички копия на хранилището ще се пазят на едно място и рискът от загубата на данни при хардуерна повреда е значително по-голям.

Ако имате споделена монтирана файлова система, тогава вие клонирате, изпращате и изтегляте данни от локално, файлово-базирано хранилище. За да клонирате хранилище от този вид или за да добавите такова като отдалечено към съществуващ проект, използвайте

пътя до хранилището вместо URL. Например, за да клонирате, може да изпълните нещо подобно:

```
$ git clone /srv/git/project.git
```

Или пък така:

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

Git работи малко по-различно ако изрично укажете `file://` като префикс на пътя. Ако укажете само пътя, системата се опитва да използва hardlinks или директно копира файловете, които са нужни. Ако укажете `file://`, Git ще стартира процеси, които нормално се използват за мрежов трансфер, което е една идея по-малко ефективен метод за трансфер на данни. Главната причина да използвате префикса `file://` е, че можете да искате чисто копие на хранилището, без неприсъщи референции или обекти в него - основно при импорт от други VCS системи или нещо подобно (вижте Git на ниско ниво за повече подробности). Тук ние ще използваме нормални пътища, защото това почти винаги работи по-бързо.

За да добавите локално хранилище като remote към съществуващ Git проект, можете да направите това:

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

След това, можете да изпращате и издърпвате от него използвайки името му `local_proj` по същия начин, по който ако то беше в мрежа.

Предимствата

Предимствата на файл-базираните хранилища са в това, че са прости и използват наличните права за файловете и мрежовия достъп. Ако вече имате споделена файлова система, до която има достъп целия ви екип, създаването на хранилище е много просто. Пазите копие от bare хранилището някъде, където всички имат достъп и задавате права за чете и писане като на всяка друга споделена директория. Ще дискутираме как да експортирате копие от хранилището като bare копие за целта в Достъп до Git на сървъра.

Това също е полезен начин за бързо изтегляне на работата на друг човек от неговото работещо хранилище. Ако с ваш колега работите по един и същи проект и той поиска да погледнете нещо по неговата работа, то една команда от рода на `git pull /home/john/project` вероятно ще е по-лесна опция от това той да изпрати нещо до мрежовото хранилище и вие след това да го теглите при вас.

Недостатъци

Неудобствата при този подход се състоят в това, че споделеният достъп в повечето случаи се настройва и достъпва от различни локации по-трудно в сравнение със стандартния

мрежов достъп. Ако искате да изпратите данни от домашния си лаптоп, докато сте вкъщи, ще трябва да монтирате отдалечения диск, което често може да е трудно и досадно бавно.

Също така трябва да посочим, че локалният протокол не винаги е най-бързата опция, ако използвате споделен монтиран ресурс от някои видове. Локалното хранилище е бързо само, ако имате бърз достъп до данните. Едно хранилище разположено върху NFS ресурс често е по-бавно от SSH такова на същия сървър (което освен това позволява на Git да работи през локалните дискове на всеки от компютрите).

Накрая, този протокол не защитава хранилището от непредвидени поражения. Всеки потребител разполага с пълен шел достъп до "отдалеченото" хранилище и нищо не пречи на един невнимателен колега да промени или изтрие служебни Git файлове, което от своя страна да повреди цялото хранилище.

HTTP протоколите

Git може да работи през HTTP в два различни режима. Преди Git версия 1.6.6, начинът беше само един и то доста опростен и в общия случай - read-only. С тази версия обаче, беше представен нов, по-усъвършенстван протокол, който позволява на Git интелигентно да уговаря транфера на данни по маниер подобен на този, който се използва през SSH. В последните няколко години този нов HTTP протокол придоби голяма популярност, защото е по-прост за потребителя и по-интелигентен в механизма на комуникация. Тази нова версия често е наричана *Smart* HTTP протокол, докато старата е известна като *Dumb* HTTP. Ще разгледаме първо smart HTTP протокола.

Smart HTTP

Протоколът smart HTTP работи много подобно на SSH или Git протоколите, но използва стандартните HTTP/S портове и може да използва различни механизми за HTTP автентикация, което често го прави по-лесен за много потребители, защото можете да използвате неща като оторизиране с име и парола, вместо създаване на SSH ключове.

Този протокол в момента е най-популярния начин за използване в Git, понеже може да работи както анонимно, подобно на протокола `git://`, така и с криптиране подобно на SSH и автентикация. Вместо да създавате различни URLи за тези неща, сега можете да използвате единичен URL за всички тях. Ако се опитате да изпратите данни към хранилище настроено да изисква оторизация (както би следвало да е), сървърът може да ви попита за потребителско име и парола за достъп. Същото важи и за достъпа само за четене.

На практика, в услуги като GitHub, URL-ът който ползвате за да разглеждате хранилището в браузъра (например, https://github.com/schacon/simplegit) е същият URL, който можете да използвате за клонирането му или пък за изпращане на промени към него (ако имате права за това).

Dumb HTTP

Ако сървърът не разполага с Git HTTP smart услуга, Git клиентът ще се опита да използва протокола *dumb* HTTP. Този вид комуникация очаква bare Git хранилището да се обслужва като нормални файлове от уеб сървъра. Красотата на dumb протокола се крие в простотата

на настройката му. В общи линии, всичко което трябва да направите е да копирате едно bare Git хранилище там където уеб сървърът има достъп и да настроите специфичен `post-update` hook, след което сте готови (вижте Git Hooks). Сега всички, които имат достъп до уеб сървъра, ще могат да клонират хранилището ви. За да позволите достъп за четене до хранилището през HTTP, направете нещо такова:

```
$ 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
```

Това е всичко. Инструментът `post-update`, който идва с Git, по подразбиране изпълнява съответната команда (`git update-server-info`) така че HTTP издърпването и клонирането да работят коректно. Тази команда се изпълнява, когато изпращате към това хранилище, вероятно през SSH, след което други потребители могат да клонират посредством нещо такова:

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

В този специфичен случай, ние използваме `/var/www/htdocs` пътя, който е стандартен за Apache инсталациите, но вие можете да ползвате който и да е статичен уеб сървър — просто сложете bare хранилището в неговия път. Git данните се обслужват като базови статични файлове (вижте Git на ниско ниво за повече информация как точно става това).

Като обобщение, имате два избора, да пуснете read/write Smart HTTP сървър или read-only такъв с Dumb HTTP. Рядко се случва да се прави комбинация от двете.

Предимствата

Ще разгледаме предимствата на Smart HTTP версията на протокола.

Простотата да имате единичен URL за всички видове достъп и да оставите сървърът да пита за име и парола, когато се налага, прави нещата много по-лесни за крайния потребител. Възможността за оторизация с име и парола е голямо предимство сравнена с SSH, защото потребителите няма нужда да генерират SSH ключове локално и да изпращат публичния си ключ към сървъра преди да могат да комуникират с него. За по-неопитните потребители или за потребителите на системи, в които SSH не се ползва интензивно, това може да бъде голямо улеснение по отношение на лекотата на ползване. В допълнение, протоколът е бърз и ефективен, подобно на SSH.

Можете също да обслужвате хранилищата си само за четене през HTTPS, което значи че можете да криптирате съдържанието на трансфера или дори да стигнете и до по-рестриктивни мерки, като например да изисквате от клиентите да използват специфични signed SSL сертификати.

Друго предимство е и факта, че HTTP и HTTPS са толкова разпространени протоколи, че

корпоративните защитни стени често вече са настроени да пропускат трафика през техните портове.

Недостатъци

Git през HTTP/S може да е една идея по-сложен за настройване в сравнение с SSH на някои сървъри. Отделно от това, съществува съвсем леко предимство, което другите протоколи за обслужване на Git имат, в сравнение със Smart HTTP.

Ако използвате HTTP за автентикирано изпращане на промени към хранилището, изпращането на името и паролата понякога може да е малко по-сложно отколкото използването на SSH ключове. Обаче, съществуват няколко credential caching инструменти, които можете да ползвате, включително Keychain access на OSX или Credential Manager под Windows, за да си улесните нещата. Погледнете Credential Storage система за да видите как да настроите системата за защитено кеширане на HTTP пароли.

SSH протоколът

Често използван протокол за Git при self-hosting инсталации е SSH. Това е защото SSH достъпът до сървърите е много разпространен и настроен на повечето от тях - а и да не е, лесно може да се пусне. SSH също така е автентикиран мрежов протокол, повсеместно използван и лесно управляем.

За да клонирате Git хранилище през SSH, използвайте `ssh://` URL като този:

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

Или, може да предпочетете съкратения, подобен на scp синтаксис:

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

И в двата случая отгоре, ако не укажете потребителско име, Git ще използва това, с което сте логнати в системата.

Предимствата

Предимствата на SSH са много. Първо, SSH е сравнително лесен за настройка - SSH демоните са често използвани и познати, повечето системни администратори имат опит с тях и повечето OS дистрибуции идват с настроен SSH или имат съответните средства за настройка и управление на SSH комуникация. След това, комуникацията е сигурна - целият трансфер през SSH е криптиран и оторизиран. Последно, подобно на HTTP/S, Git и Local протоколите, SSH е ефективен и прави данните максимално компактни преди да ги изпрати.

Недостатъци

Негативната страна на SSH е, че не можете да имате анонимен достъп. Потребителите трябва да имат достъп до машината ви през SSH, за да се доберат до хранилището ви, дори и

в режим само за четене, което не прави SSH толкова подходящ за проекти с отворен код. Ако го използвате само в рамките на корпоративната мрежа, SSH може да е единственият протокол, с който се налага да работите. Ако искате да позволите анонимен достъп само за четене до вашите проекти и същевременно искате да ползвате SSH, ще трябва да настроите SSH за вас, за да изпращате до хранилището, но за колегите, които ще теглят - ще трябва да настроите друг метод.

Git протокол

Следва протоколът Git. Той е реализиран чрез специален daemon, който идва заедно с Git, слуша на специфичен порт (9418) и осигурява услуга подобна на тази на SSH, но без абсолютно никаква автентикация. Ако искате хранилището ви да е достъпно през този протокол, трябва да създадете файл `git-daemon-export-ok`, иначе daemon-ът няма да го обслужва. Това е единственият вид защита при този протокол. Такова Git хранилище няма опции - или е достъпно за клониране от всички или не е. Това означава, че по подразбиране при този протокол не можете да изпращате данни към хранилището (pushing). Можете да го разрешите, но предвид тоталната липса на автентикация, това не е добра идея - всеки в Интернет, който се сдобие с URLа на хранилището ви, ще може да го променя. Достатъчно е да се каже, че това е рядкост.

Предимства

Git протоколът често е най-бързия мрежов протокол. Ако обслужвате голям трафик за публичен проект, или пък проектът е много голям и не изисква автентикация за четене, вероятно ще си заслужава да пуснете Git daemon. Той използва същия механизъм за трансфер на данни като SSH, но без забавянето за криптиране и автентикация.

Недостатъци

Недостатъкът на Git протокола е липсата на автентикация. Като цяло е нежелателно Git протоколът да е единствения протокол за достъп до вашия проект. В повечето случаи, може да го ползвате в комбинация с SSH или HTTPS за достъп от ваши сътрудници, които трябва да имат права за запис в хранилището, а всички останали - read-only достъп през `git://`. Освен това, Git вероятно е най-трудния за настройка протокол. Той трябва да пусне свой собствен daemon, което може да изисква `xinetd`/`systemd` конфигурация или нещо подобно, а това не е сред най-приятните неща за един системен администратор. Също така, протоколът изисква защитната стена да пропуска трафик през порт 9418, което не е стандартна опция за корпоративните такива и обикновено се блокира.

Достъп до Git на сървъра

Сега ще разгледаме настройката на Git услуга, ползваща тези протоколи на ваш собствен сървър.

Тук ще демонстрираме командите и стъпките за базови опростени инсталации на Linux базиран сървър, но разбира се, възможно е това да стане на macOS и Windows машини. В действителност, изграждането на production сървър в рамките на вашата инфраструктура ще изисква различни стъпки по отношение на мерките за сигурност или според конкретните инструменти на операционната ви система, но да се надяваме, че тези стъпки ще ви дадат първоначална насока за това какво се изисква.

Като първа стъпка, за да получите Git на сървъра, ще трябва да експортирате налично хранилище в ново, bare хранилище - това е хранилище, което не съдържа работна директория. Това обикновено е съвсем лесно. Използвайте командата за клониране с параметър `--bare`. По конвенция, директориите за bare хранилището завършват на `.git`, например така:

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

Сега трябва да имате копие от Git директорията във вашата директория `my_project.git`.

Това е приблизително еквивалентно на резултата от командата:

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

Съществуват някои незначителни разлики в конфигурационния файл, но за нашите цели резултатът е почти един и същ. Командата взема Git хранилището, без работната му директория и създава директория специално за него.

Изпращане на Bare хранилище към сървъра

След като вече имате копие на хранилището, всичко което трябва да сторите е да го копирате на сървъра и да настроите съответния протокол/протоколи за достъп. Нека кажем, че имате сървър наречен `git.example.com`, към който имате SSH достъп и искате да пазите всичките си Git хранилища в директорията `/srv/git`. Като приемаме, че `/srv/git` съществува на сървъра, можете да създадете ново хранилище копирайки наличното такова:

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

В този момент, другите потребители с SSH достъп до същия сървър и права за четене към `/srv/git` директорията му, вече могат да клонират вашето хранилище изпълнявайки:

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

Ако някой от тях има и права за писане до директорията `/srv/git/my_project.git`, то той ще има автоматично и push права до хранилището.

Git автоматично ще добави group write права до хранилището по коректен начин, ако изпълните `git init` с параметъра `--shared`. Изпълнението на тази команда не унищожава никакви къмити, референции или други обекти

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

Виждате колко лесно е да вземете Git хранилище, да създадете bare версия и да го поставите в сървър, към който колегите ви имат SSH достъп. Сега сте готови да работите съвместно по проекта.

Важно е да посочим, че това буквално е всичко, от което имате нужда за да пуснете използваем Git сървър - просто добавете акаунти с SSH достъп за колегите ви и копирайте едно bare хранилище там, където те имат права за четене и писане. Сега сте готови, не трябва нищо повече.

В следващите секции ще видим как да направим по-модерни конфигурации. Ще направим обзор на това как да настроите нещата така, че да не се нуждаете от отделни акаунти за всеки потребител, как да добавим публичен достъп за четене до хранилища, настройване на уеб потребителски интерфейси и др. Обаче, просто помнете, че това са допълнения - всичко, което ви *трябва* за да работите съвместно по частен проект е SSH сървър и bare хранилище.

Малки конфигурации

Ако сте малък екип или просто тествате Git във вашата организация и имате само няколко разработчика, нещата могат да са простички за вас. Един от най-сложните аспекти в настройката на Git сървъра е управлението на потребителите. Ако искате някои хранилища да са само за четене за определени потребители, а други да са достъпни за писане, то настройките на достъпа и съответните права могат да са по-трудни за наместване.

SSH достъп

Ако имате сървър, към който всичките ви колеги имат SSH достъп, най-лесно е да разположите хранилищата си в него, защото както видяхме в предната секция - няма да имате почти никаква работа по настройките. Ако искате по-комплексен контрол на достъпа, можете да се справите с нормалните средства за достъп до файловата система, които операционната система предлага.

Ако искате да разположите хранилищата си на сървър, който няма акаунти за всички в екипа ви, за които допускате, че ще е нужен достъп с права за писане, тогава трябва да настроите SSH достъп за тях. Допускаме, че ако имате сървър с който да правите това, вече имате инсталиран SSH за достъп до него.

Има няколко начина да дадете достъп на всеки от екипа. Първо, можете да създадете

акаунти за всички колеги, което е лесно, но може да е досадно. Може да не искате да изпълнявате `adduser`/`useradd` и да правите временни пароли за всеки колега.

Втори начин е да създадете единичен *git* потребител на машината, да помолите всеки ваш колега, който трябва да има права за писане да ви изпрати свой SSH публичен ключ, и да добавите ключовете във файла `~/.ssh/authorized_keys` на потребителя *git*. Така всеки от колегите ви ще има достъп до машината през потребителското име *git*. Това не засяга по никакъв начин commit данните — SSH потребителят, с който се свързвате към машината не се отразява на записаните къмити.

Друг начин е да настроите вашия SSH сървър да автентикира потребителите през LDAP сървър или някакъв друг централизиран източник за автентикация, който може да имате. Докато всеки от потребителите може да получи шел-достъп на машината, всеки SSH оторизационен механизъм за който се сещате, би трябвало да работи.

Генериране на SSH публичен ключ

Много Git сървъри автентикират достъпа ползвайки SSH публични ключове. За да осигури такъв ключ, всеки потребител на системата ви трябва първо да си го генерира. Процесът е подобен за всички операционни системи. Първо, трябва да проверите дали вече нямате ключ. По подразбиране, генерираните от потребителя SSH ключове се пазят в директория `~/.ssh` в домашната му папка. Може лесно да проверите дали имате ключове като просто отворите директорията и покажете съдържанието ѝ:

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

Търсите чифт файлове с имена от рода на `id_dsa` или `id_rsa` и съответен файл с разширение `.pub`. Файлът с разширение `.pub` е публичният ви ключ, а другият е секретния. Ако нямате тези файлове (или дори не разполагате с `.ssh` директория), можете да ги създадете с програмата `ssh-keygen`, която идва с пакета SSH под Linux/macOS и също така с Git for Windows:

```
$ ssh-keygen -o
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
```

Програмата първо пита къде да съхранява ключа (`.ssh/id_rsa`) и след това пита два пъти за

парола, която можете да оставите празна, ако не желаете да я въвеждате всеки път, когато използвате ключа. Обаче, ако използвате парола, уверете се че сте добавили флага `-o`; това ще съхрани частния ключ във формат, който е по-устойчив на brute-force атаки за пароли в сравнение с формата по подразбиране. Може също да използвате `ssh-agent` инструмента за да избегнете въвеждането на паролата всеки път.

След това, всеки потребител трябва да изпрати публичния си ключ на вас или който администрира Git сървъра (подразбираме, че използвате схема, при която SSH сървърът изисква публични ключове). Всичко което трябва да се направи е да се копира съдържанието на `.pub` файла и да се изпрати по имейл. Публичните ключове изглеждат по подобен начин:

```
$ 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
```

За повече информация и детайлно упътване за създаване на SSH ключове на множество операционни системи, погледнете GitHub SSH keys страницата на адрес https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent.

Настройка на сървъра

Нека преминем през настройката на SSH достъпа от страна на сървъра. В този пример, ще използвате метода `authorized_keys` за автентикиране на вашите потребители. Подразбираме също така, че използвате стандартна Linux дистрибуция, например Ubuntu.

Голяма част от описаното тук може да се автоматизира с командата `ssh-copy-id`, вместо чрез ръчно копиране и инсталиране на публични ключове.

Първо, създавате `git` потребител и `.ssh` директория за него.

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

След това, трябва да добавите няколко публични ключа на разработчици към файла `authorized_keys` на потребителя `git`. Нека кажем, че имате няколко такива ключа и ги съхранявате във временни файлове. Да припомним, публичните ключове изглеждат така:

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

Просто трябва да ги добавите към `authorized_keys` файла на потребителя `git` в `.ssh` директорията му:

```
$ 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
```

Сега можете да инициализирате празно хранилище за тях изпълнявайки `git init` с опцията `--bare`, което ще създаде хранилище без работна директория:

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

След като направите това, John, Josie, или Jessica могат да изпратят първата версия на своя проект в това хранилище като го добавят като отдалечено и изпратят някой клон. Отбележете, че е необходимо някой да се логва в тази машина и да създава празно хранилище всеки път, когато искате да добавите проект. Нека ползваме `gitserver` за име на сървъра, който настроихме. Ако го използвате само локално и настроите DNS сървъра си да сочи към адреса му, тогава може да използвате командите буквално така (подразбираме, че `myproject` е съществуващ проект с файлове):

```
# от компютъра на John
$ cd myproject
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

Сега вече другите могат да клонират проекта и да изпращат промени към него също така лесно:

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

Ползвайки този подход можете лесно да пуснете read/write Git сървър за малък екип разработчици.

Следва да сте забелязали, че сега всички тези потребители могат също така да се логнат на сървъра като потребител `git`. Ако искате да ограничите това, ще трябва да смените шела на `git` с нещо различно във файла `/etc/passwd`.

Можете лесно да ограничите `git` потребителя само до Git дейности с рестриктивния инструмент `git-shell`, който идва с Git. Ако го използвате за login шел за вашия `git` потребител, то той ще има доста по-ограничени права в сървъра. Просто използвайте `git-shell` вместо `bash` или `csh` за шел на потребителя. За да го направите, първо трябва да добавите `git-shell` към `/etc/shells`, ако той вече не е там:

```
$ cat /etc/shells   # проверявате дали git-shell е вече във файла и ако не е...
$ which git-shell   # уверете се, че git-shell е инсталиран на системата
$ sudo -e /etc/shells  # и добавете пътя до него, който показва командата which
```

Сега можете да редактирате шела за даден потребител изпълнявайки `chsh <username> -s <shell>`:

```
$ sudo chsh git -s $(which git-shell)
```

Сега вече `git` потребителят може да използва SSH комуникация само за да изтегля и изпраща Git хранилища и няма да има пълноценен шел достъп в машината. Ако пробвате, ще получите отказ:

```
$ 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.
```

В този момент обаче, потребителите все още могат да използват SSH port forwarding за достъп до всеки хост, който git сървърът вижда. Ако искате да избегнете това, може да редактирате файла `authorized_keys` и да добавите следните опции за всеки ключ, който искате да ограничите:

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

Резултатът трябва да изглежда така:

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h
PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC
IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd
LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ
ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...
```

Сега мрежовите команди на Git ще работят нормално, но потребителите няма да имат шел достъп. Както се вижда от изхода на командата, можете също така да направите директория в домашната такава на потребителя `git`, което ще специализира малко `git-shell` командата. Например, можете да ограничите наличните Git команди, които сървърът приема или да промените съобщението, които потребителите виждат, ако се опитат да се логнат през SSH. Изпълнете `git help shell` за повече информация за настройване на шела.

Git Daemon

Следващата стъпка е да настроим демон, който обслужва хранилища чрез “Git” протокола. Това е често срещан избор за бърз достъп до вашите Git данни, без автентикация. Помнете, че в този случай данните ви са публично достъпни в рамките на съответната мрежа.

Ако използвате протокола на сървър извън защитната стена, това следва да става само за публично достъпни проекти. Ако сте зад защитна стена обаче, бихте могли да го ползвате за проекти, до които достъп за четене трябва да имат голям брой сътрудници и за които не желаете да настройвате SSH ключове поотделно.

Независимо от случая, Git протоколът е сравнително лесен за настройка. В общи линии, трябва да изпълните долната команда в daemonized режим:

```
$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

Опцията `--reuseaddr` позволява сървърът да се рестартира без да изчаква таймаут на старите конекции, а `--base-path` позволява на хората да клонират проекти без да трябва да указват пълния път. Пътят в края на командата указва на Git демона къде да следи за хранилища за експорт. Ако използвате защитна стена, ще трябва също така да разрешите достъпа до порт 9418 на сървъра.

Пускането на демона може да се прави по различни начини в зависимост от използваната операционна система.

Понеже `systemd` вече е най-разпространената init система в модерните Linux дистрибуции,

бихте могли да я ползвате за целта. Просто създайте файл `/etc/systemd/system/git-daemon.service` със следното съдържание:

```
[Unit]
Description=Start Git Daemon

[Service]
ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/ /srv/git/

Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon

User=git
Group=git

[Install]
WantedBy=multi-user.target
```

Може да видите, че Git демонът се пуска с потребител и група `git`. Ако се налага, променете ги според вашите нужди и се уверете, че потребителят който сте написали съществува в системата. Също така, проверете дали изпълнимият файл се намира в `/usr/bin/git` и го коригирайте, ако не е.

Накрая трябва да изпълните `systemctl enable git-daemon` за да пуснете услугата автоматично при рестарт на компютъра. Ръчното пускане и спиране на услугата се прави с командите `systemctl start git-daemon` и `systemctl stop git-daemon`, съответно.

На други системи, може да искате да ползвате `xinetd`, скрипт в `sysvinit` системата или пък нещо друго като похват — трябва просто да пуснете командата като демон и да наблюдавате статуса ѝ.

Остава да кажете на Git в кои хранилища трябва да се разреши свободния достъп през Git протокола. Можете да направите това във всяко хранилище създавайки файл с име `git-daemon-export-ok`.

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

Наличието му казва на Git, че е OK да обслужва съответния проект без автентикация.

Smart HTTP

Вече имаме автентикиран достъп през SSH и неавтентикиран през `git://` протокола, но съществува и протокол, който може да прави и двете едновременно. Процесът по настройка

на Smart HTTP всъщност се свежда до разрешаването на CGI скрипт, който идва с Git и е известен като `git-http-backend` на сървъра. Този скрипт ще чете пътя и хедърите изпратени от `git fetch` или `git push` към HTTP URL и ще разбере дали клиентът може да комуникира през http (което е така за всеки клиент след версия 1.6.6). Ако CGI скриптът види, че клиентът е смарт, той ще комуникира с него по интелигентен начин. В противен случай, ще се върне към по-опростения способ (така че да е обратно съвместим с по-старите клиенти).

Нека минем през съвсем простата настройка. Ще използваме Apache като CGI сървър. Ако нямате настроен Apache, можете да го направите в Linux система приблизително така:

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

Това ще разреши модулите `mod_cgi`, `mod_alias`, и `mod_env`, които са необходими за нашите цели.

Ще трябва също така да промените групата на директориите в `/srv/git` на `www-data`, така че уеб сървърът да има права за четене и писане в хранилищата, защото инстанцията на Apache, която ще изпълнява CGI скрипта, ще работи по подразбиране като този потребител:

```
$ chgrp -R www-data /srv/git
```

След това трябва да променим някои неща по конфигурацията на Apache, така че да използва `git-http-backend` скрипта като средство за обработка на всички заявки, идващи в `/git` пътя на уеб сървъра.

```
SetEnv GIT_PROJECT_ROOT /srv/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

Ако оставите променливата `GIT_HTTP_EXPORT_ALL`, тогава Git ще допуска неавтентикираните потребители само до хранилищата, съдържащи файла `git-daemon-export-ok` - точно както го прави и Git демона.

Накрая, ще искаме да накараме Apache да позволява заявки от автентикирани потребители с права на запис към `git-http-backend` с блок подобен на следния:

```
<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /srv/git/.htpasswd
    Require expr !(%{QUERY_STRING} -strmatch '*service=git-receive-pack*' ||
%{REQUEST_URI} =~ m#/git-receive-pack$#)
    Require valid-user
</Files>
```

Това значи, че трябва да създадете `.htpasswd` файл, съдържащ паролите за всички валидни потребители. Ето пример за добавен потребител “schacon” в такъв файл:

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

Има много различни начини да накарате Apache да автентикира потребители, просто трябва да изберете подходящия за вас. Тук просто посочихме един от най-простите варианти. Вероятно също ще искате да настроите достъп през SSL, така че комуникацията да е криптирана.

Няма да навлизаме по-навътре в конфигурационните детайли на Apache, тъй като може да използвате различни похвати за автентикация или изцяло различен уеб сървър. Идеята е, че Git идва с CGI скрипт наречен `git-http-backend`, който когато бъде извикан, ще поеме цялата комуникация по приемането и изпращането на данни през HTTP. Самият скрипт не извършва сам по себе си никаква автентикация, но това може лесно да бъде контролирано при уеб сървъра, който го извиква. Можете да го използвате почти с всеки CGI уеб сървър, така че изберете този, който предпочитате.

За повече информация за настройка на Apache, вижте документацията тук: https://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Сега, когато имате базисен read/write и read-only достъп до вашия проект, може да искате да добавите прост уеб базиран визуализатор. Git предоставя CGI скрипт наречен GitWeb, който понякога се ползва за целта.

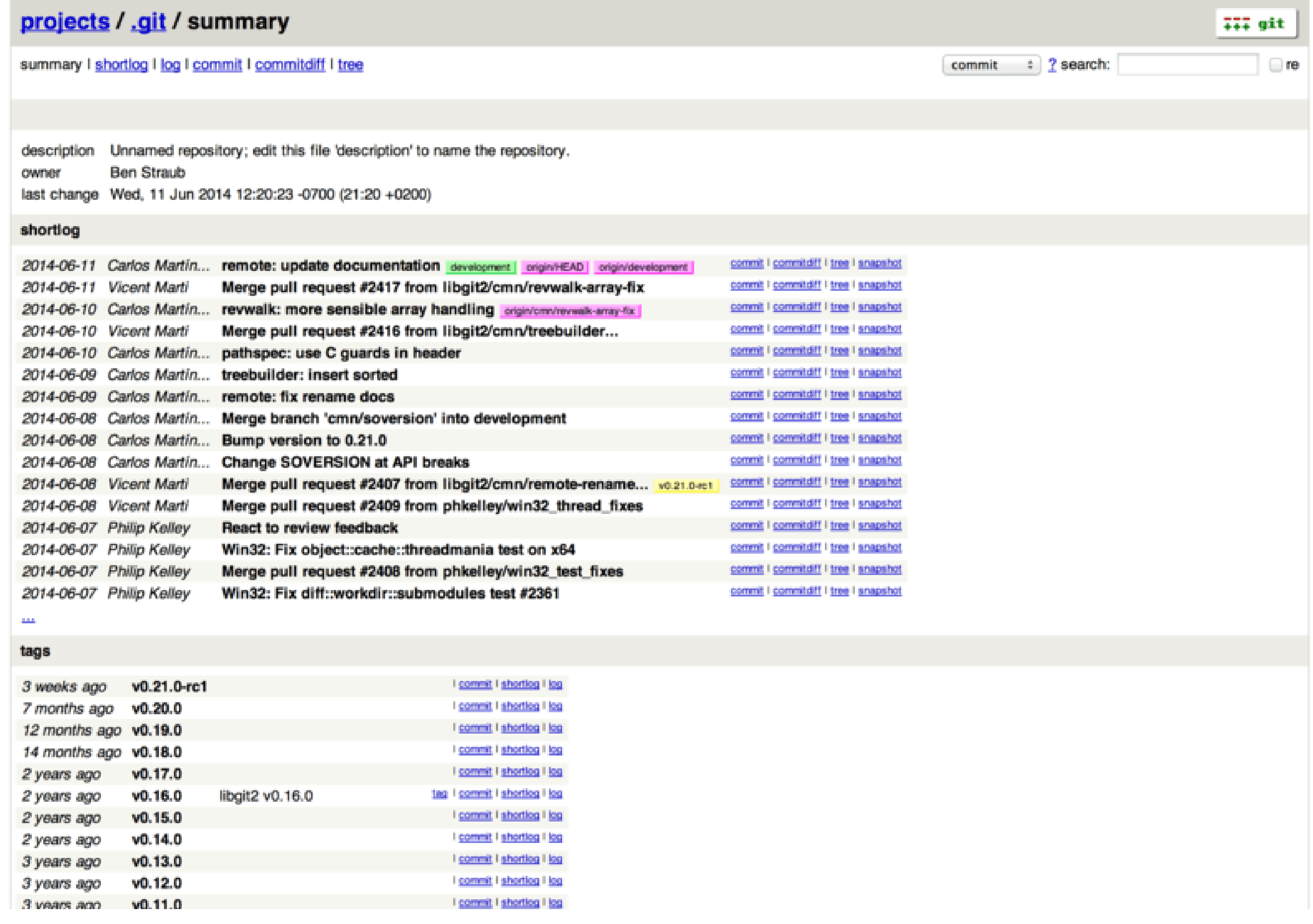


Figure 49. GitWeb уеб базиран потребителски интерфейс

Ако искате да проверите как ще изглежда GitWeb за вашия проект, Git предлага команда с която да пуснете временна инстанция при условие че системата ви има инсталиран олекотен уеб сървър като `lighttpd` или `webrick`. На Linux машини, `lighttpd` често идва предварително инсталиран, така че може да го стартирате с командата `git instaweb` в директорията на проекта. Ако сте на Mac, Leopard идва с инсталиран Ruby, така че `webrick` може да е удобна опция. За да пуснете `instaweb` с non-lighttpd сървър, можете да я изпълните с аргумента `--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]
```

Това ще ви пусне HTTPD сървър на порт 1234 и след това автоматично ще се стартира уеб браузър, който отваря тази страница. Лесно е от ваша страна. Когато сте готови и искате да спрете сървъра, изпълнете командата с аргумента `--stop`:

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

Ако пък искате уеб интерфейсът да е постоянно достъпен, например за екипа ви или за проект с отворен код, ще трябва да направите така, че CGI скриптът да се обслужва от нормален уеб сървър. Някои Linux дистрибуции имат пакета `gitweb`, който може да се инсталира през `apt` или `dnf`, така че може да ползвате и този начин. Ще преминем набързо

през инсталацията на GitWeb. Първо, ще ви трябва сорс кода на Git, с който идва GitWeb и след това генерирате custom CGI скрипт:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/srv/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/
```

Отбележете, че трябва да кажете на командата къде да намира Git хранилищата ви посредством променливата `GITWEB_PROJECTROOT`. След това, трябва да накарате Apache да използва CGI за този скрипт, за което може да добавите виртуален хост:

```
<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>
```

Да кажем пак, GitWeb може да се обслужва с произволен CGI или Perl съвместим уеб сървър; ако предпочитате различен от Apache, няма проблем да го ползвате. В този момент трябва да можете да отворите адреса `http://gitserver/` за да видите хранилищата си онлайн.

GitLab

GitWeb е доста семпъл вариант за визуализация. Ако търсите по-модерен, напълно функционален Git сървър, съществуват няколко проекта с отворен код, които са на ваше разположение. Понеже GitLab е един от най-популярните, ще разгледаме него за пример. Това е малко по-сложно от GitWeb и вероятно ще изисква повече поддръжка, но е и много по-пълноценна алтернатива.

Инсталация

GitLab е уеб приложение използващо база данни, така че инсталацията му изисква малко повече усилия сравнена с някои други Git сървъри. За щастие, този процес е много добре документиран и проектът активно се поддържа. GitLab горещо препоръчва инсталирането

на сървъра ви да се прави през официалния Omnibus GitLab пакет.

Другите опции за инсталиране са:

- GitLab Helm chart, за ползване с Kubernetes.
- Dockerized GitLab пакети за използване с Docker.
- От сорс-файлове.
- От облачен доставчик като AWS, Google Cloud Platform, Azure, OpenShift и Digital Ocean.

За повече информация прочетете GitLab Community Edition (CE) ръководството.

Администрация

Административният интерфейс на GitLab е достъпен през браузър. Отворете страницата на IP адреса на машината, на която е инсталиран GitLab и се логнете като администратор. Фабричното потребителско име е `admin@local.host`, а паролата `5iveL!fe` (ще бъдете помолени да я промените веднага след входа). След това натиснете иконата "Admin area" в менюто в горния десен край.

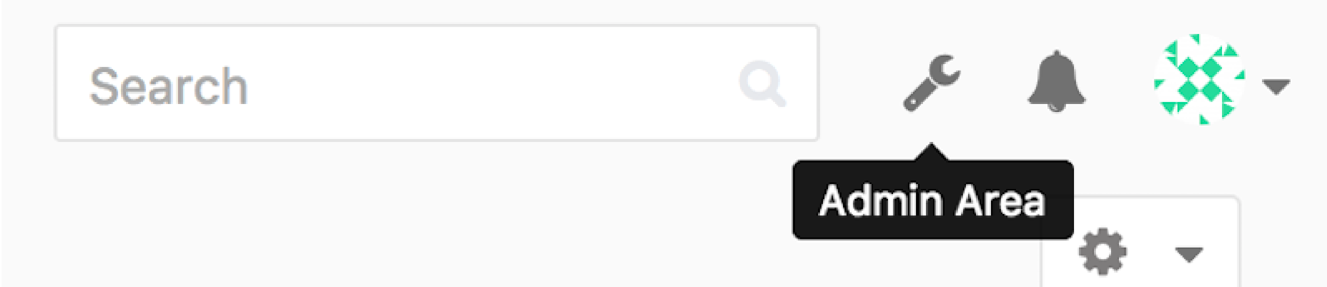


Figure 50. Admin area елемента в менюто на GitLab

Потребители

Потребителите в GitLab са акаунти, които съответстват на хора. Самите акаунти не са сложни; в основни линии представляват колекция от персонални данни прикачени към логин информацията. Всеки потребителски акаунт притежава **namespace**, за логическо групиране на проектите, които притежава. Така, ако потребител с име `jane` има проект с име `project`, то този проект ще е с url `http://server/jane/project`.

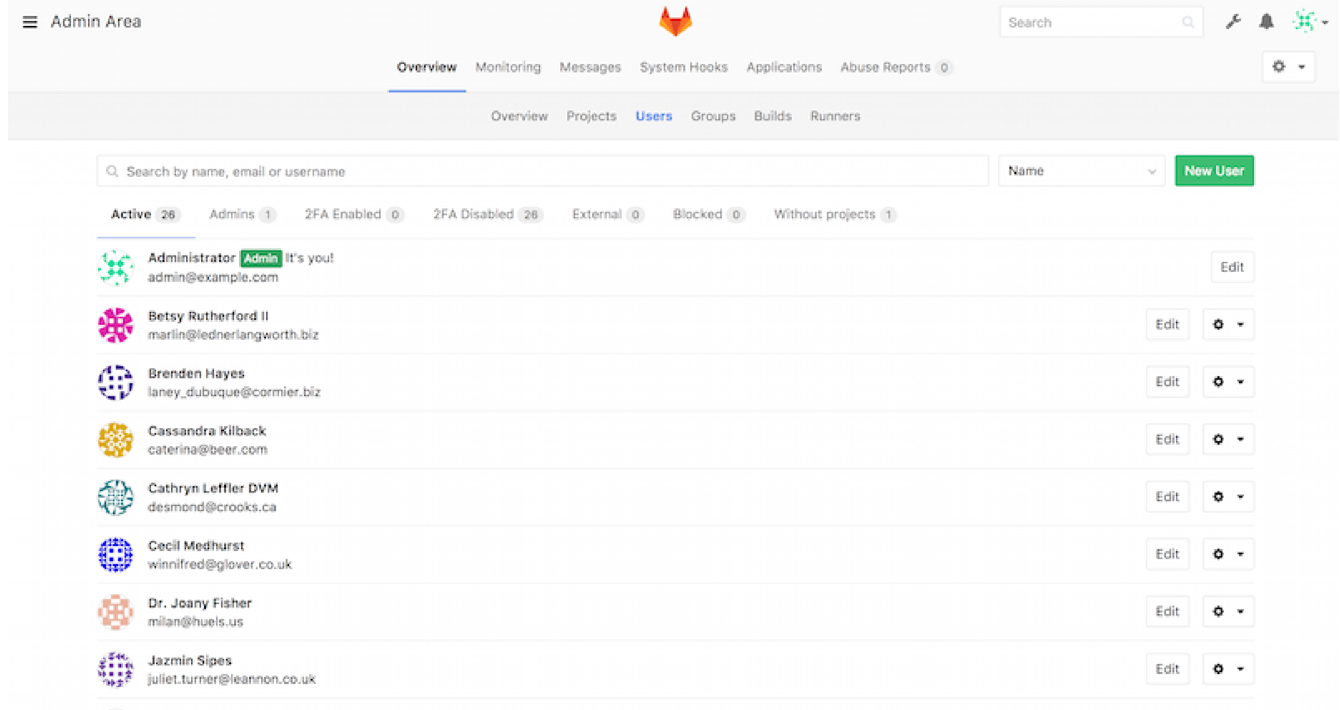


Figure 51. Административният екран на GitLab за управление на потребители

Изтриването на потребител може да стане по два начина: "Блокирането" на потребител забранява логването му в GitLab инстанцията, но всички данни в съответния namespace ще бъдат запазени и всички къмити подписани с имейл адреса на този потребител ще сочат все още към съответния потребителски профил.

"Изтриването" от своя страна, тотално унищожава потребителя от базата данни и файловата система. Всички проекти и данните в съответния namespace се изтриват и всички групи, които потребителят притежава, също се изтриват. Очевидно тази втора опция е много по-деструктивна и се използва рядко.

Групи

GitLab групата представлява съвкупност от проекти плюс информация за това как потребителите имат достъп до тях. Всяка група има namespace за проектите (по същия начин като потребителите), така че ако групата `training` съдържа проект `materials`, то неговият url ще бъде `http://server/training/materials`.

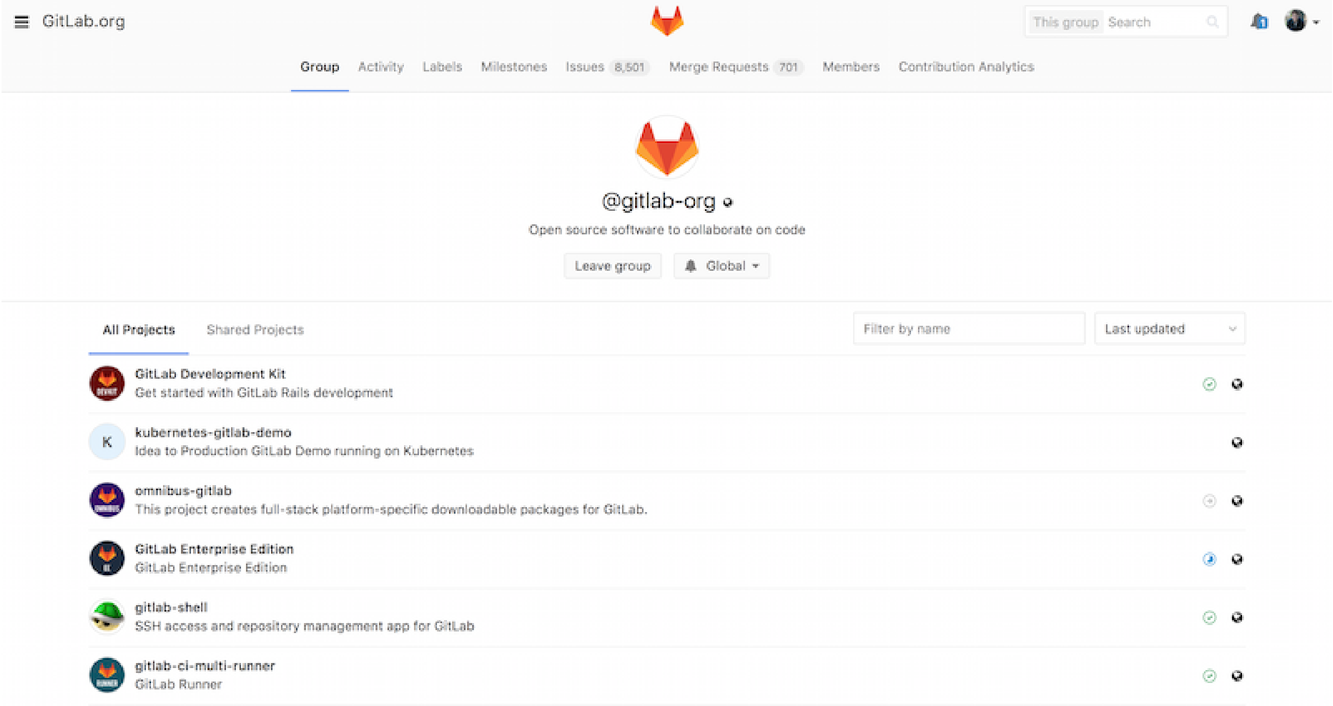


Figure 52. Административният екран за групите в GitLab

Всяка група е асоциирана с множество потребители, всеки от които разполага с ниво на достъп до групата и проектите в нея. То варира от “Guest” (с достъп до issues и chat) до “Owner” (пълен контрол над групата, членовете ѝ и нейните проекти). Типовете права са твърде много, за да ги изброяваме всичките, но GitLab има полезни линкове за помощ в административния интерфейс.

Проекти

Един GitLab проект приблизително съответства на единично Git хранилище. Всеки проект принадлежи на един namespace, потребителски или на група. Ако проектът принадлежи на потребител, то този потребител има пряк контрол върху това кой ще има достъп до него; ако е в група, то тогава user-level правата на групата ще се приложат съответно.

Всеки проект има също така ниво на видимост, което контролира кой има права за четене на страниците и хранилището му. Ако един проект е *Private*, то собственикът му трябва изрично да даде права за достъп на него до потребителите, които пожелае да го виждат. *Internal* проектите са видими за всеки логнат потребител, а *Public* проектите са видими за всички. Това контролира както `git fetch` достъпа, така и достъпа до уеб UI интерфейса за този проект.

Hooks

GitLab поддържа hooks като способ за сигнализация, на ниво проект и система. За всеки от тях, GitLab сървърът ще направи HTTP POST заявка с описателна JSON-фирматирана информация, когато възникнат съответните събития. Това е полезен начин да свържете вашите Git хранилища и GitLab инстанцията си към останалата част от автоматизацията на разработките като например CI сървъри, чат стаи или deployment инструменти.

Използване

Вероятно най-напред ще искате да създадете GitLab проект. Това се прави чрез иконата "+" на лентата с инструменти. Ще трябва да въведете име за проекта, към кой namespace ще принадлежи той и какво ниво на видимост ще има. Повечето от тези неща не са перманентни и могат да се редактират по-късно. Натиснете "Create Project" и сте готови.

След като проектът съществува, вероятно ще искате да го свържете с локално Git хранилище. Всеки проект е достъпен през HTTPS или SSH и всеки от тези протоколи може да се използва като Git remote. Адресите са видими в горната част на страницата на проекта. В налично локално Git хранилище, тази команда ще създаде отдалечена референция с име `gitlab` към хостваната локация:

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

Ако нямате локално копие на хранилището, може просто да изпълните:

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

Уеб интерфейсът осигурява достъп до множество удобни изгледи на самото хранилище. Домашната страница на всеки проект показва последната активност и линковете в горната част ще ви преведат до изгледи на файловете в проекта и историята на къмитите.

Съвместна работа

Най-простият начин за съвместна работа в GitLab е да дадете на друг потребител директно push права за достъп до Git хранилището ви. Можете да добавите потребител към проекта от секцията "Members" в настройките на проекта, където може да асоциирате и ниво на достъп за него (различните нива на достъп се разглеждат по-подробно в Групи). Давайки на потребителя ниво "Developer" или по-високо, вие му позволявате свободно да изпраща къмити и клонове директно в хранилището.

Друг метод за колаборация е чрез използването на merge requests. Тази функция позволява на всеки потребител, който вижда даден проект, да сътрудничи в него по контролиран начин. Потребителите с директен достъп могат просто да създадат клон, да изпратят в него къмити и да отворят merge request от техния клон обратно в `master` или който и да е друг клон. Потребителите без push права за дадено хранилище могат да го клонират при себе си ("fork"), да правят къмити в *това* тяхно копие и да отворят merge request от него обратно към основния проект. Този модел на работа позволява на собственика на проекта да има пълен контрол върху това какво и кога влиза в хранилището и едновременно с това да позволява сътрудничество от непознати потребители.

Merge requests и issues са основните обекти на дискусиите в GitLab. Всеки merge request позволява line-by-line дискусия на предложената промяна (което поддържа олекотен вид code review), както и обща дискусионна нишка. И двата обекта могат да се асоциират към потребители или да се организират в milestones.

Тази секция е фокусирана основно върху Git функциите на GitLab, но като един наистина зрял проект, той предлага много други възможности за съвместна работа на екипа ви като например множество wiki за проектите и инструменти за системна поддръжка. Едно от предимствата на GitLab е в това, че веднъж настроен и пуснат, рядко ще ви се налага да променяте конфигурационен файл или да влизате в сървъра през SSH; повечето административни задачи се осъществяват през уеб интерфейса.

Други опции за хостване

Ако не желаете да се занимавате с работата по инсталация и поддръжка на собствен Git сървър, налице са доста опции за хостване на Git проектите ви на външен такъв. Това си има предимства: обикновено хостваната услуга е лесна за създаване и поддържане на проекти и освен това не се нуждае да правите поддръжка и мониторинг. Дори и да сте си инсталирали собствен сървър, може все още да искате да се възползвате от публичните хостинг услуги за съхранение на проектите ви с отворен код – така по-лесно ще бъдете в контакт с общността от open source разработчици.

В днешни дни броят на хостинг опциите е достатъчно голям, така че да изберете вашата претегляйки предимствата и недостатъците им. Актуален списък на вариантите се поддържа в страницата GitHosting в основното wiki на Git на адрес https://git.wiki.kernel.org/index.php/GitHosting.

Ще разгледаме в детайли GitHub в главата GitHub, понеже е най-голямата хостинг система в момента и вероятно ще искате да я ползвате така или иначе. Но съществуват и дузини други варианти, ако не желаете да инсталирате и поддържате собствен Git сървър.

Обобщение

Разполагате с няколко опции за създаване и ползване на отдалечени Git хранилища, чрез които да сътрудничите заедно с екипа разработчици.

Пускането на собствен сървър ви дава висока степен на контрол и ви позволява да го разположите зад собствена защитна стена, но пък изисква известно време за настройка и регулярна поддръжка. Ако разположите кода си на хостван сървър, тогава първоначалната настройка и поддръжката не изискват никакви усилия, но от друга страна - не всички организации са съгласни на такъв един подход.

Би следвало да е лесно да изберете кое решение (или комбинация от решения) е подходящо за вашата организация.

Git в разпределена среда

Сега, когато вече имате настроено отдалечено Git хранилище като обща точка за всички разработчици по даден проект и сте достатъчно сигурни в познанията за основните Git команди в процеса на локална работа, ще се спрем на това как да ползваме някои разпределени похвати за работа.

В тази глава ще видите как се работи с Git в разпределена работна среда като сътрудник и интегратор. Това включва инструкции за това как да допринасяте успешно със свой код в общ проект по възможно най-лесния начин за вас и за автора на проекта, а също така и как самите вие да поддържате успешно проект, по който работят множество разработчици.

Разпределени работни процеси

За разлика от централизираните Version Control Systems (CVCSs), разпределената натура на Git позволява по-голяма гъвкавост по отношение на това как разработчиците сътрудничат в проектите. В централизираните системи всеки разработчик е нещо като възел работещ повече или по-малко с един централен хъб. В Git обаче, всеки разработчик може потенциално да бъде и хъб и възел – това значи, че може както да сътрудничи в други хранилища, така и да поддържа свое собствено публично такова, на което негови колеги да базират работата си и да сътрудничат. Това отваря доста възможности за структуриране на работните процеси за проекта и/или екипа, така че ще разгледаме някои познати парадигми, които съществуват благодарение на тази гъвкавост. Ще погледнем силните и слаби страни на всеки дизайн, а вие може да изберете един от тях или да смесвате възможности от различни такива.

Централизиран работен процес

В централизираните системи съществува единичен модел за съвместна работа – централизираният работен процес. Един централен хъб, *хранилище*, може да приема код и всеки трябва да синхронизира работата си с него. Множеството разработчици са възлите, консуматорите в този хъб и те трябва да се синхронизират с неговия статус.

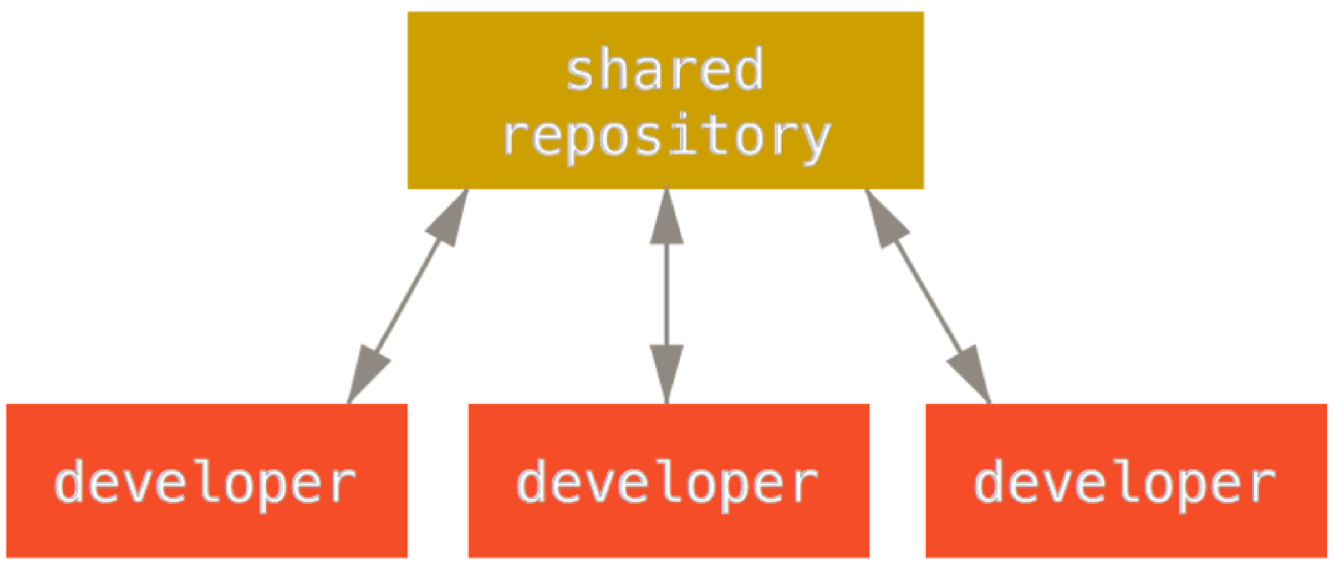


Figure 53. Централизиран работен процес

Това ще рече, че ако двама души клонират от централното хранилище и направят промени, то само първият от тях, който изпрати обратно промените си, ще може да направи това без проблем. Закъснелият разработчик ще трябва първо да слее при себе си работата на първия си колега, преди да публикува своите промени - така първите направени промени не се презаписват. Тази концепция работи както при Git, така и при Subversion (или всяка друга CVCS).

Ако вече се чувствате в свои води с този стил на работа във вашата компания или екип, можете лесно да продължите да го следвате и с Git. Просто направете единично хранилище и дайте на всеки от екипа си push достъп; Git няма да позволи на колегите да се презаписват един друг.

Да кажем, че John и Jessica започват да работят едновременно. John завършва промените си и ги изпраща към сървъра. След това Jessica опитва да публикува своите, но сървърът отказва това. Тя ще бъде уведомена, че се опитва да публикува non-fast-forward промени и че няма да може да го направи докато първо не изтегли и слее работата на John. Този работен процес е популярен, защото следва познати на много хора парадигми на работа.

Освен това не е ограничен до малки екипи. С модела за разклоняване на Git е възможно стотици разработчици успешно да работят едновременно по единичен проект през множество различни клонове код.

Integration-Manager работен процес

Понеже Git позволява да имате множество отдалечени хранилища, възможно е да имате работен процес, при който всеки разработчик има достъп за писане до собственото си публично хранилище и достъп само за четене до всички останали. Този сценарий често включва canonical хранилище, което представлява "официалния" проект. За да сътрудничите в него, вие създавате собствено публично негово копие и публикувате промените си в него. След това, можете да изпратите заявка до собственика на canonical проекта за интегриране на вашите промени. Тогава този автор може да добави вашето хранилище като свое отдалечено такова, да тества локално промените ви, да ги слее в собствен клон и накрая да ги публикува в публичното хранилище така че да са достъпни за

всички. Процесът работи така (вижте Integration-manager работен процес):

1. Поддържащият проекта изпраща промени в публично хранилище.
2. Сътрудник клонира хранилището и прави собствени промени по него.
3. Сътрудникът публикува промените си в собственото си публично копие.
4. Сътрудникът изпраща на поддържащия проекта съобщение със заявка за интегриране на промените.
5. Поддържащият проекта добавя хранилището на сътрудника като отдалечено, издърпва и слива промените локално при себе си.
6. Поддържащият проекта публикува слетите промени обратно в главното хранилище.

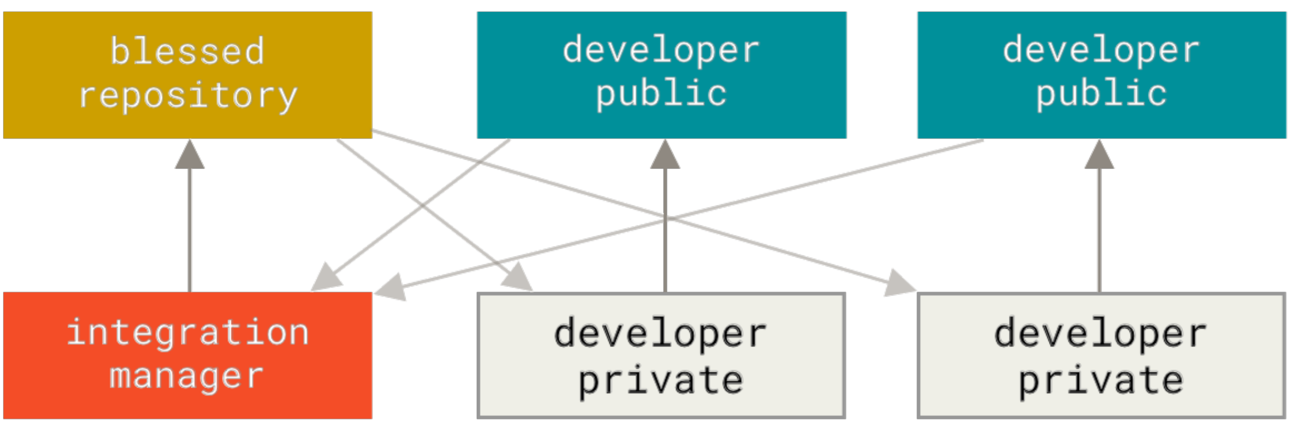


Figure 54. Integration-manager работен процес

Това е доста разпространен начин на работа в хъб-базираните платформи като GitHub или Gitlab, където е лесно да клонирате проект и да публикувате промените си в него така, че всички да ги виждат. Едно от най-съществените предимства на този подход е, че позволява да продължите работата си, а поддържащият централното хранилище може да интегрира промените ви по всяко време. Сътрудниците не е нужно да чакат промените им да бъдат интегрирани – всяка страна може да работи със собствени темпове.

Dictator and Lieutenants работен процес

Това е вариант на работа с множество хранилища. Обикновено се използва в проекти от много голям мащаб, със стотици сътрудници; един пример за такъв е ядрото на Linux. Различни интегриращи мениджъри се грижат за определени части от хранилището; те се наричат *лейтенанти*. Всеки от лейтенантите има един интегриращ мениджър, известен като благосклонен диктатор (benevolent dictator). Този диктатор публикува код от собствена директория към референтно хранилище, от което всички сътрудници трябва да теглят. Процесът работи така (вижте Benevolent dictator работен процес):

1. Обикновените разработчици работят по своя topic клон и пребазират работата си върху `master`. `master` клонът е този от референтното хранилище, към което диктаторът публикува.
2. Лейтенантите сливат topic клоновете на разработчиците в своя `master` клон.
3. Диктаторът слива `master` клоновете на лейтенантите в своя `master` клон.

4. Накрая, диктаторът изпраща този `master` клон към референтното хранилище, така че разработчиците могат да се пребазират по него.

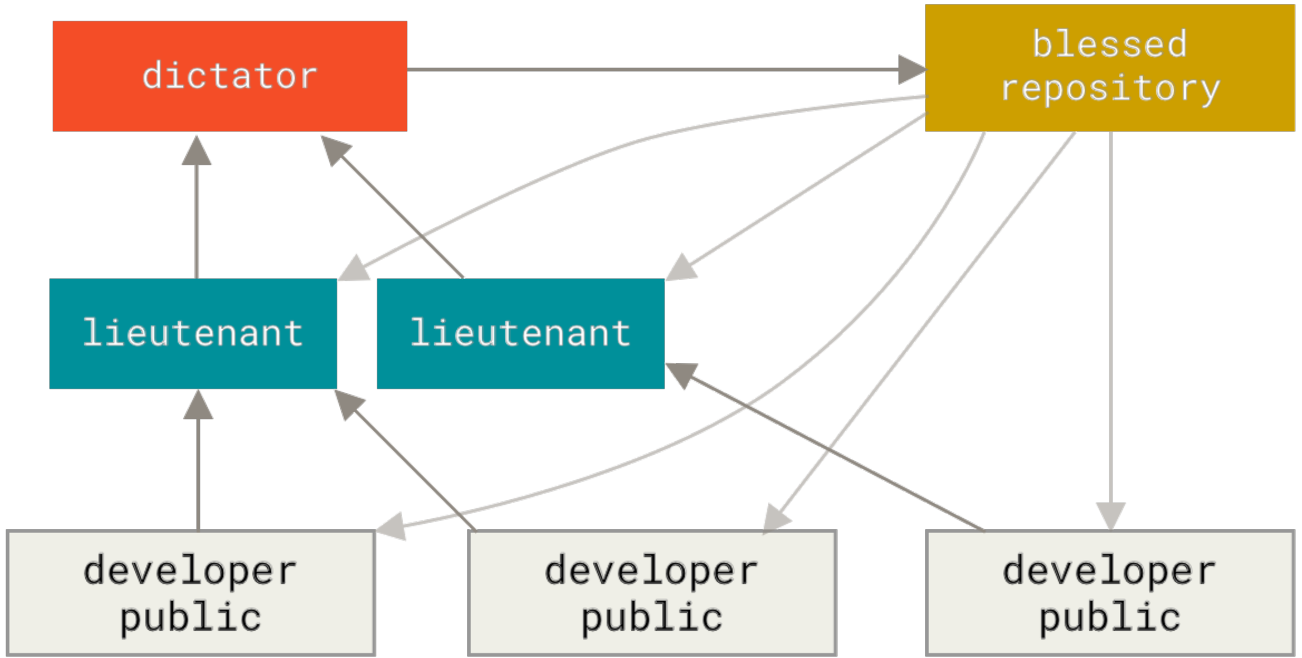


Figure 55. Benevolent dictator работен процес

Този процес на работа се ползва рядко, но може да е полезен в много големи проекти или в йерархични работни среди. Той позволява на лидера на проекта (диктатора) да делегира повече работа и да събира големи подмножества от код от много локации преди да ги интегрира.

Patterns for Managing Source Code Branches

Martin Fowler е автор на ръководството "Patterns for Managing Source Code Branches". То обхваща всички популярни работни процеси в Git и обяснява как и кога да се използват. Налична е и глава сравняваща high и low integration frequencies.

https://martinfowler.com/articles/branching-patterns.html

Обобщение

Това са част от популярните работни процеси, които могат да се реализират с разпределена система като Git, но може да видите, че са възможни голям брой варианти, които да съответстват на вашата специфична работна среда. Сега, след като (надяваме се) можете да изберете варианта, който е най-подходящ за вас, ще разгледаме някои по-специфични примери за това как да изпълняваме основните роли, които съставляват тези работни процеси. В следващата секция ще посочим няколко основни правила за това как да сътрудничим в проект.

Как да сътрудничим в проект

Основната трудност с обяснението на това как се допринася към проект са големия брой варианти как да го направите. Понеже Git е много гъвкав, хората могат (и го правят) да си вършат работата по различни начини – затова е проблематично да се даде съвет как да постъпите – всеки проект си има специфики. Сред факторите, които влияят са броя разработчици, избрания вид работен процес, вашите права за достъп до хранилището и вероятно външния метод за съвместна работа.

Първата неизвестна е броят разработчици – колко са много активните колеги и колко често сътрудничат? В много случаи ще имате двама или трима програмисти с малко къмити на ден, или дори и по-малко при не толкова интензивни проекти. В големи компании или проекти обаче, могат да съществуват хиляди сътрудници със стотици или хиляди къмити на ден. Това е важно, защото с увеличаването на броя програмисти, ще срещате повече проблеми с това да се уверите, че кодът се прилага чисто или може лесно да се слива. Промените, които предлагате могат да се окажат остарели или несъвместими с друга работа, която е била слята докато сте работили или докато сте чакали вашия код да бъде одобрен. Възниква въпросът как да поддържате кода си актуален и къмитите си валидни?

Следващият фактор е избраният вид работен процес. Дали той е централизиран, при който всеки разработчик има еднакъв достъп за писане до основното хранилище? Дали проектът има поддържащ потребител или интегриращ мениджър, които да проверяват всички пачове? Дали всички пачове се разглеждат и одобряват колективно? Дали вие участвате в този процес? Съществува ли lieutenant-система и трябва ли първо да изпращате работата си на вашия лейтенант?

Следват правата ви на достъп. Последователността на работа, когато трябва да участвате в проект е доста различна в зависимост от това дали имате права за писане или не. Ако нямате права за писане, как този проект предпочита да приема външна помощ? Дали въобще съществува политика за това? Какво количество работа изпращате всеки път? Колко често правите това?

Всички тези въпроси могат да се отразят на начина, по който вие допринасяте към даден проект и какви работни процеси са предпочитани или достъпни за вас. Ще разгледаме различните аспекти в серия от примери, започвайки от прости към по-сложни и в края би следвало да си изградите представа какъв похват ще ви е необходим в различните случаи от реалната практика.

Упътвания за къмитване

Преди да се фокусираме върху конкретни примери, едно малко отклонение касаещо къмит съобщенията. Да имате добра насока за създаване на къмити и да се придържате към нея ще направи работата ви с Git и сътрудничеството с колегите много по-лесни. Самият проект Git предоставя документ формулиращ множество добри съвети за създаване на къмити, от които да изпращате пачове — можете да го прочетете в сорс кода на Git, във файла Documentation/SubmittingPatches`.

Първо, вашите промени не трябва да съдържат никакви whitespace грешки. Git осигурява лесен начин да проверите това — преди да къмитнете, изпълнете командата `git diff`

`--check`, която намира възможните whitespace грешки и ви ги показва.

```
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Figure 56. Изход от `git diff --check`

Ако я изпълните преди да къмитнете, можете да разберете дали не вкарвате в проекта излишни интервали, които да отегчат останалите ви колеги.

След това, опитайте се да направите от всеки един къмит логически отделена, самостоятелна и специфична за даден проблем промяна. Ако можете, опитайте да правите промените си по-компактни — не програмирайте цял уикенд по пет различни задачи и после да ги изпращате като един масивен къмит в понеделник. Дори ако през уикенда не сте къмитвали, използвайте индексната област в понеделник за да разделите работата си на няколко къмита, всеки от които обслужва конкретен решен проблем и съдържа съответното подходящо съобщение. Ако някои от промените засягат един и същи файл, опитайте да използвате `git add --patch` за да индексирате файловете частично (показано в подробности в Интерактивно индексиране). В края на краищата, snapshot-тът на края на клона ви ще е идентичен независимо дали сте направили един или пет къмита, така че пробвайте да улесните живота на вашите колеги, когато започнат да разглеждат работата ви.

Този подход също така ви позволява по-късно да премахнете или коригирате само част от промените си, ако се наложи. Манипулация на историята описва няколко полезни Git трика за презапис на историята и интерактивно индексиране на файлове — използвайте тези инструменти като помощ за поддържане на чиста и разбираема история преди да изпратите работата си към някой друг.

Последното нещо, което също не трябва да се пренебрегва, е къмит съобщението. Изработете си навик да създавате качествени къмит съобщения и ще работите с Git и колегите си много по-приятно. Като основно правило, съобщенията ви трябва да започват с един ред не по-дълъг от 50 символа, описващ стегнато промените, следван от празен ред, след което може да се разположат по-дългите обяснения. Git проектът например изисква подробното описание да включва мотивите ви за дадената промяна и да обяснява разликите с оригиналния вариант на кода — и това е добро правило, което може да

следвате. Пишете съобщенията си в императивна форма: "Fix bug" а не "Fixed bug" или "Fixes bug" Тук има един шаблон, написан отначало от Tim Pope, който леко преработихме:

```
Кратко (50 или по-малко символа), изписано с големи букви описание на
промените

По-детайлно описание, ако е нужно. Пренасяйте го на около 72 символа.
В някои случаи, първият ред се третира като subject на
email и останалото от текста като тяло. Празният ред разделящ двете
е много важен (освен ако не пропускате тялото
изцяло); инструменти като rebase може да се объркат
ако пуснете двете заедно.

Напишете съобщението си в императивна форма: "Fix bug" а не
"Fixed bug" или "Fixes bug". Тази конвенция съответства на къмит
съобщенията генерирани от команди като git merge и git revert.

Следващите абзаци се разделят с празни редове.

  - точките (Bullet) също са OK

  - Обикновено за bullet се използва звездичка или тире следвани
    от един интервал, с празни редове помежду им, но правилата тук варират

  - Използвайте hanging indent
```

Ако всички ваши къмит съобщения следват този модел, нещата между вас и колегите ви разработчици ще вървят по-гладко. Git проектът има добре форматирани къмит съобщения — пробвайте да пуснете `git log --no-merges` в хранилището и ще видите колко добре форматирана е историята на проекта.

Правете каквото казваме, а не каквото правим ние.

Много от примерите в тази книга съвсем не съдържат добре форматирани къмит съобщения; ние използваме `-m` параметъра към `git commit` за по-кратко.

Накратко, следвайте правилото за добра практика, а не гледайте какво сме направили ние.

Малък частен екип

Най-простата схема, на която може да попаднете е частен проект с един или двама разработчици. “Частен,” в този смисъл, означава със затворен код — недостъпен за външния свят. Вие всички имате достъп за писане до хранилището.

В такава среда, можете да следвате работен процес подобен на този, който бихте ползвали в Subversion или друга централизирана система. Може все още да използвате предимствата на неща като офлайн къмитване и съвсем прости разклонявания и сливания, но работният

процес може да е много подобен; основната разлика е, че сливанията се случват от страна на клиента, вместо на сървъра по време на къмита. Нека видим как би могло да изглежда това, когато двама разработчика започнат съвместна работа със споделено хранилище. Първият програмист, John, клонира хранилището, прави промяна и къмитва локално. Протоколните съобщения са заменени с `...` в тези примери за да ги съкратим.

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

Вторият разработчик, Jessica, прави същото нещо — клонира хранилището и къмитва промяна:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Add reset task'
[master fbff5bc] Add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Сега, Jessica публикува работата си на сървъра и това работи безпроблемно:

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

Последният ред от изхода отгоре показва полезно съобщение за резултата от push операцията. Основният формат е `<oldref>..<newref> fromref -> toref`, където `oldref` означава референция към предишния къмит, `newref` - към текущия, `fromref` е името на локалния клон, който се изпраща, и `toref` - на отдалечения, който се обновява. Ще виждате подобен изход по-натам в дискусиите, така че да имате представа какво означава това ще е полезно в осмислянето на различните статуси на хранилищата.

Повече подробности има в документацията на командата git-push.

Продължаваме с този пример. Скоро след това, John прави някакви промени, къмитва ги в

локалното си хранилище и се опитва да ги изпрати в същия сървър:

```
# 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'
```

В този случай, опитът му завършва с неуспех, защото Jessica е изпратила *нейните* промени по-рано. Това е особено важно да се разбере, ако сте ползвали Subversion, защото вероятно забелязвате, че двамата разработчици не са редактирали един и същи файл. Въпреки че Subversion ще направи автоматично сливане на сървъра в случаи като този (променени различни файлове), при Git вие трябва *първо* да слеете къмитите локално. С други думи, John трябва първо да изтегли upstream промените на Jessica, да ги слее в локалното си хранилище и едва след това ще може да изпраща към сървъра.

Като първа стъпка, John изтегля работата на Jessica (командата отдолу само *изтегля* последната работа на Jessica, не я слива автоматично):

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

В този момент, локалното хранилище на John изглежда подобно на това:

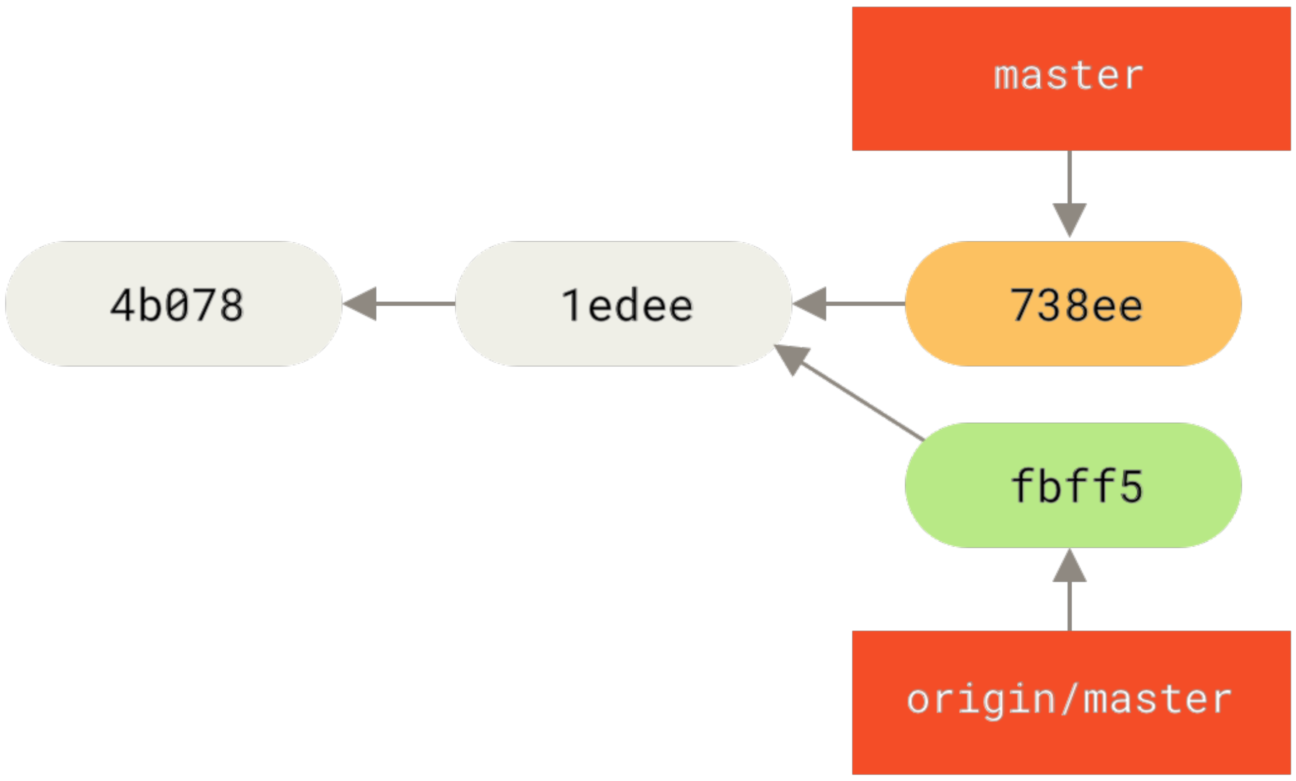


Figure 57. Разклонена история на John

Следва командата за сливане в локалното хранилище:

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

След като това локално сливане мине гладко, обновената история на John ще изглежда така:

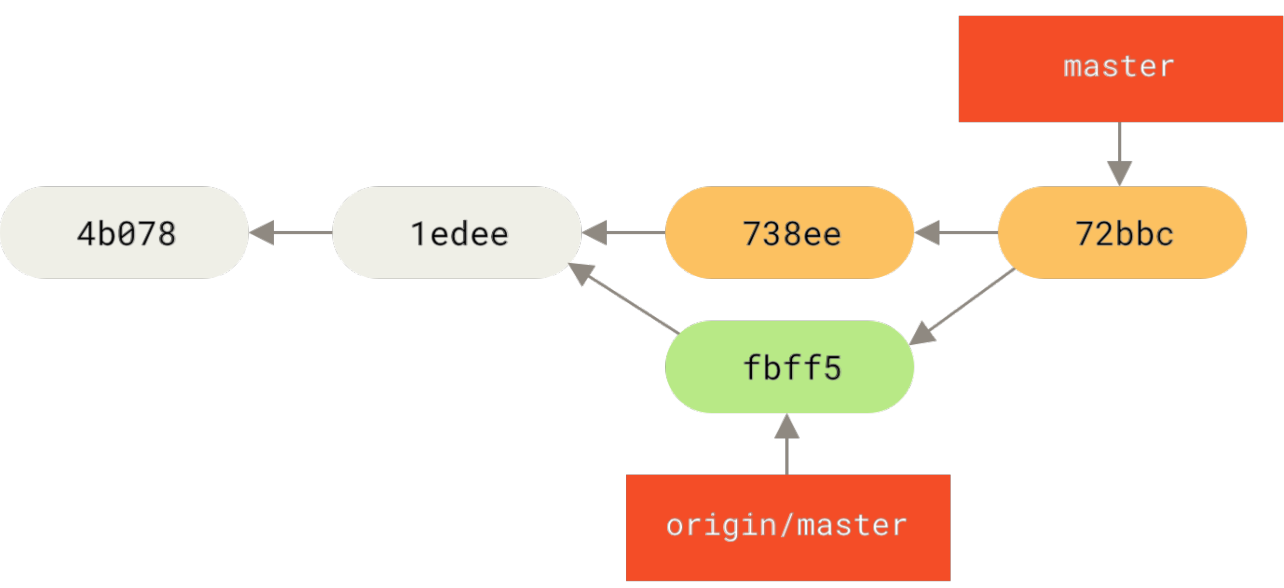


Figure 58. Хранилището на John след сливането на `origin/master`

В този момент, John може да поиска да провери дали новия код не засяга по някакъв начин неговата работа и след като всичко е нормално, може да изпрати всичко към сървъра:

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

В края, историята на къмитите на John ще изглежда така:

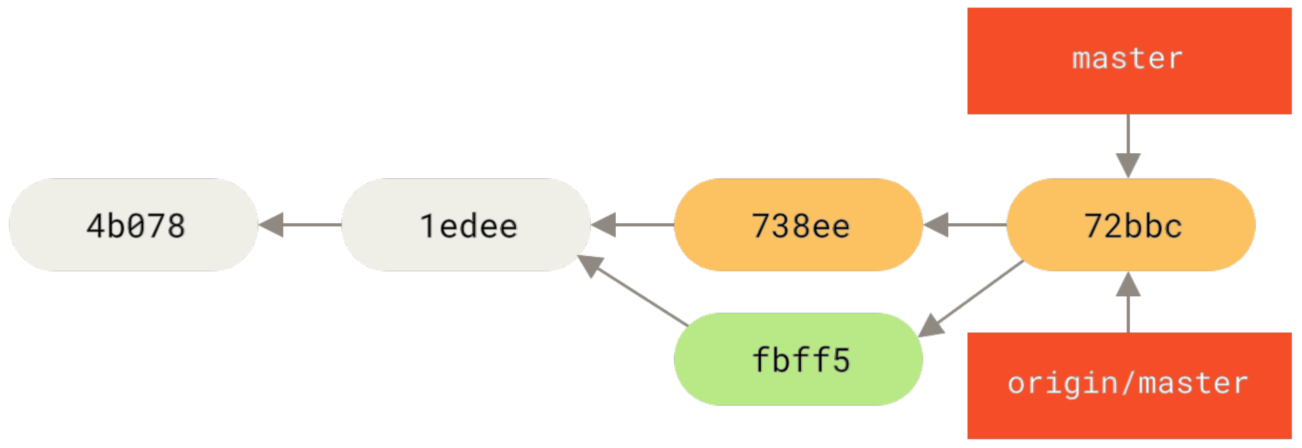


Figure 59. Историята на John след изпращане към `origin` сървъра

Междувременно, Jessica създава нов topic клон наречен `issue54` и прави три къмита в него. Тя все още не е издърпала промените на John, така че историята ѝ изглежда така:

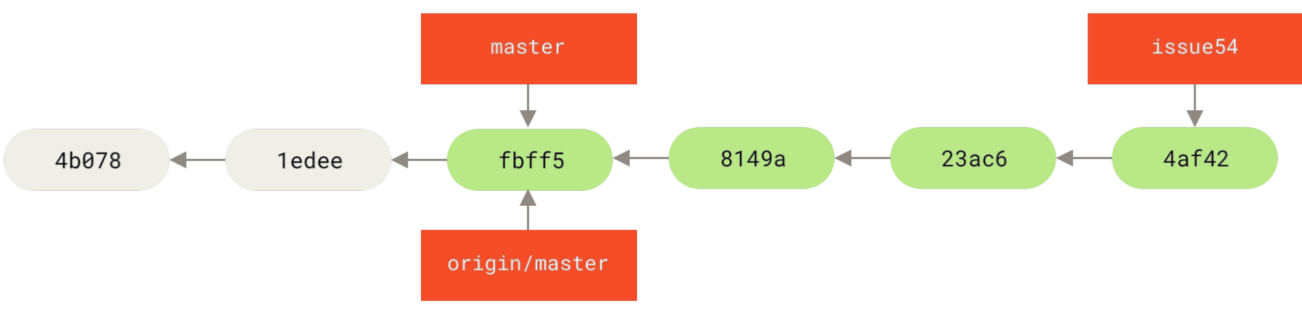


Figure 60. Topic клонът на Jessica

Внезапно Jessica научава, че John е публикувал някакви нови промени и иска да ги погледне, така че може да издърпа от сървъра:

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

Това ще изтегли промените на John и историята на Jessica изглежда така:

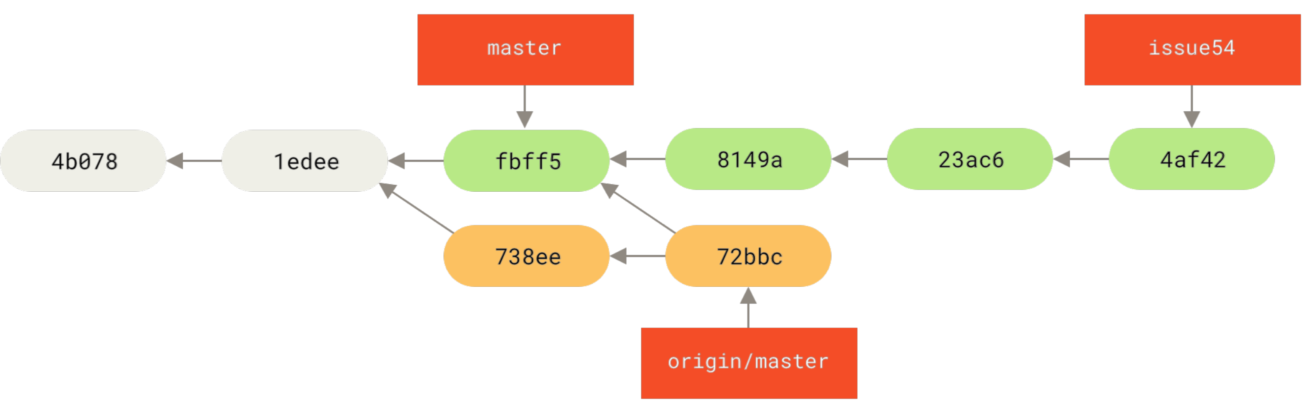


Figure 61. Историята на Jessica след изтегляне на промените на John

Jessica решава, че нейният topic клон вече е готов, но иска да знае коя част от работата на John трябва да слее със своята, така че да може да публикува. За целта тя изпълнява командата `git log`:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   Remove invalid default value
```

Синтаксисът `issue54..origin/master` е филтър, който указва на Git да покаже само тези къмити, които са налични във втория клон (в случая `origin/master`), но липсват в първия (`issue54`). Ще разгледаме в подробности този синтаксис в Обхвати от къмити.

От горния изход можем да видим, че съществува един къмит направен от John, който Jessica не е сляла в локалното си копие. Ако тя слее `origin/master`, това ще е единственият къмит, който би променил локалната ѝ работа.

Сега тя може да слее своя topic клон в `master` клона си, да слее работата на John (`origin/master`) пак в него и накрая да публикува всички промени към сървъра.

Първо (след като е къмитнала всичко в `issue54` клона), Jessica превключва обратно към `master` клона си:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Jessica може да слее първо `origin/master` или `issue54` — и двата са upstream, така че редът е без значение. Крайният snapshot трябва да е идентичен, само историята ще се различава. Тя решава да слее първо `issue54` клона:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Не възникват проблеми, това е просто fast-forward сливане. Jessica сега завършва процеса по локално сливане като вмъква и работата на John от клона `orgin/master`:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Всичко минава чисто и историята на Jessica сега изглежда така:

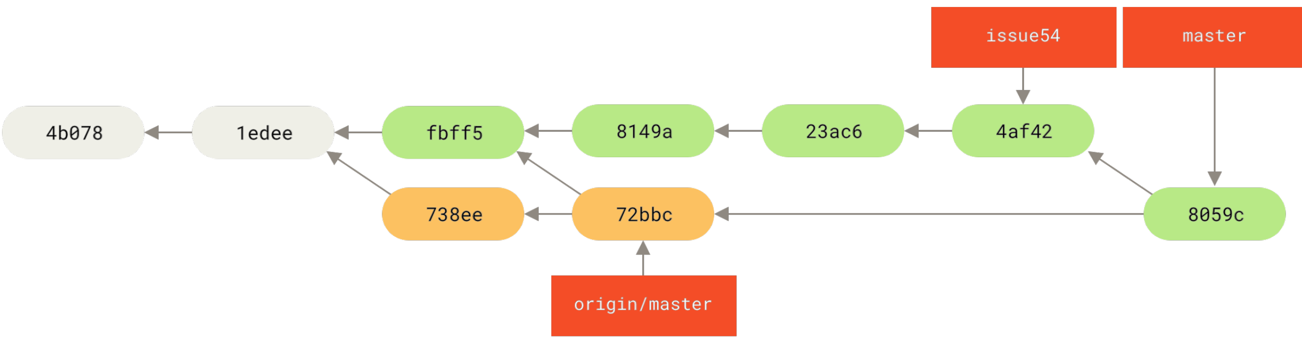


Figure 62. Историята на Jessica след сливане на промените от John

Сега `origin/master` е достъпен от `master` клона на Jessica, така че тя може успешно да публикува промените (приемаме, че междувременно John не е изпращал нови такива):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Сега всеки разработчик е къмитнал по няколко пъти и успешно е слял работата на колегата си.

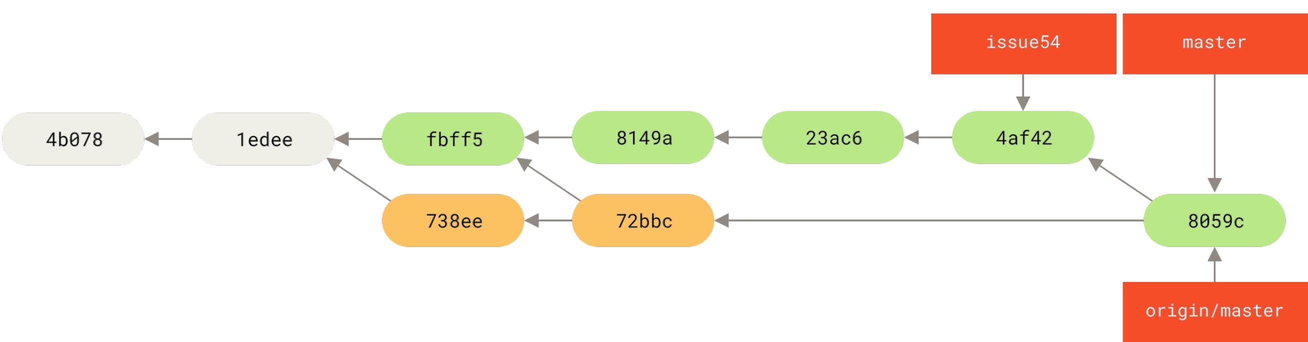


Figure 63. Историята на Jessica след изпращането на всички промени към сървъра

Това е един от най-простите процеси на работа. Вие работите известно време (обикновено в topic клон) и сливате работата си в `master` клона, когато е свършена. Когато желаете да споделите тази работа, вие изтегляте и сливате вашия `master` с `origin/master`, ако той е променен, накрая публикувате `master` клона си в сървъра. Общата последователност изглежда подобно:

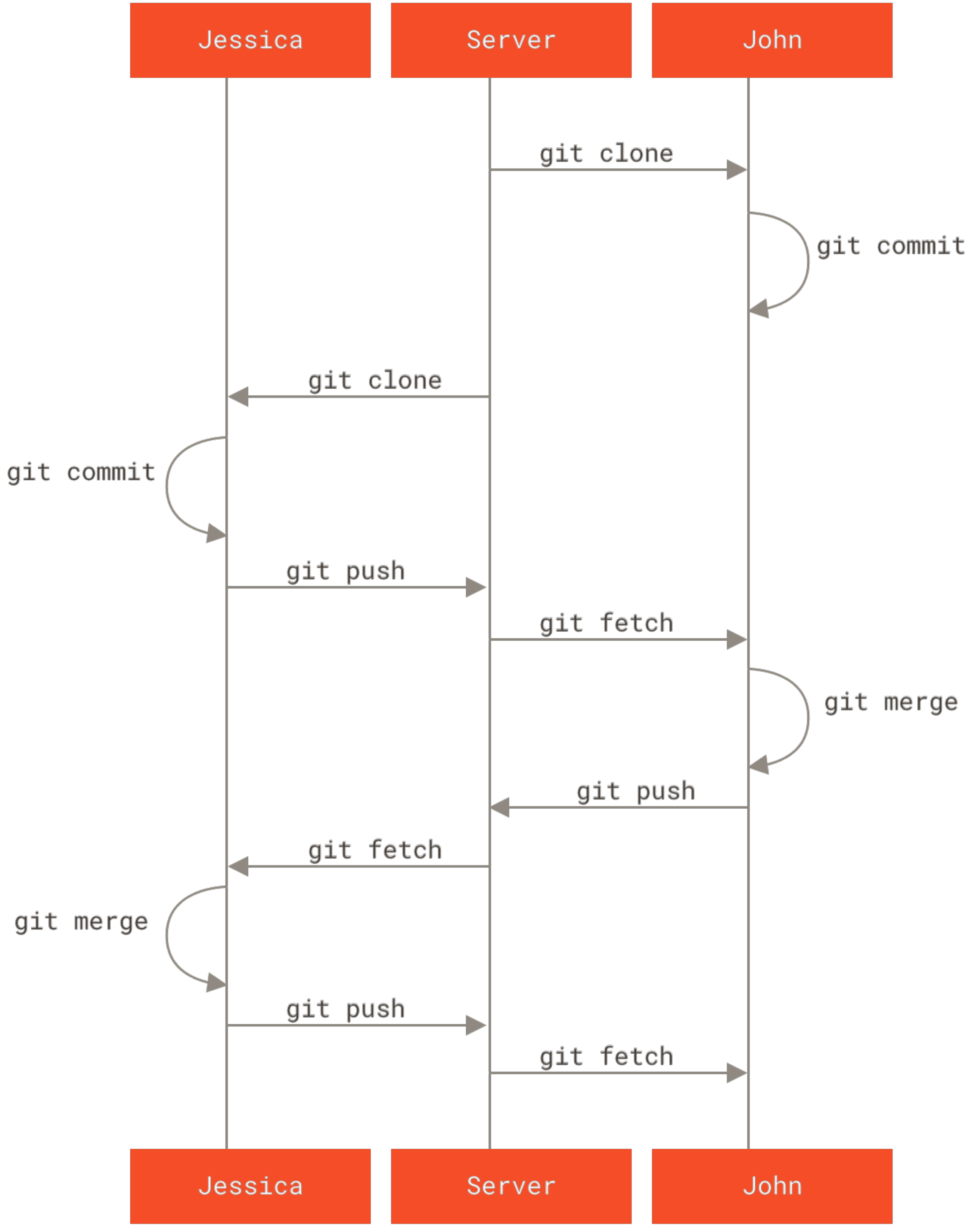


Figure 64. Обичайна последователност от събития при прост работен процес с няколко разработчици в Git

Работа в управляван екип

В този сценарий, ще разгледаме как да допринасяме в по-голям частен екип от разработчици. Ще разберете как да работите в обкръжение, в което малки групи си

сътрудничат по определени функционалности, след което резултатът от работата на тези групи се интегрира в проекта от трети човек.

Нека приемем, че John и Jessica работят по функционалността “feature A”, докато Jessica и друг програмист, Josie, работят по друга опция, “feature B”. В този случай, компанията използва работен процес от тип integration-manager, при който работата на индивидуалните групи се интегрира само от определени хора и `master` клонът на главното хранилище може да се обновява само от тези хора. В сценарий като този, цялата работа се върши в екипни клонове, които интеграторите събират по-късно.

Нека проследим работата на Jessica, защото тя работи по две неща, в паралел с двама различни разработчици. Допускаме, че тя вече има клонирано своето хранилище и решава да започне работа първо по `featureA`. Тя създава нов клон и работи по него тук:

```
# 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(-)
```

В този момент, тя трябва да сподели работата си с John, така че изпраща къмитите от клона `featureA` към сървъра. Тя обаче няма push достъп до `master` клона, само интеграторите имат такъв — ето защо трябва да публикува в друг клон, за да работи съвместно с John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica изпраща на John съобщение за това, че е изпратила своята работа в клон наречен `feature A` и сега той може да го погледне. Докато чака за мнението на John, Jessica решава да започне работа по `feature B` с Josie. За да започне, тя стартира нов feature клон, базирайки го на `master` клона от сървъра:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Сега тя прави няколко къмита в клона `featureB`:

```
$ vim lib/simplegit.rb
$ git commit -am 'Make ls-tree function recursive'
[featureB e5b0fdc] Make 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(-)
```

Хранилището на Jessica сега изглежда така:

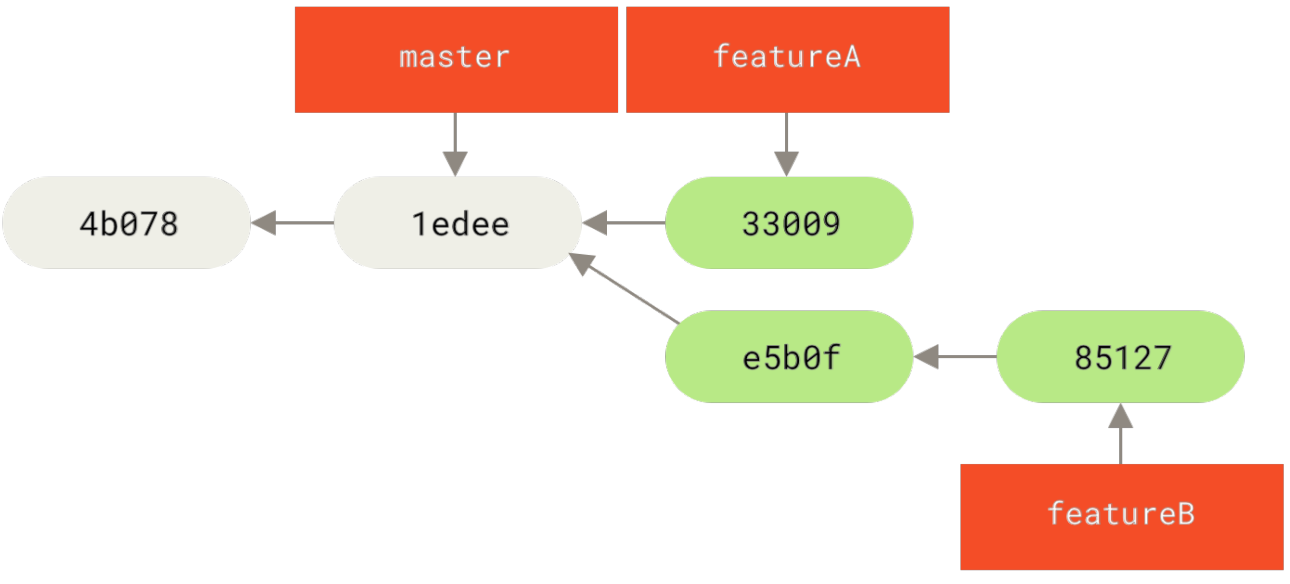


Figure 65. Първоначалните къмити на Jessica

Готова е да изпрати своята работа, но в този момент получава имейл от Josie, в който се съобщава, че на сървъра вече има създаден клон с някаква предварително свършена работа по "featureB". Този клон се казва `featureBee`. Jessica сега трябва първо да слее тези промени със своите собствени преди да може да изпрати към сървъра. Тя изтегля промените на Josie с `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Приемайки, че локално тя е все още в клона `featureB`, тя може да слее работата на Josie с `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

В този момент, тя иска да изпрати цялата си слята работа от “featureB” обратно към сървъра, но не иска да го направи просто изпращайки своя собствен `featureB` клон. Вместо това, понеже Josie вече е направил upstream `featureBee` клон, Jessica ще иска да изпраща промените си именно към *този* клон, затова тя прави така:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

Това се нарича *refspec*, указване на референция. Вижте Refspec спецификации за повече детайли относно Git refspecs и различните възможности, които ви се предоставят с тях. Отбележете също `-u` флага; това е съкращение за `--set-upstream`, който аргумент конфигурира клоновете за лесно дърпане и изпращане.

Внезапно Jessica получава имейл от John, който ѝ казва, че е публикувал някои промени по `featureA` клона, по който си сътрудничат и моли Jessica да ги погледне. Отново, Jessica изпълнява `git fetch` за да изтегли *цялото* ново съдържание от сървъра, включително последните промени на John:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Jessica може да покаже историята на промените на John сравнявайки съдържанието на новоизвлечения `featureA` клон с локалното копие на същия такъв:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    Increase log output to 30 from 25
```

Ако Jessica харесва това, което вижда, тя може да слее промените от John в локалния си `featureA` клон:

```
$ 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(-)
```

Накрая, Jessica може да желае да направи няколко малки корекции към цялото това слято съдържание, така че е свободна да ги реализира, да ги къмитне в локалния `featureA` клон, и да публикува финалния резултат на сървъра:

```
$ git commit -am 'Add small tweak to merged content'
[featureA 774b3ed] Add small tweak to merged content
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Сега историята на къмитите на Jessica ще изглежда по подобен начин:

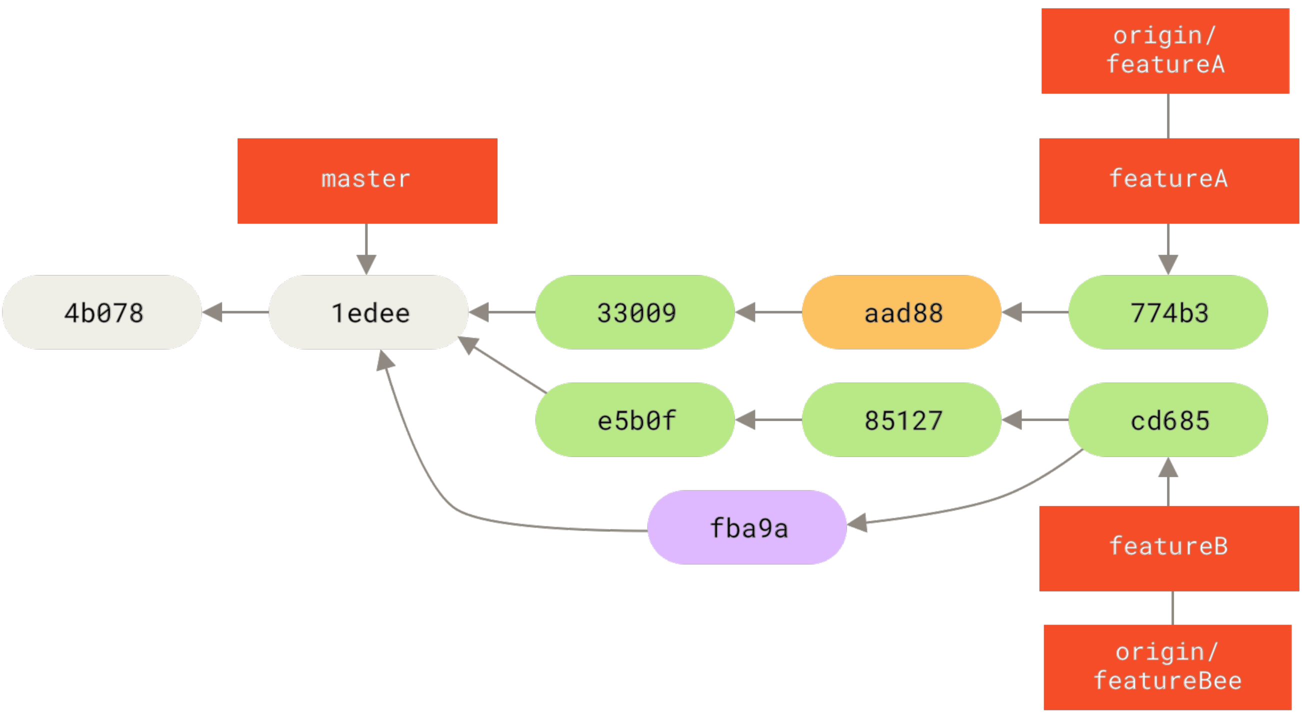


Figure 66. Историята на Jessica след къмитите във feature клона

В даден момент, Jessica, Josie и John информират интеграторите, че `featureA` и `featureBee` клоновете на сървъра са готови за интегриране. След като интеграторите слеят тези клонове в основния проект, едно изтегляне ще предостави новия merge къмит, променяйки историята така:

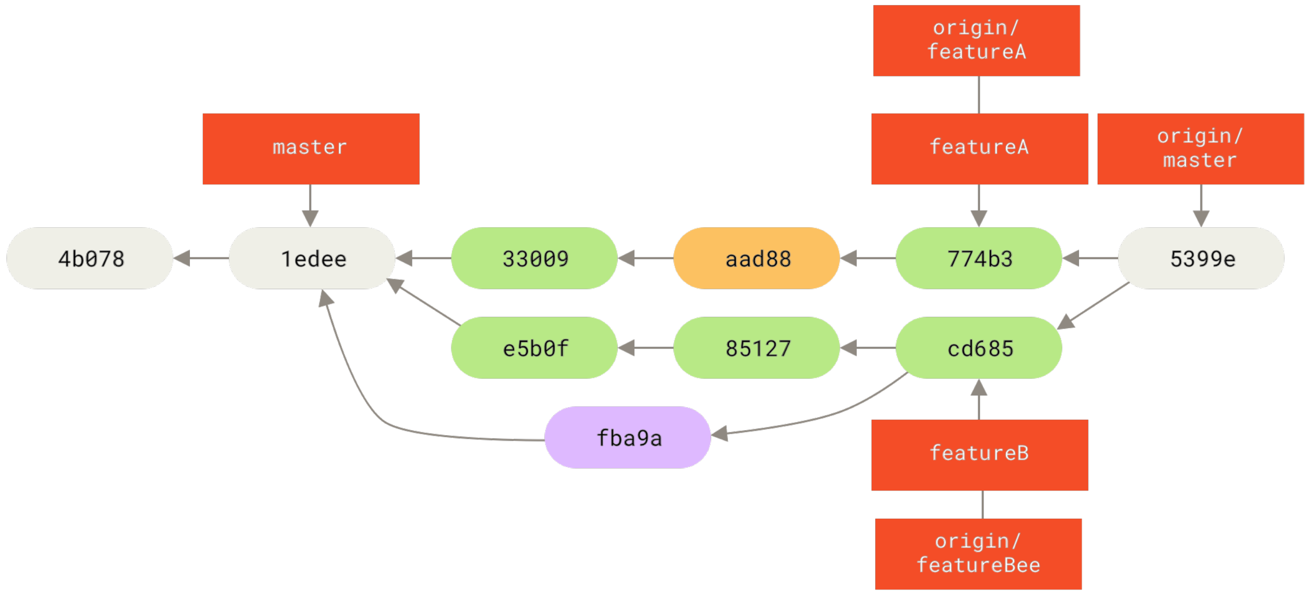


Figure 67. Историята на Jessica след сливането на двата ѝ topic клона

Много екипи преминават към Git поради възможностите за паралелна работа на множество тимове и сливане на различните работни нишки по-късно в процеса. Възможността малки подгрупи от един екип да работят съвместно чрез отдалечени клонове без да е необходимо да въвличат в това целия екип или да възпрепятстват работата на другите, е огромно преимущество с Git. Последователността на работния процес, който описахме, би могла да се илюстрира така:

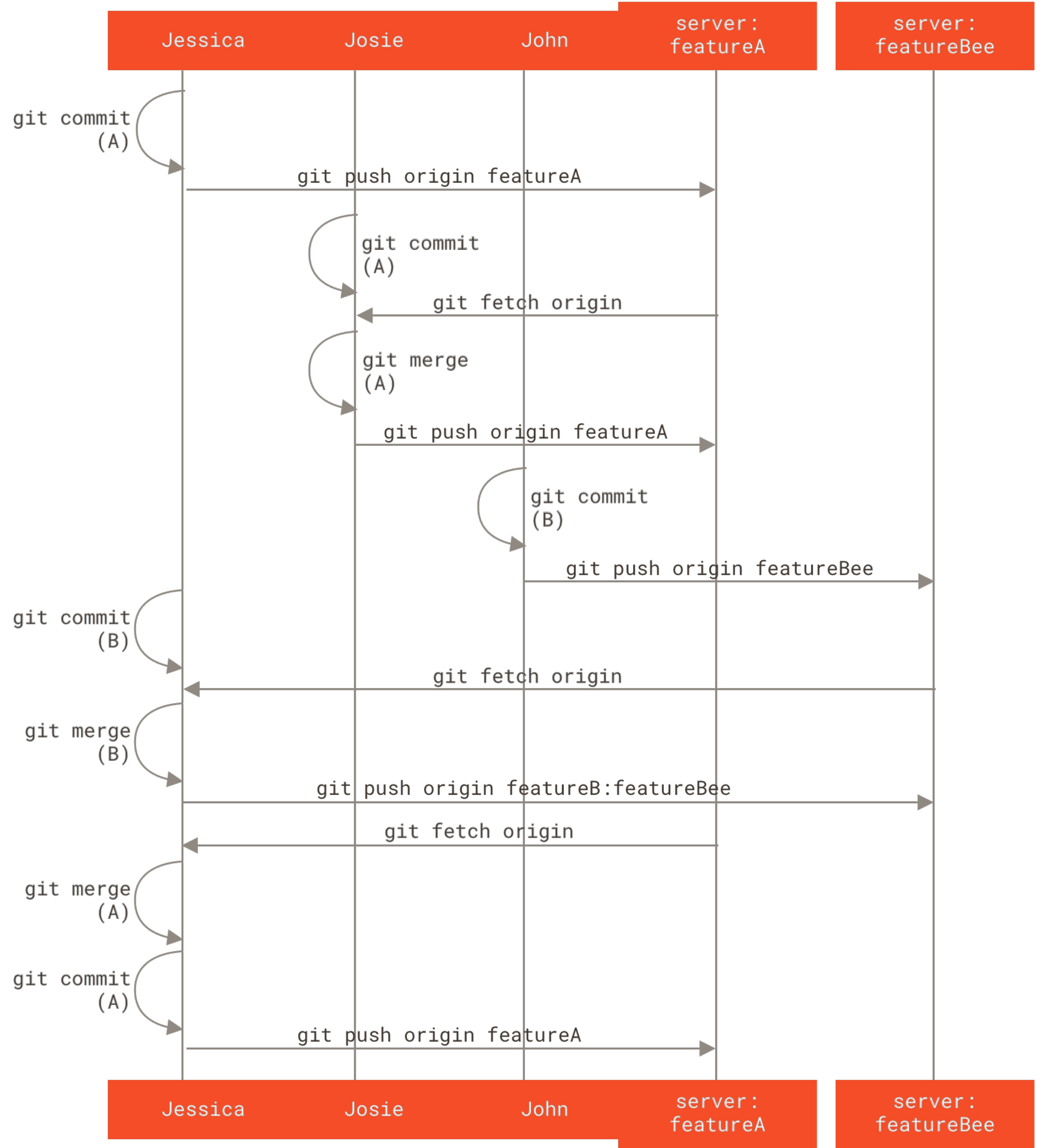


Figure 68. Последователност на действията в managed-team работен процес

Клониран публичен проект

Допринасянето към публични проекти е малко по-различно. Понеже нямате права директно да обновявате клоновете на един такъв проект, по някакъв начин трябва да изпратите работата си до хората, които го поддържат. Този първи пример описва как се сътрудничи чрез клониране на Git хостове, които поддържат easy forking. Много от публичните хостинг платформи (вкл. GitHub, BitBucket, repo.or.cz, и други), както и много от поддържащите дадени проекти, очакват именно такъв подход за външна помощ. Следващата секция се занимава с проекти, които предпочитат да приемат предложените пачове чрез имейл.

Първо, вероятно ще искате да клонирате главното хранилище, да създадете topic клон за пача или серията пачове, които планирате да предложите, и да работите в него. Последователността изглежда така:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
  ... work ...
$ git commit
  ... work ...
$ git commit
```

Може да искате да използвате `rebase -i` за да съберете работата си в единичен къмит, или пък да реорганизирате нещата в множество къмити, така че мениджърът на проекта по-лесно да я разгледа — вижте Манипулация на историята за повече информация за интерактивното пребазиране.

Когато работата ви по клона е завършена и сте готови да я предложите на поддържащите проекта, отидете обратно в неговата страница и използвайте бутона "Fork", така че да си направите свое собствено копие от проекта в сайта, с права за писане в него. След това трябва да добавите URL-а на копието като нова отдалечена референция за локалното ви хранилище, за този пример - нека да я наречем `myfork`:

```
$ git remote add myfork <url>
```

След това трябва да публикувате локалната си работа в това отдалечено хранилище. Най-лесно е да публикувате topic клона, по който работите, вместо първо са го сливате в `master` клона и да публикувате него. Причината за това е, че ако работата не бъде приета или е cherry-picked, няма да има нужда да превъртате обратно вашия `master` клон (`cherry-pick` операцията на Git се разглежда в повече подробности в Rebasing и Cherry-Picking работни процеси). Ако поддържащите проекта `слеят`, `пребазират`, или `cherry-pick`-нат вашата работа, в крайна сметка ще я получите обратно чрез издърпване от тяхното хранилище.

Във всеки случай, можете да публикувате работата си чрез:

```
$ git push -u myfork featureA
```

След като работата ви е била публикувана във вашето клонирано онлайн хранилище, ще трябва да уведомите поддържащите оригиналния проект, че имате промени, които бихте искали да интегрират. Това често се нарича *pull request* и обикновено такава заявка се генерира или директно през уеб сайта — GitHub има собствен "Pull Request" механизъм, който ще видим в GitHub — или пък с командата `git request-pull` чийто изход трябва да изпратите по имейл на мениджъра на проекта ръчно.

Командата `git request-pull` взема за параметри базовия клон, в който искате промените ви

да бъдат интегрирани и адреса на Git хранилището, от което тези промени да бъдат изтеглени, след което изготвя списък с всички промени, които искате да бъдат изтеглени. Например, ако Jessica иска да изпрати на John pull request и е направила два къмита в topic клона, който току що е публикувала, тя може да изпълни следното:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
        Create new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      Add limit to log function
      Increase log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Този изход може да се изпрати към поддържащия проекта — той ще покаже откъде е разклонена работата, обобщава къмитите и указва откъде може да се изтегли новата работа.

В проект, който не поддържате, обикновено е по-лесно да имате клон като `master`, който винаги следи origin/master ` и да вършите работата си в topic клонове, които лесно можете да премахнете, в случай че бъдат отхвърлени. Изолирането на работата в topic клонове също така прави по-лесно пребазирането на вашата работа, ако "върха" на главното хранилище се премести междувременно и къмитите ви вече не се прилагат чисто. Например, ако искате да изпратите втора промяна в проекта, не продължавайте да работите в topic клона, който вече сте изпратили -- започнете от `master клона на главното хранилище:

```
$ git checkout -b featureB origin/master
  ... work ...
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
  ... email generated request pull to maintainer ...
$ git fetch origin
```

Сега, всяка от промените ви се съдържа в силоз — подобно на опашка от пачове — които можете да пренапишете, пребазирате и променяте без различните теми да се преплитат или да зависят една от друга:

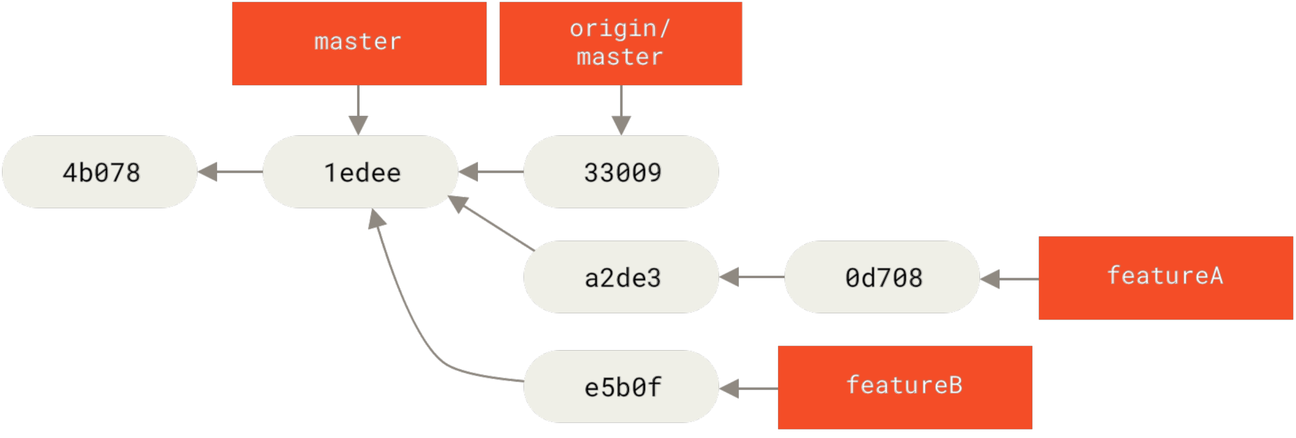


Figure 69. Първоначална история на къмитите с `featureB`

Да кажем, че мениджърът на проекта е интегрирал множество други пачове и е опитал вашия първи клон, който обаче вече не се слива чисто. В този случай, можете да пробвате да пребазирате този клон върху `origin/master`, да оправите конфликтите и да изпратите повторно промените си:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Това ще промени историята ви така:

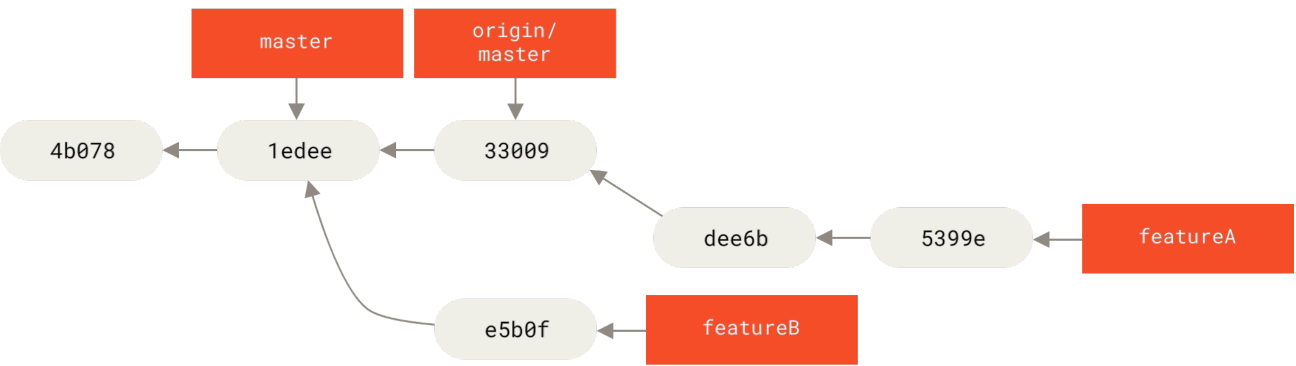


Figure 70. Историята след работа по `featureA`

Понеже пребазирахте клона, трябва да подадете параметъра `-f` към push командата за да можете да замените `featureA` клона на сървъра с къмит, който не е негов наследник. Като алтернатива, можете да публикувате тази нова работа към различен клон в сървъра (например `featureAv2`).

Нека погледнем още един възможен сценарий: поддържащият проекта е погледнал работата във втория ви клон и харесва идеята ви, но би желал да промените малко имплементацията. Вие също ще се възползвате от тази възможност за да преместите работата си така, че да е базирана на текущия `master` клон на проекта. Стартирате нов клон базиран на текущия `origin/master`, премествате `featureB` промените в него, решавате евентуалните конфликти, променяте имплементацията и го публикувате като нов клон:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
  ... change implementation ...
$ git commit
$ git push myfork featureBv2
```

Опцията `--squash` взема цялата работа от слетия клон и я събира в едно множество промени, в резултат на което статусът на хранилището ще изглежда така сякаш е станало реално сливане, без в действителност да се прави merge къмит. Това означава, че бъдещият ви къмит ще има само един родител и ви позволява да въведете всички промени от друг клон и да направите още такива преди да запишете новия къмит. Също така, опцията `--no-commit` може да е полезна за отлагане на merge къмита в случай на нормален процес по сливане.

В този момент, можете да уведомите поддържащия проекта, че сте направили поисканите промени и те могат да се намерят в клона `featureBv2`.

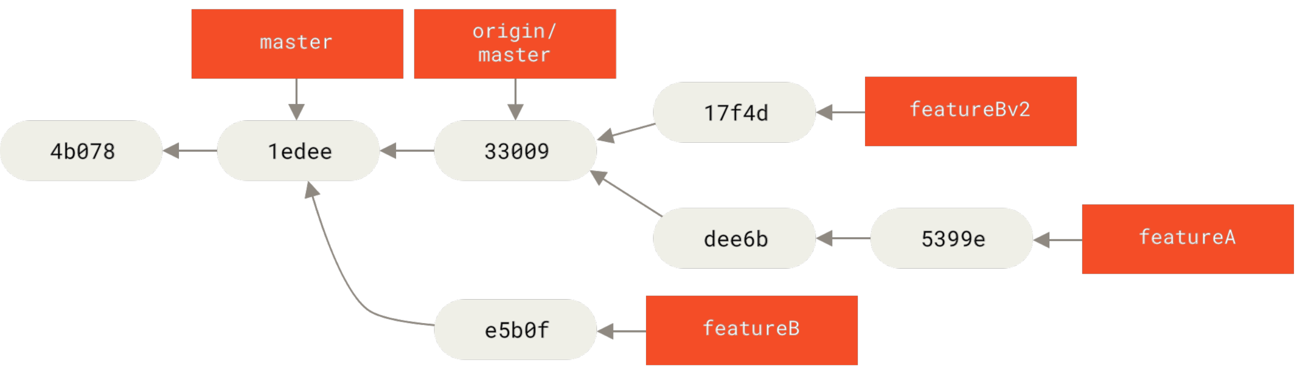


Figure 71. Историята след работата по `featureBv2`.

Публичен проект с комуникация по имейл

Много проекти имат установени процедури за приемане на пачове — ще трябва да проверите специфичните изисквания според вашия случай. Понеже все още съществуват множество по-стари, големи по мащаб проекти, които приемат пачовете през мейлинг лист за разработчици, ще дадем и един пример за такъв случай.

Работният процес е подобен на предишния — вие създавате topic клонове за всяка серия от пачове, по която работите. Разликата е в това как ги изпращате към проекта. Вместо да клонирате проекта и да публикувате във вашата версия, в която имате права за писане, вие генерирате имейл версии на всяка серия къмити и ги изпращате към мейлинг лист:

```
$ git checkout -b topicA
  ... work ...
$ git commit
  ... work ...
$ git commit
```

Сега имате два къмита, които искате да пратите. Използвате `git format-patch` командата, за да генерирате mbox-форматирани файлове, които можете да пратите към списъка — тя превръща всеки къмит в имейл съобщение със subject отговарящ на първия ред от къмит съобщението и останалата част от съобщението плюс пача - като тяло на имейла. Хубавото на това е, че прилагането на пач от имейл съобщение генерирано с тази команда запазва коректно цялата информация за къмита.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
```

Командата `format-patch` отпечатва имената на пач-файловете, които създава. Параметърът `-M` казва на Git да търси за преименувания. Файловете изглеждат така:

```
$ 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
```

Можете също да редактирате пач файловете за да добавите допълнителна информация за мейлинг листата, която не желаете да се показва в къмит съобщението. Ако добавите текст между реда `---` и началото на пача (реда `diff --git`), то разработчиците могат да я четат, но това съдържание ще бъде игнорирано от процеса по пачването.

За да изпратите това към листата, можете или да копирате файла в имейл програмата си

или да използвате програма за изпращане на поща от команден ред. Копирането на текст често предизвиква проблеми с форматирането, особено с "по-умните" клиенти, които не запазват новите редове и празните символи коректно. За щастие, Git предоставя инструмент за изпращане на правилно форматирани пачове през IMAP, което може да е по-лесно за вас. Ще покажем как да изпращаме пач през Gmail, тъй като е много популярна услуга, но можете да намерите детайлни инструкции за използването на много други имейл клиенти в края на гореспоменатия файл `Documentation/SubmittingPatches` в сорс кода на Git.

Първо, трябва да подготвите imap секцията във вашия `~/.gitconfig` файл. Можете да настроите всяка стойност отделно чрез серия от `git config` команди или пък да ги въведете ръчно, но независимо от подхода, конфигурационният файл ще изглежда по подобен начин:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Ако IMAP сървърът ви не използва SSL, то последните два реда вероятно няма да са нужни и стойността за хоста ще започва с `imap://` вместо с `imaps://`. Когато това е готово, можете да използвате командата `git imap-send` за да поставите пачовете в Drafts папката на указания IMAP сървър:

```
$ 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
```

В този момент, би трябвало да можете да отворите Drafts папката, да смените To полето към адреса на мейлинг-листата, към която изпращате пача, вероятно да добавите CC поле за поддържащия проекта или човекът, който отговаря за тази секция и да изпратите пощата.

Можете да изпращате и през SMTP сървър. Както и преди, ще трябва да отразите промените в `~/.gitconfig`, в секцията му sendemail, ръчно или с `git config` команди:

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

След това, използвайте командата `git send-email` за да изпратите пачовете си:

```
$ git send-email *.patch
0001-add-limit-to-log-function.patch
0002-increase-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
```

След това Git показва множество лог информация, която изглежда по подобен начин за всеки от пачовете:

```
(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] Add 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
```

За повече информация относно конфигурирането на вашата система за работа с имейли, съвети и трикове, както и sandbox за изпращане на примерни пачове по електронна поща, посетете git-send-email.io.

Обобщение

В тази секция разгледахме различни работни процеси и разликите в работата в малък екип с closed-source проекти и тази при сътрудничество в голям публиен проект. Сега знаете, че трябва да проверявате за white-space грешки преди къмитването и да пишете перфектни къмит съобщения. Научихме как да форматираме пачове и да ги разпращаме по пощата до мейлинг-листа на разработчиците. В контекста на различните работни процеси обърнахме внимание и на особеностите при сливане. Сега сме достатъчно подготвени да сътрудничим в произволен проект.

Следва да погледнем от обратната страна на монетата: поддържането на Git проект. Ще научите как да изпълнявате ролите на benevolent dictator или integration manager.

Управление на проект

След като разгледахме как се допринася в проект по ефективен начин, вероятно ще ви интересува и обратната страна, как да поддържаме собствен такъв. Това може да включва приемане и прилагане на пачове генерирани през `format-patch` и изпратени до вас или пък интегриране на промени в отдалечени клонове за хранилища добавени като remotes към проекта ви. Независимо дали поддържате canonical хранилище или искате да помогнете проверявайки и одобрявайки пачове, трябва да знаете как да приемате работа от колегите ви по начин ясен за тях и удобен за вас във времето.

Работа в Topic клонове

Когато мислите да интегрирате новополучени промени, добра идея е да ги изпробвате в *topic клон* — временен такъв създаден специално за теста. По такъв начин е лесно да поправите пач индивидуално и да го зарежете, ако той не работи, докато имате време да го разгледате по-подробно. Ако създадете клон с име съответстващо на темата на изпратената работа, например `ruby_client` или нещо подобно, можете лесно да го запомните и по-късно да се върнете в него. Поддръжникът на Git проекта например, дори се стреми да използва namespaces за тези имена — като `sc/ruby_client`, където `sc` е съкращение за човека, който е изпратил работата. Както помним, създаването на клон базиран на `master` се прави така:

```
$ git branch sc/ruby_client master
```

Или, ако искате да превключите към него веднага, може да използвате `checkout -b` варианта:

```
$ git checkout -b sc/ruby_client master
```

Сега сте готови да добавите новата работа, която сте получили, в този нов topic клон и да решите дали искате да я слеете в long-term клоновете си.

Прилагане на пачове от Email

Ако сте получили пач по имейл, ще трябва да го приложите в topic клона и да го изпробвате. Това се прави с помощта на една от двете команди `git apply` или `git am`.

Прилагане на пач с apply

Ако сте получили пача от някого, който го е генерирал с `git diff` или някакъв вариант на Unix `diff` инструмента (което не е препоръчително, вижте следващата секция), можете да го приложите с командата `git apply`. Ако сте записали пача в `/tmp/patch-ruby-client.patch`:

```
$ git apply /tmp/patch-ruby-client.patch
```

Това модифицира файловете в работната директория. Командата е почти идентична на

`patch -p1`, въпреки че е по-параноична и приема по-малко fuzzy matches от patch. Тя също така обработва добавянето, изтриването и преименуването на файлове, ако тези процеси са правилно описани в `git diff` формата, нещо което `patch` няма да направи. Освен това `git apply` използва модела “apply all or abort all”, където или всичко се прилага успешно или нищо не се прилага. За разлика от нея, `patch` може да прилага частично patchfiles, оставяйки работната ви директория в странен статус. `git apply` като цяло е много по-консервативна от `patch`. Процесът няма автоматично да ви направи къмит — след като завърши ще трябва сами да индексирате и къмитнете промените.

Можете също да използвате `git apply` за да видите дали пачът ще се приложи коректно преди в действителност да се изпълни, командата е `git apply --check` с името на пача:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Ако няма изход, тогава пачът би следвало да се приложи чисто. В допълнение, командата завършва с код за грешка, ако проверката установи неуспех, така че можете да я използвате и в скриптове, ако желаете.

Прилагане на пач с `am`

Ако работата идва от напреднал Git потребител, който е наясно с `format-patch`, тогава работата ви се улеснява, защото пачът ще съдържа също и информация за автора и къмит съобщение. Ако можете, окуражавайте сътрудниците ви да използват `format-patch` вместо `diff`, когато генерират пачове за вас. Използвайте `git apply` само за legacy пачове и неща като това (`diff` генерирани).

За да приложите пач генериран с `format-patch`, по-добрият вариант е да използвате `git am` (наречена е `am` понеже се използва за "apply a series of patches from a mailbox"). Технически, `git am` е проектирана да чете mbox файл, който е plain-text формат за съхранение на едно или повече имейл съобщения в един файл. Изглежда по подобен начин:

```
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
```

Това е началото на изхода от командата `git format-patch`, който видяхме в предната секция, също валиден mbox имейл формат. Ако някой ви е изпратил пача по пощата коректно с `git send-email` и го запишете в mbox формат, тогава можете да подадете на `git am` въпросния mbox файл и тя ще започне да прилага всички пачове, които намери. Ако имате имейл клиент способен да записва множество съобщения в mbox формат, тогава можете да запишете цялата серия пачове в един файл и след това да пуснете `git am`, която да ги приложи последователно.

Обаче, ако някой е качил пач файл генериран с `git format-patch` към ticketing система или нещо подобно, можете да запишете файла локално и след това да го подадете на `git am` да го приложи:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Може да видите, че той се е приложил чисто и автоматично се създава нов къмит за вас. Информацията за автора е взета от хедърите `From` и `Date` на имейла, а къмит съобщението от `Subject` хедъра и тялото му (частта преди пача). Например, ако този пач беше приложен от mbox примера отгоре, генерираният къмит би изглеждал така:

```
$ 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
```

`Commit` реда индикира кой е човекът, който е приложил пача и времето, когато това е станало. `Author` от своя страна носи информация за създателя на пача и кога е създаден първоначално.

Възможно е обаче пачът да не се прилага чисто. Може главният ви клон да се е разделил твърде много от клона, от който е направен пача или пък самият пач да зависи от друг такъв, който все още не сте приложили. В подобен случай `git am` ще прекъсне и ще ви попита как искате да продължите:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See 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".
```

Командата поставя маркери за конфликт в съответните файлове, както това става при конфликтно сливане или пребазиране. Разрешаването на конфликтите също е аналогично — редактирате файла, индексирате го и след това изпълнявате `git am --resolved` за да продължите цикъла със следващия пач:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

Ако искате Git да се опита да разреши конфликта по малко по-интелигентен начин, може да подадете флага `-3` и Git ще се опита да направи three-way сливане. Тази опция е забранена по подразбиране, защото няма да работи в случай, че пачът рапортува, че е базиран на къмит, който не присъства в хранилището ви. Ако обаче имате този къмит (ако пачът например е базиран на публичен такъв), тогава `-3` опцията е много по-гъвкава в прилагането на конфликтен пач:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See 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.
```

В този случай, без `-3` флага пачът щеше се счита за конфликтен. Но понеже той е подаден, пачът се прилага чисто.

Ако прилагате множество от пачове от mbox, можете също така да пуснете `am` командата в интерактивен режим, при което тя ще спира на всеки пач и ще ви пита дали желаете да го приложи:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
See if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Това е полезно при много пачове, защото можете да видите всеки от тях, ако сте забравили за какво е или да го откажете, ако вече е приложен.

Когато всички пачове са приложени и къмитнати в topic клона, може да изберете дали и кога да ги интегрирате в long-term клон.

Извличане от отделечени клонове

Работата на вашите колеги може и да не идва по имейл. Те могат да имат свои собствени онлайн хранилища, да са извършили много промени и да са ви пратили URL до хранилището и клона, където промените се пазят. В случаи като този, можете да добавите отдалечените хранилища и да сливате локално вместо да пачвате.

Ако Jessica ви съобщи, че има нова функционалност в клона `ruby-client` на хранилището ѝ, можете да тествате бързо добавяйки го като remote референция и извличайки клона локално:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Ако впоследствие тя ви съобщи за нова функционалност в отделен клон, можете директно да направите `fetch` и `checkout`, защото вече отдалеченото хранилище е конфигурирано при вас.

Това е най-полезно, ако работите сравнително често с даден колега. Ако някой иска да ви изпрати само единичен пач и няма намерение за продължително сътрудничество, тогава имейл методът вероятно е по-бързо решение и няма нужда колегата ви да поддържа онлайн хранилище. Освен това едва ли бихте искали да имате стотици remotes, всяко от които допринася само с един-два пача. Обаче, скриптовете и хостнатите публични услуги могат да улеснят това — зависи най-вече от това как разработвате вие и как колегите ви.

Друго предимство на този подход е, че получавате историята на къмитите. Въпреки, че може да имате merge проблеми, вие все пак знаете на коя точка от историята ви е базирана работата на колегата и правилното three-way сливане е по подразбиране вместо да трябва да подавате `-3` и да се надявате, че пачът е бил генериран на публичен къмит, до който имате достъп.

Ако не работите често с определен колега, но въпреки това искате да получавате работата му по този начин, можете да подадете на `git pull` директно адреса на отдалеченото хранилище. По този начин правите one-time pull и не съхранявате URL-а като remote референция:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

Изследване на промените

Сега имате topic клон с новата работа от колега. На този етап може да определите какво бихте искали да правите с нея. Тази секция преглежда няколко команди, които ви помагат да разберете какво точно ще бъде въведено в главния клон, ако решите да направите сливане на topic клона.

Често е полезно да имате представа за всички къмити, които са налични във временния клон, но все още не са в главния. Можете да извадите къмитите в `master` клона добавяйки опцията `--not` преди името му. Това прави същото като формата `master..contrib`, който видяхме по-рано. Например, ако вашият колега ви изпрати два пача и създадете клон `contrib`, в който сте ги приложили, може да изпълните това:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

Ако ви трябват и промените, които въвежда всеки къмит, можете да подадете флага `-p` към `git log` и тя ще ви изведе в допълнение diff информацията за всеки от къмитите.

За да видите пълен diff на това какво ще се случи, ако слеете този topic клон с друг, може да се наложи да използвате малък трик, за да получите коректните резултати. Може би си мислите за това:

```
$ git diff master
```

Командата действително извежда diff, но той може да е заблуждаващ. Ако `master` клонът се е придвижил напред след като сте създали topic клона от него, тогава ще получите изглеждащи странно резултати. Това се случва, защото Git директно сравнява snapshot-а на последния къмит от topic клона, в който сте и snapshot-а на най-новия къмит от `master` клона. Ако например сте добавили ред във файл от `master` след създаването на topic клона, директното сравнение на snapshot-ите ще изглежда така сякаш при сливане topic клона ще изтрие този ред.

Ако `master` е директен родител на topic клона това не е проблем. Но ако двете истории са се разклонили, тогава diff изходът ще показва, че добавяте всичките нови промени от topic и изтривате всичко уникално за `master` клона.

Но това, което наистина искате да видите, са промените добавени в topic — работата, която ще бъде въведена при сливането му в `master`. Начинът да получите този резултат е да накарате Git да сравни последния къмит в topic клона с първия общ предшественик от `master`.

Технически това може да направите като изрично установите кой е този предшественик и след това изпълните diff към него:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

или по-съкратено:

```
$ git diff $(git merge-base contrib master)
```

И понеже нито един от тези два начина не е достатъчно удобен, Git осигурява още едно съкратено изписване за същия резултат: triple-dot синтаксиса. В контекста на командата `git diff`, можете да поставите три точки между имената на два клона и тогава Git ще изпълни `diff` между последния къмит на клона отдясно в израза (contrib) и най-актуалния му общ предшественик от клона вляво (master):

```
$ git diff master...contrib
```

Така получавате само работата, която текущия topic клон въвежда от последния общ родителски къмит в `master`. Това е един много полезен синтаксис и си заслужава да се запомни наизуст.

Интегриране на получена работа

Когато работата в topic клона е готова да бъде изпратена към един или няколко mainline клонове, възниква въпроса как да го направите. Друг въпрос е какъв да е принципния работен процес, който да използвате за поддръжка на проекта си? Има множество варианти и ще разгледаме някои от тях.

Merging работни процеси

Един простичък процес за действие е просто да слеете цялата нова работа директно в `master` клона. В такъв сценарий се предполага, че имате `master` клон, който съдържа стабилен код. Когато получите промени в topic клон и смятате, че той е готов за интегриране или пък когато сте получили и проверили работата на колега, просто сливате нещата в `master` клона, изтривате topic клона и това се повтаря във времето.

Например, ако имаме хранилище с работа в два клона, `ruby_client` и `php_client`, изглеждащи като в История с няколко topic клона и слеем `ruby_client` последван от `php_client`, историята накрая ще изглежда като След сливане на topic клон.

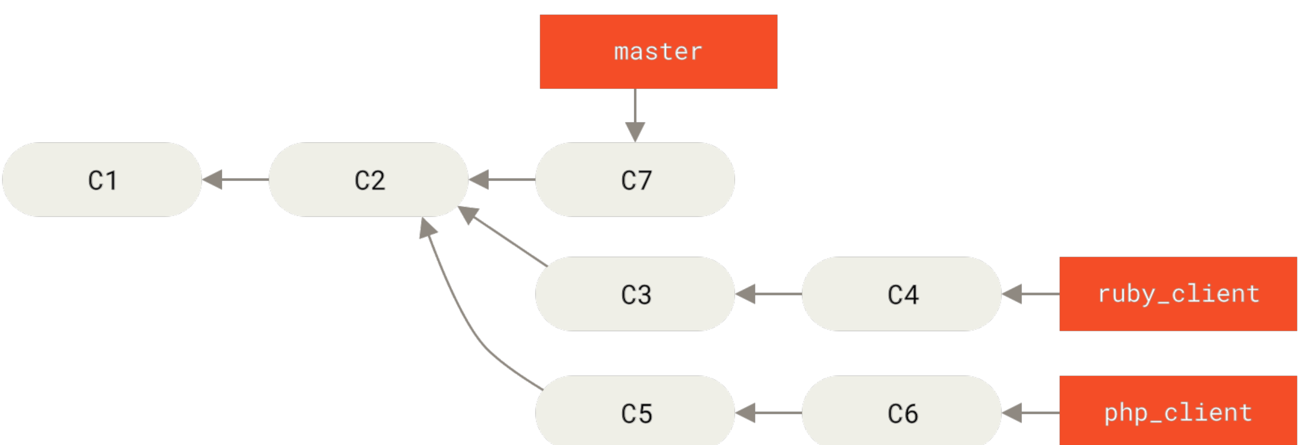


Figure 72. История с няколко topic клона

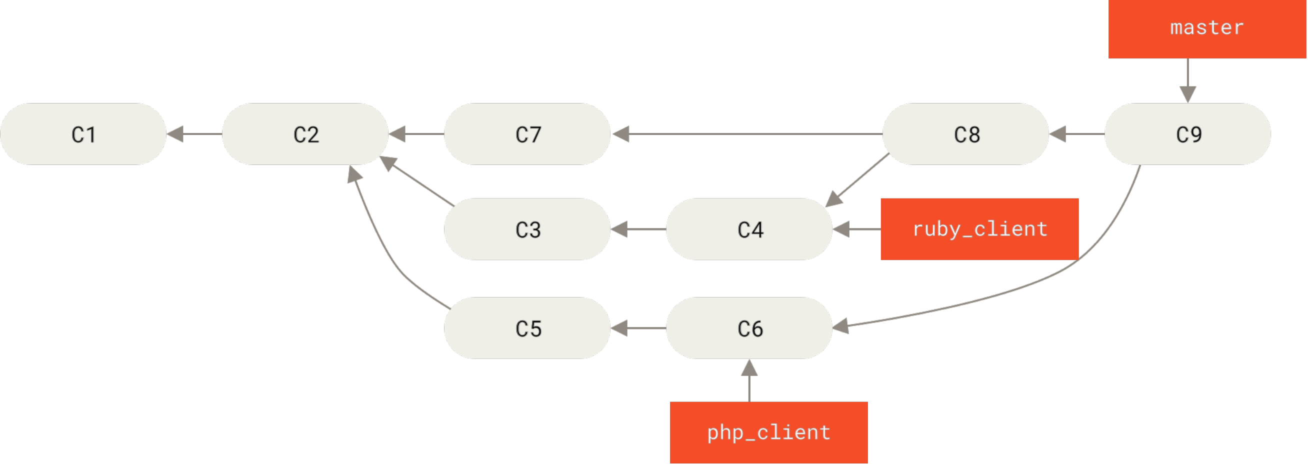


Figure 73. След сливане на topic клон

Вероятно това е най-простият работен процес, но може да бъде проблематичен, ако си имате работа с по-големи или по-стабилни проекти, където искате да сте много внимателни с това, което въвеждате като промени.

Ако проектът е особено важен, може да искате да използвате двустъпков цикъл на сливане. При такъв сценарий, имате два long-running клона, `master` и `develop`, при които `master` се обновява само, когато излезе много стабилна версия на проекта и промените преди това са интегрирани успешно в `develop`. И двата клона редовно се публикуват в публично хранилище. Всеки път, когато имате нов topic клон за сливане (Преди сливане на topic клон), вие го сливате в `develop` (След сливане на topic клон), след което тагвате нова версия и правите fast-forward на `master` към мястото, където сега е стабилния `develop` (След нова версия на проекта).

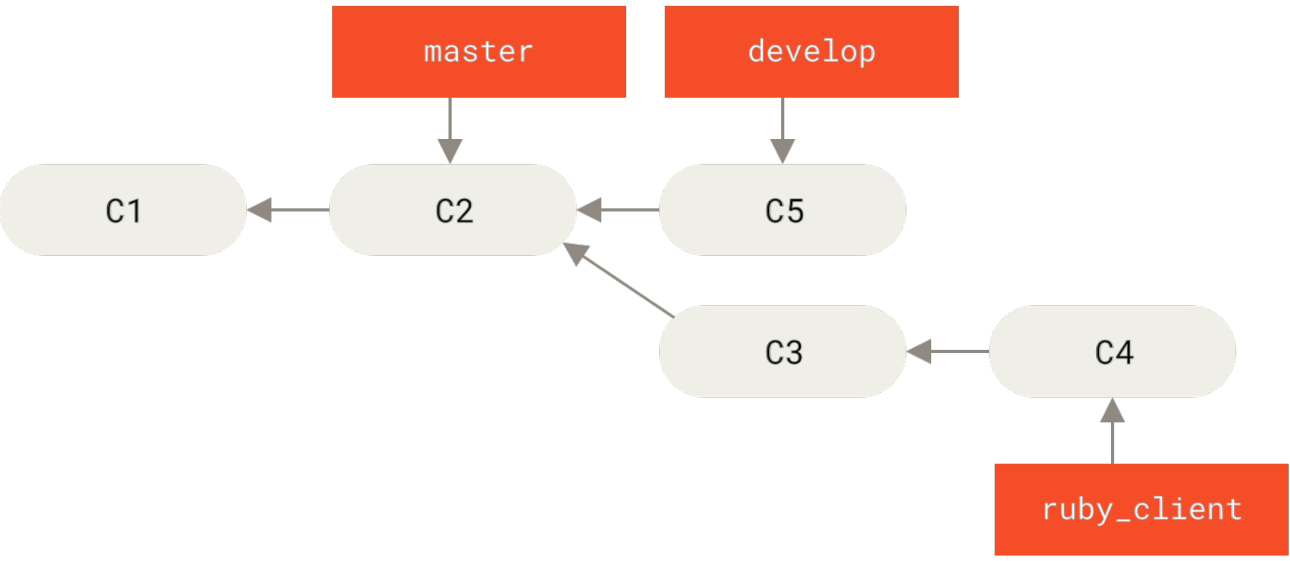


Figure 74. Преди сливане на topic клон

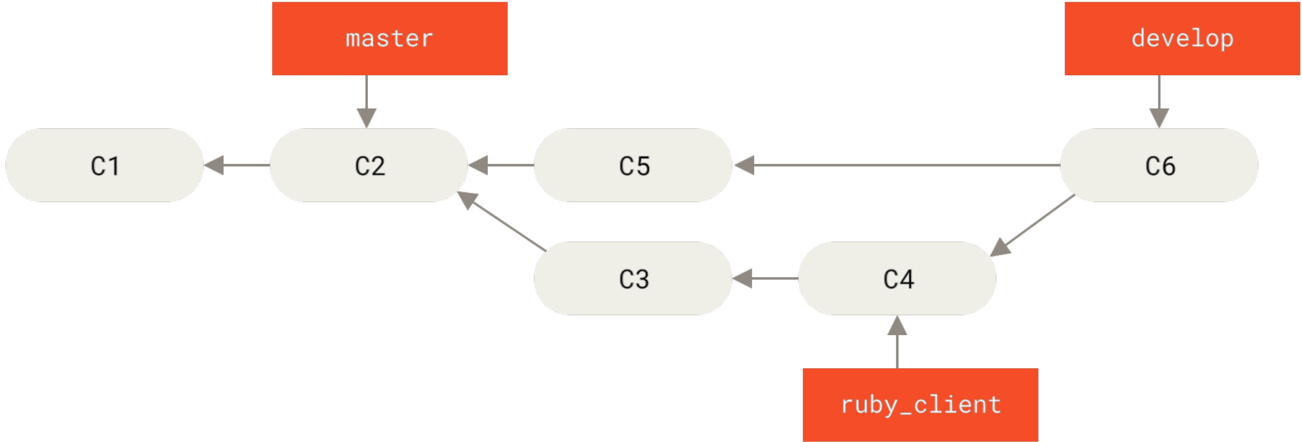


Figure 75. След сливане на topic клон

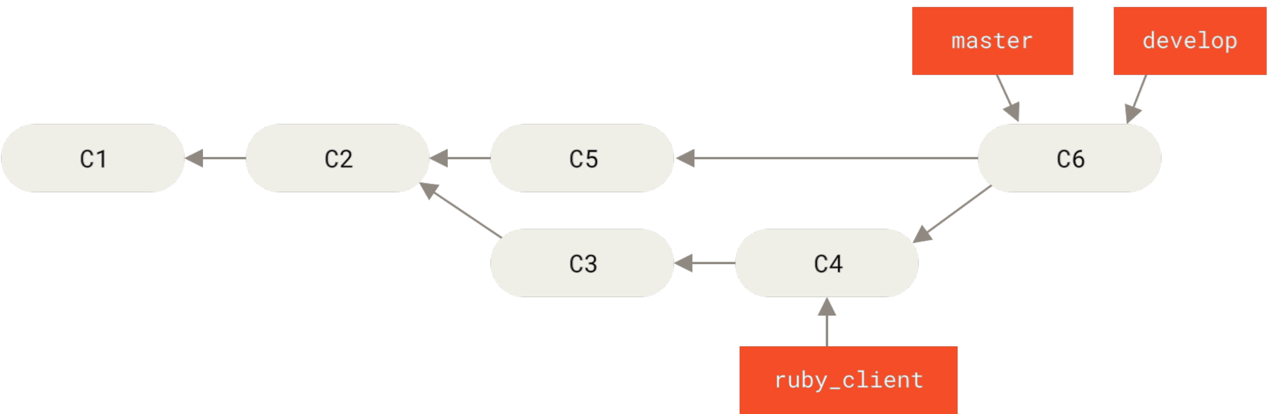


Figure 76. След нова версия на проекта

По такъв начин, когато някой клонира хранилището, може да извлече или `master` и да компилира последната стабилна версия, или `develop`, който съдържа най-пресните новости. Можете допълнително да разширите тази концепция с отделен `integrate` клон, където цялата работа е слята заедно. След това, когато кодът в него е стабилен и преминава тестовете, го сливате в `develop` клона и ако времето покаже, че и той е наистина надежден, правите fast-forward на `master`.

Large-Merging работни процеси

Проектът Git има четири long-running клона: `master`, `next`, и `seen` (преди това известен като *pu* — proposed updates) за нови промени, и `maint` за maintenance backports. Когато сътрудниците изпратят нова работа, тя се събира в topic клонове в хранилище на поддържащия проекта в маниер подобен на този, който описахме (вижте Управление на сложни серии от parallel contributed topic клонове.). В този момент, topic клоновете се изпитват, за да се определи дали са безопасни и готови за използване или трябва да се поправят допълнително. Ако са добре, те се сливат в `next` и този клон се публикува, така че всички могат да изпробват новите промени както са интегрирани заедно.

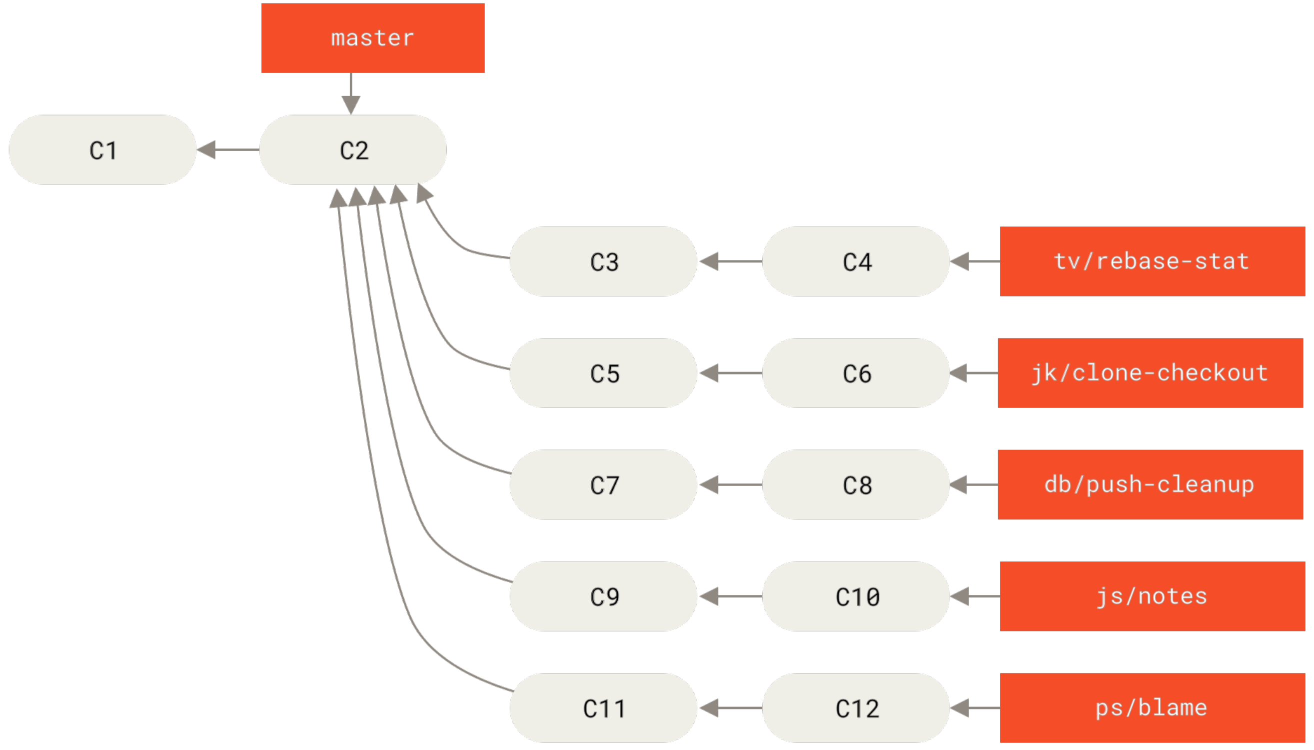


Figure 77. Управление на сложни серии от parallel contributed topic клонове.

Ако промените не са задоволителни, те вместо това се сливат в клона `seen`. Когато се установи, че са окончателно надеждни, промените биват интегрирани в `master`. След това `next` и `seen` се построяват отново от `master`. Това значи, че `master` винаги се мести напред, `next` от време на време се пребазира, а `seen` се пребазира още по-често:

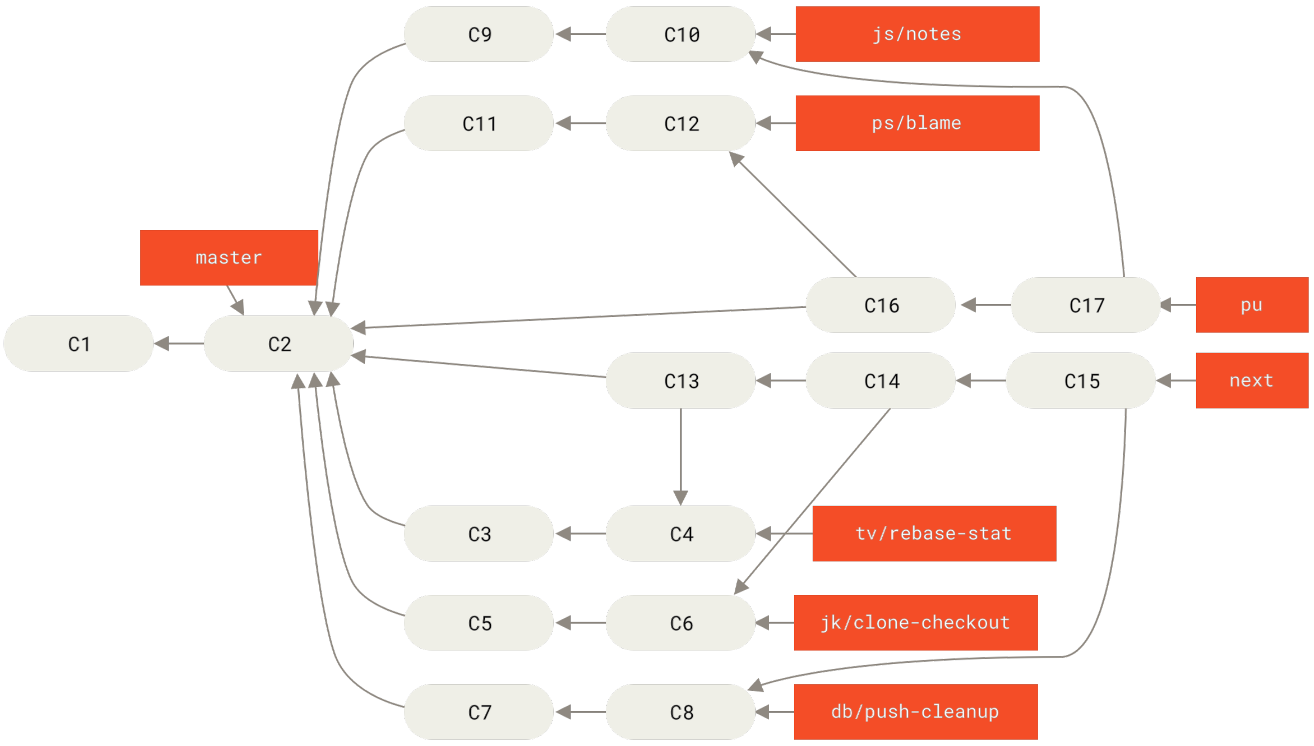


Figure 78. Сливане на topic клонове в long-term integration клонове.

Когато даден topic клон най-сетне се слее в `master`, той се изтрива от хранилището. Проектът Git също има и клон `maint`, който се прави от последната версия за да осигури backported

пачове в случай, че се изисква maintenance release. Така при клониране на Git хранилището, имате четири клона, които може да използвате за да пробвате проекта в различните му състояния на разработка, в зависимост от това колко актуален искате да бъде или как искате да допринасяте към него. А поддържащият проекта има структуриран работен процес за управление на новите промени. Все пак, Git проектът е със специализиран работен процес. За да го разберете напълно, можете да погледнете Git Maintainer's guide.

Rebasing и Cherry-Picking работни процеси

Други мениджъри на проекти предпочитат да пребазират или cherry-pick-ват нови промени на базата на `master` клона си, вместо да ги сливат. Целта е да поддържат почти линейна история. Ако се установи, че промените в topic клона са подходящи за интегриране, превключвате към него и изпълнявате rebase командата, за да възпроизведете промените на базата на текущия `master` клон (или от `develop` и т.н.). Ако това работи добре, можете да направите fast-forward на `master` клона и да запазите линейната история на проекта.

Другият начин да преместите нова работа от един клон в друг е да я cherry-pick-нете. Процесът cherry-pick в Git е подобен на rebase за единичен къмит. Той взема пача, който е бил въведен в даден къмит и се опитва да го приложи повторно в текущия клон. Това е полезно, ако имате множество къмити в topic клон и искате да интегрирате само един от тях или ако къмитът е само един, но предпочитате да не изпълнявате rebase. Да кажем, че имате проект изглеждащ така:

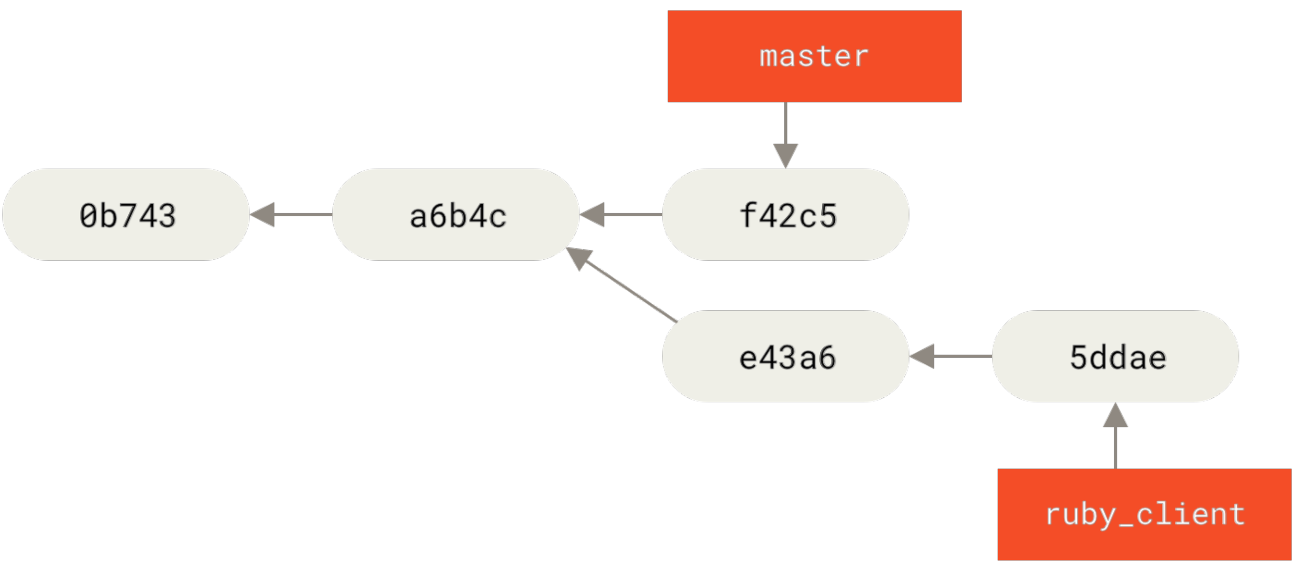


Figure 79. Примерна история преди cherry-pick

Ако искате да издърпате къмита `e43a6` в `master` клона, можете да направите следното:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Това въвежда същата промяна от `e43a6`, но получавате нов SHA-1 хеш за къмита, защото

приложената дата е различна. Сега историята изглежда така:

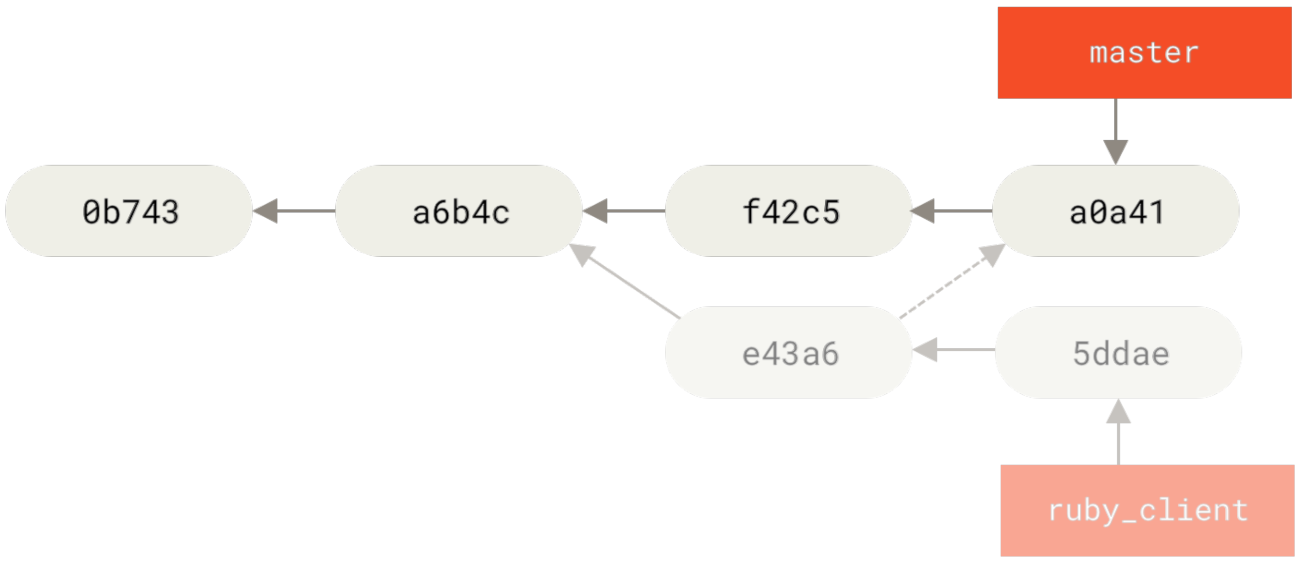


Figure 80. История след cherry-picking на къмит в topic клон

Сега можете да премахнете topic клона и да изоставите къмитите, които не желаете да въвеждате.

Rerere

Ако ви се налага да правите често сливания и пребазирания или ако поддържате topic клон с по-дълъг живот, Git предлага във ваша услуга опцията известна като “rerere”.

Терминът Rerere идва от “reuse recorded resolution” — това е начин за бързо прилагане на ръчно разрешаване на конфликти. Когато rerere е активна, Git ще съхранява набор от pre- и post-images от успешни сливания и ако установи, че е налице конфликт, който изглежда подобен на такъв, който е бил разрешен в миналото, автоматично ще използва записания начин за разрешаването му без да ви занимава с него.

Тази функционалност идва в две части: конфигурационна настройка и команда. Настройката е `rerere.enabled` и е достатъчно да имате следното в глобалната конфигурация:

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

Сега след като направите ръчно разрешаване на конфликт, резолюцията ще бъде запомнена в кеша и ще бъде използвана в бъдеще.

Ако е необходимо, може да комуникирате с rerere кеша с командата `git rerere`. Когато тя се използва самостоятелно, Git проверява базата си данни с решения за конфликти и се опитва да намери съответствие (въпреки, че това се прави автоматично, ако `rerere.enabled` е `true`). Също така има подкоманди за разглеждане на това какво ще се запише, за изтриване на конкретна резолюция от кеша и за изтриване на целия кеш. Ще разгледаме детайлно rerere в Rerere.

Тагване на версии

Ако сте решили да обявите стабилна версия на проекта, вероятно бихте желали да присъедините таг към нея, така че ако се налага по-късно да можете да я пресъздадете отново. Как да създадете таг видяхме в Основи на Git. Ако решите и да подпишете тага, тогава командата може да изглежда така:

```
$ 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
```

Ако правите подписване, може да се наложи да се погрижите за евентуални проблеми свързани с разпространението на публичния PGP ключ, който се използва в процеса. Поддържащият проекта на Git е решил това включвайки публичния си ключ директно като blob в хранилището и добавяйки таг, който сочи към него. За да направите и вие нещо подобно, може да установите кой е желания ключ изпълнявайки `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]
```

След това можете директно да импортирате ключа в базата данни на Git като го експортирате и прекарате през `git hash-object`, което ще създаде и запише нов blob обект със съдържанието в Git и ще ви върне обратно SHA-1 хеша му:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

След като вече имате съдържанието на ключа в Git, можете да създадете таг, който сочи директно към него посредством въпросната SHA-1 стойност:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Ако изпълните `git push --tags`, тогава тагът `maintainer-pgp-pub` ще бъде публикуван и споделен с всички останали. Ако някой от колегите ви желае да провери таг, може директно да извлече вашия PGP ключ като издърпа blob обекта от базата данни и го импортира в PGP:

```
$ git show maintainer-pgp-pub | gpg --import
```

Впоследствие този ключ може да се използва за проверка на всички подписани от вас

тагове. Също така, в таг съобщението може да включите инструкции за това как да се проверява тага и всеки би могъл да ги прочете изпълнявайки `git show <tag>`.

Генериране на номера на версии

Git не увеличава монотонно числата в стил *v123* или нещо подобно с всеки къмит. Ето защо, ако искате да имате нещо описателно с къмитите, бихте могли да изпълните `git describe` върху съответните къмити. В отговор, Git генерира стринг състоящ се от името на последния таг, който е по-ранен от този къмит, последвано от броя къмити след този таг и от частичната SHA-1 стойност на описвания къмит (добавя се и префиксът "g", означаващ Git):

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Така можете да експортирате snapshot или компилирана версия и да я именувате с разбираемо за хората описание. В действителност, ако компилирате Git от изходен код клониран от Git хранилището, `git --version` ще ви върща нещо като горното. Ако описвате къмит, който е бил тагнат директно, ще получите просто името на тага.

Командата `git describe` по подразбиране изисква анотирани тагове (тези създадени с флаговете `-a` или `-s`). Ако искате да се възползвате от lightweight (не-анотирани) такива, подайте ѝ опцията `--tags`. Може да използвате този стринг с командите `git checkout` или `git show`, въпреки че това зависи от SHA-1 стойността в края и може да не е валидно завинаги. Например, Linux kernel проекта напоследък премина от 8 на 10 символа за да гарантира SHA-1 уникалността на обектите и по този начин по-старите `git describe` изходни имена станаха невалидни.

Подготовка за издаване на Release

Допускаме, че искате да публикувате готова версия на продукта. Едно от нещата, които ще се наложи да направите, е да генерирате архив на най-новия snapshot за потребителите, които не ползват Git. Командата за това е `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Ако някой отвори този tarball ще получи актуалния snapshot на проекта ви в директория `project`. Можете също да създадете zip архив по подобен начин, подавайки `--format=zip` опцията към `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Сега имате tarball и zip версии на проекта, които можете да качите в уебсайт или да изпратите по имейла.

Shortlog

Време е да информирате хората от мейлинг листата ви за промените в проекта ви. Един удобен начин да получите бързо changelog на добавеното в проекта от последния път, когато сте изпратили известие, е командата `git shortlog`. Тя резюмира всички къмити в подадения ѝ обхват. Например, следното ще ви изведе обобщение на всички къмити от последния release насам, ако този release е бил наречен v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (6):
      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
```

Както се вижда, получавате доклад за всички къмити от v1.0.1 групирани по автор и можете да го изпратите на когото трябва.

Обобщение

Сега следва да се чувствате сравнително комфортно като сътрудник към Git проект и също така – като автор на ваш собствен проект, който интегрира в него промени от своите колеги. Поздравления, вече сте ефективен Git разработчик! В следващата глава ще ви срещнем с GitHub – най-мащабната и популярна хостинг платформа в Интернет.

GitHub

GitHub е най-голямата онлайн услуга за съхранение на Git хранилища и централна точка за съвместна работа на милиони разработчици и проекти. Голям процент от всички Git хранилища се хостват в GitHub и много проекти с отворен код ползват платформата като средство за проследяване на проблеми, преглед на код и много други неща. Въпреки, че GitHub не е директно свързан със самия Git като проект с отворен код, доста вероятно е да искате или да трябва да комуникирате с GitHub в определени моменти от професионалната ви работа.

Тази глава е посветена на това как да използваме ефективно GitHub. Ще разгледаме създаването и управлението на акаунт, създаване и ползване на Git хранилища, популярните методи за сътрудничество в проекти, за приемане на външна помощ във ваши проекти, програмният интерфейс на GitHub и много други малки трикове, които да ви улеснят работата.

Ако за вас GitHub не представлява интерес, можете спокойно да пропуснете тази глава и да преминете към Git инструменти.

Потребителският интерфейс се променя

Важно е да отбележим, че както нормално се случва с всички активно поддържани уеб сайтове, UI елементите в използваните тук екрани, могат да се променят с времето. Надяваме се, че все пак основната идея ще си остане достъпна, но ако искате актуални версии на екраните, онлайн версиите на книгата могат да осигурят такива.

Създаване и настройка на акаунт

Първото нещо, което трябва да сторите, е да си създадете безплатен акаунт. Просто отворете https://github.com, изберете си незаето потребителско име, посочете имейл адрес и парола и натиснете големия зелен бутон “Sign up for GitHub”.

Figure 81. Регистрационната форма на GitHub

Следващото нещо, което ще видите, е страницата с платени планове - засега можете да я игнорирате. GitHub ще ви изпрати имейл за потвърждение на подадения адрес. Продължете и направете това, доста е важно (както ще видите по-късно).

GitHub предоставя почти всичките си функции безплатно, с изключение на някои по-модерни екстри.

Платените планове на GitHub включват по-модерни инструменти и функции, както и увеличени лимити за безплатните услуги, но ние няма да се занимаваме с тях в тази книга. Повече информация за наличните планове и сравнения между плановете може да намерите на https://github.com/pricing.

Щраквайки върху Octocat логото в горния ляв ъгъл на екрана, ще попаднете във вашата dashboard страница. Вече можете да ползвате GitHub.

SSH достъп

В този момент, можете да се свързвате с хранилищата си по HTTPS с името и паролата, които създадохте при регистрацията. Обаче, за просто клониране на публични проекти, те не са ви нужни - акаунтът, който създадохте влиза в действие когато правите огледални копия на хранилища (fork) и изпращате промени към тези копия по-късно.

Ако искате да използвате SSH отдалечени хранилища, ще трябва да конфигурирате публичен ключ. Ако още нямате такъв, погледнете Генериране на SSH публичен ключ. Отворете настройките на акаунта през линка в горната дясна част на прозореца:

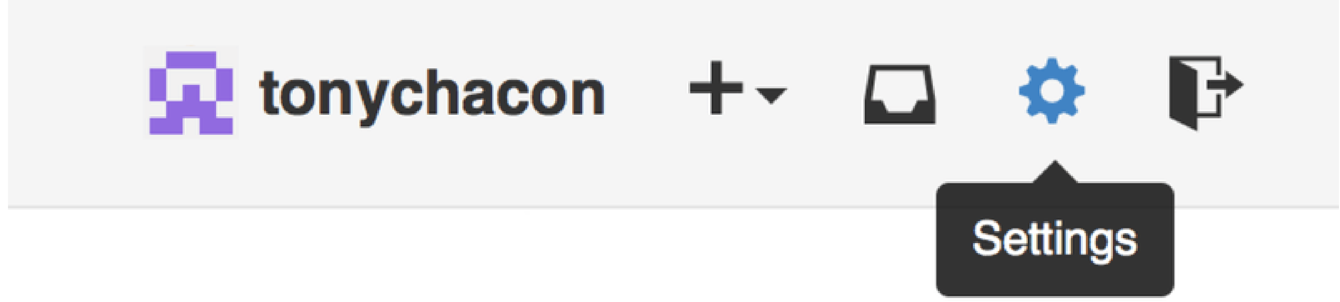


Figure 82. "Account settings" линк

След това, изберете "SSH keys" секцията отляво.

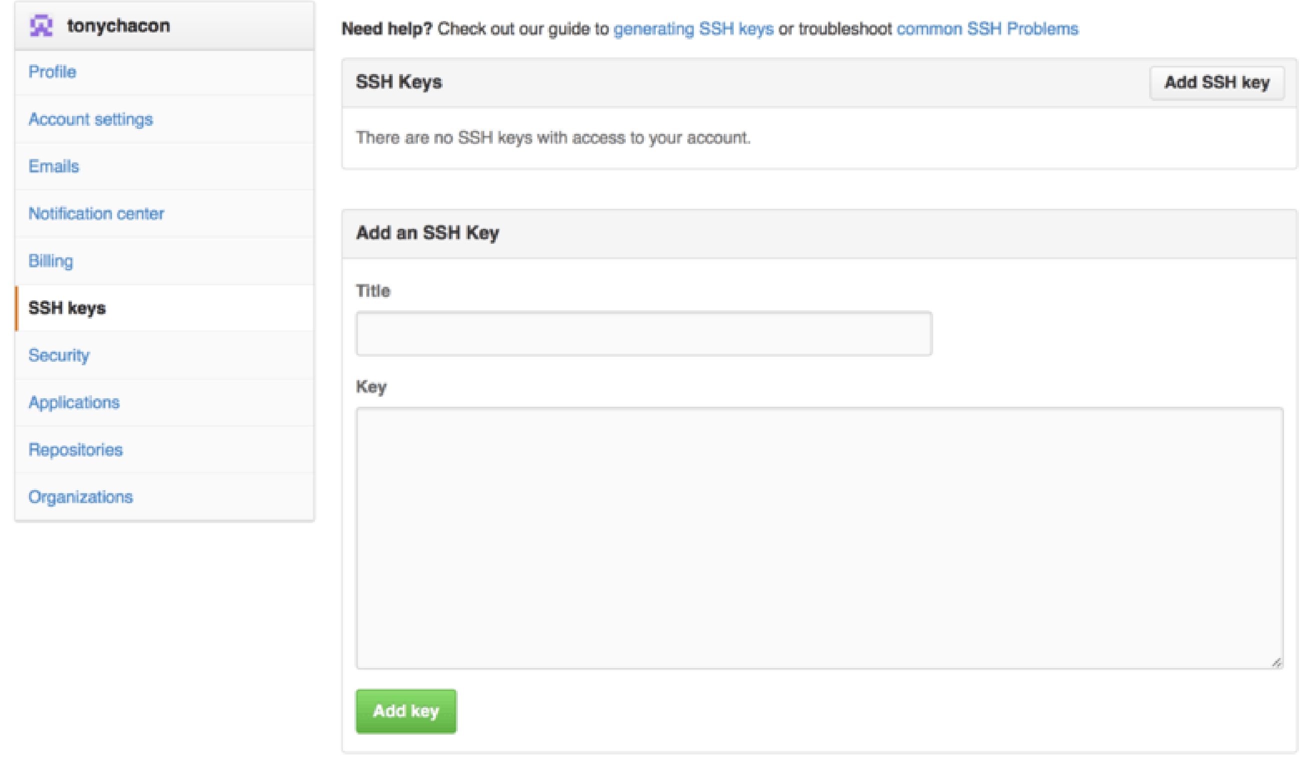


Figure 83. "SSH keys" линк

Оттук, натиснете бутона "`Add an SSH key`", дайте описателно име на ключа, вмъкнете съдържанието на файла ви `~/.ssh/id_rsa.pub` (или както сте го кръстили) в text area полето и натиснете "Add key".

Дайте име на SSH ключ, което се помни. Можете да използвате имена като "My Laptop" или "Work Account", така че ако искате по-късно да изтриете даден ключ, да можете лесно да разберете кой е той всъщност.

Вашият аватар

След това, ако желаете, можете да замените генерирания автоматично аватар с

изображение по ваш вкус. Отворете секцията “Profile” (над SSH Keys) и натиснете “Upload new picture”.

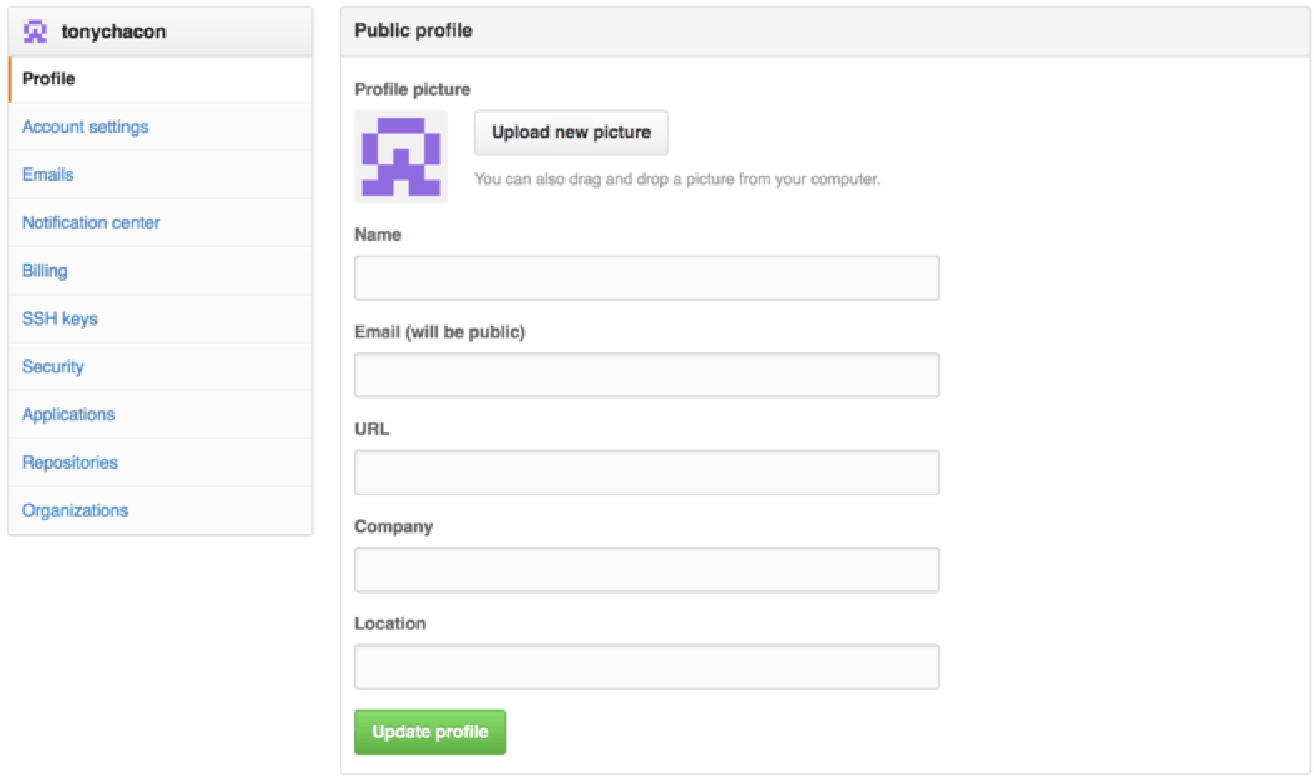


Figure 84. “Profile” линк

Ние сме избрали копие на Git логото на нашия диск и след това получаваме възможност да го изрежем.

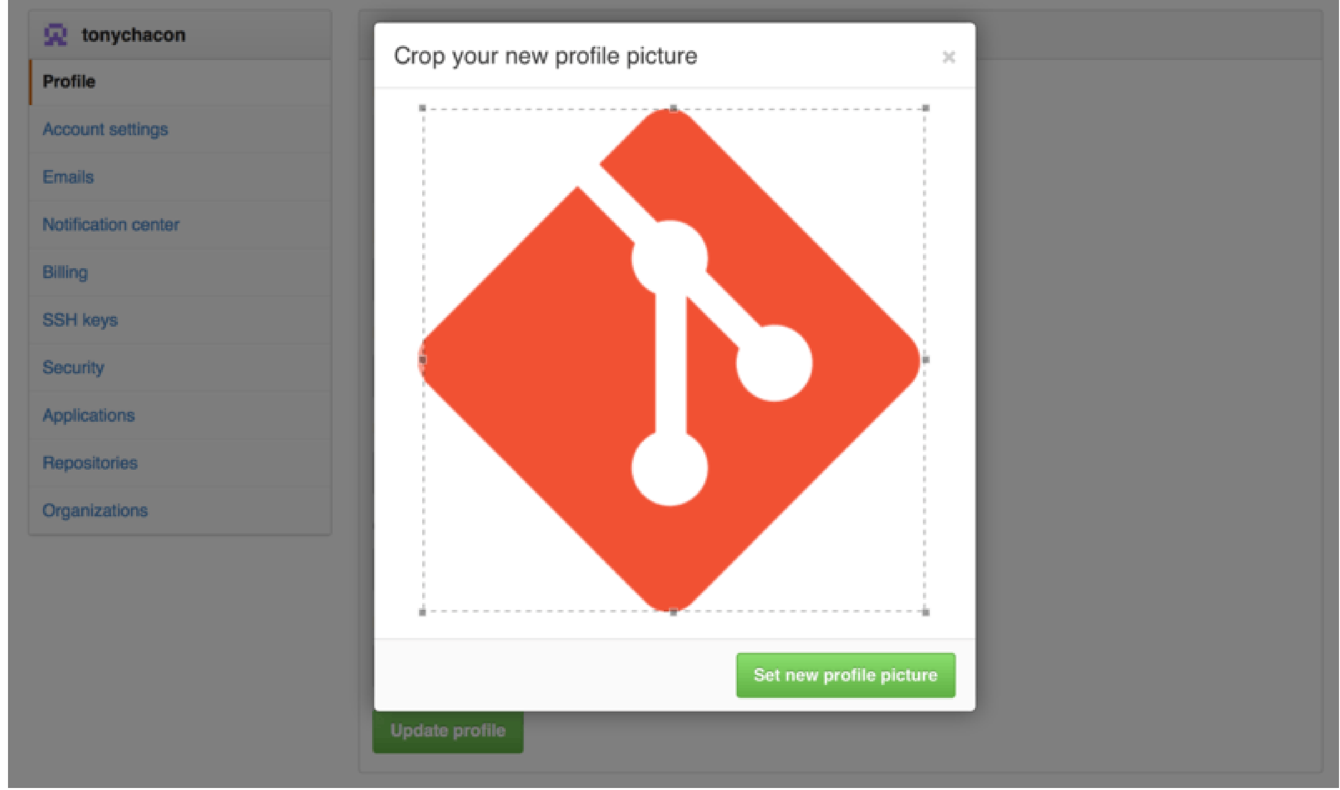


Figure 85. Изрязване на аватара

Сега всеки път, когато правите нещо в сайта, другите ще виждат аватара ви до потребителското име.

Ако сте качили аватар в популярната услуга Gravatar (често ползвана за Wordpress акаунти), то той ще бъде използван по подразбиране и няма нужда да минавате през тази стъпка.

Вашите имейл адреси

GitHub използва имейл адреса, за да асоциира Git къмитите с вашия потребител. Ако използвате няколко адреса във вашите къмити и искате GitHub да ги свързва правилно, ще трябва да ги добавите всичките в Emails секцията на административния интерфейс.

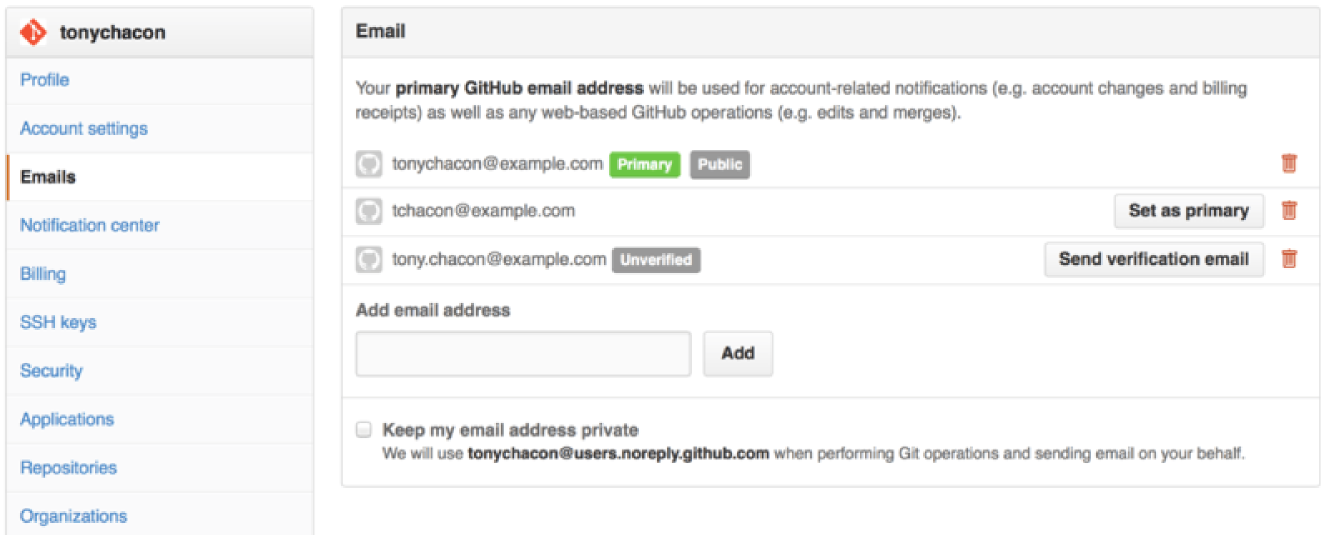


Figure 86. Добавяне на имейл адреси

В Добавяне на имейл адреси можем да видим някои от различните възможни статуси. Най-горният адрес се верифицира и определя като основен такъв, което значи че ще бъде използван за получаване на нотификации. Вторият адрес също се верифицира и може да бъде използван за основен, ако желаете да ги размените. Последният адрес не се верифицира и не може да се използва като основен. Ако GitHub види някой от тези адреси в къмит съобщенията в кое да е хранилище в сайта, то те ще бъдат асоциирани с вашия потребител.

Двустъпкова автентикация

Накрая, за допълнителна сигурност, определено е добре да настроите функцията за Two-factor Authentication или “2FA”. Двустъпковата автентикация е оторизационен механизъм, който е много популярен напоследък като средство за намаляване на риска от открадване на акаунта, в случай че някой разбере паролата ви. Включването ѝ ще укаже на GitHub да изисква два различни метода за оторизиране на достъпа, така че ако единият от тях е компрометиран, атакуващият да не може да получи достъп до акаунта ви.

Можете да откриете настройките за Two-factor Authentication в секцията Security на Account settings.

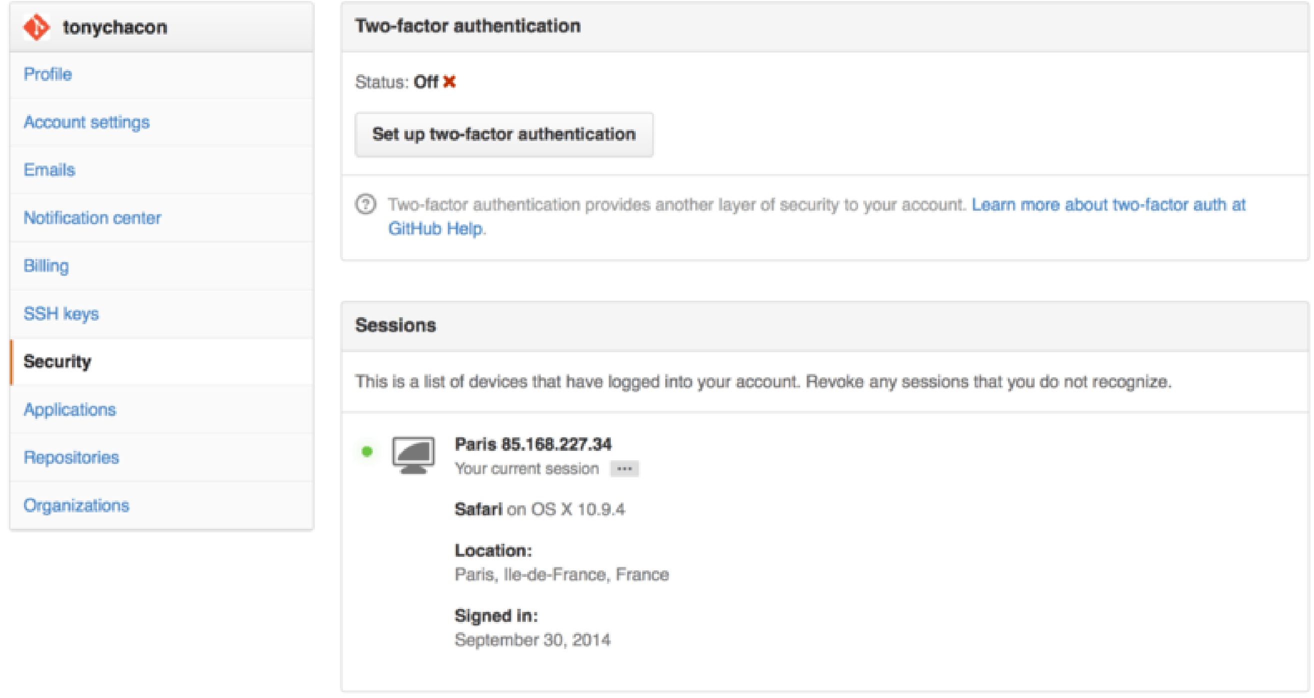


Figure 87. 2FA в секцията Security

Ако натиснете бутона “Set up two-factor authentication”, ще се отвори конфигурационна страница, в която можете да изберете да използвате телефонно приложение за генериране на вторичен код за достъп (“time based one-time password”), или пък можете да укажете на GitHub да ви изпраща SMS с кода всеки път, когато се логвате.

След като изберете желания метод и изпълните инструкциите за настройка на 2FA, акаунтът ви ще бъде малко по-защитен и ще трябва да въведете код за достъп в допълнение към паролата си всеки път, когато се логвате в GitHub.

Как да сътрудничим в проект

След като акаунтът ви е готов, нека погледнем някои полезни особености, които ще са ви от помощ, когато допринасяте към даден проект с ваш код.

Forking на проекти

Ако искате да сътрудничите в съществуващ проект, към който нямате push достъп, можете да “fork”-нете (да клонирате) проекта. Когато направите fork, GitHub ще направи копие на този проект, което е изцяло ваше; то съществува във вашия namespace и вие можете да пишете в него.

В исторически план, понятието “fork” се е възприемало и като негативно действие, при което злонамерен потребител взема проект с отворен код и го насочва в друга посока създавайки понякога конкурентен проект и разделяйки участващите в проекта потребители. В GitHub, “fork” е просто същия проект преместен във вашия namespace, който можете да променяте, начин да дадете вашия принос към оригиналния проект в по-достъпен стил.

По този начин, собствениците на проектите са освободени от грижата да дават права за

писане на потребителите-сътрудници. Хората просто fork-ват проект, пишат в копието и накрая предлагат своите промени обратно към оригиналното хранилище посредством похват известен като Pull Request, който ще разгледаме по-нататък. Това създава дискусионна нишка в сайта с възможност за code review, в която собственикът на проекта и допринасящия към него потребител могат да комуникират дотогава, докато собственикът реши, че предложените промени го задоволяват и може да ги слее в оригинала.

За да fork-нете проект, отворете страницата на съответното хранилище и натиснете “Fork” бутона в горната дясна част на страницата.

Figure 88. Бутонът “Fork”

След няколко секунди, ще бъдете прехвърлени към новата страница на проекта, където вече ще имате права за писане.

Работния процес в GitHub

GitHub е проектиран да следва специфичен модел за съвместна работа, в който централно място заемат Pull Requests заявките. Това работи както за малки екипи в единично споделено хранилище, така и в големи разпределени проекти със стотици разработчици и десетки fork-нати копия. Акцентът е върху Topic клонове работната последователност, която обсъждаме в Клонове в Git.

Ето как работят нещата накратко:

1. Fork-вате проект.
2. Създавате topic клон базиран на `master`.
3. Правите няколко къмита за да подобрите проекта.
4. Изпращате този topic клон към вашия GitHub проект, в който можете да пишете.
5. Създавате Pull Request в GitHub.
6. Дискутирате и (ако е необходимо) къмитвате допълнително код.
7. Собственикът на оригиналния проект слива или закрива отворения от вас Pull Request.
8. Синхронизирате обновения master обратно към вашия fork.

В общи линии, това е Integration Manager похвата за работа, който видяхме в Integration-Manager работен процес, само че вместо имейл съобщения за дискусия на промените, програмистите използват директно уеб-базираните инструменти в сайта на GitHub.

Нека видим това с един реален пример за проект в GitHub.

Можете да ползвате официалния **GitHub CLI** инструмент вместо уеб интерфейса на GitHub за повечето операции. Инструментът е наличен за Windows, MacOS, и Linux. Отидете на GitHub CLI homepage, където има инструкции за инсталация и ползването му.

Създаване на Pull Request

Да кажем, че Tony търси определен код, който да стартира на своя програмируем микроконтролер Arduino и го е открил в GitHub хранилището на адрес https://github.com/schacon/blink.

Figure 89. Проектът, в който искаме да сътрудничим

Кодът е ОК, единственият проблем е, че скоростта на примигване е твърде висока и решаваме да я намалим до веднъж на 3 секунди, вместо всяка секунда. Така че, нека променим програмата и да изпратим промяната си като предложим тя да бъде интегрирана в главния проект.

Първо, натискаме бутона *Fork*, за да си създаден собствено копие на проекта. Потребителското ни име е “tonychacon”, така че нашето копие на този проект ще е на адрес `https://github.com/tonychacon/blink` и там ще можем да пишем. Ще си го клонираме локално в компютъра, ще създадем topic клон, ще променим необходимото и ще изпратим промените обратно към GitHub.

```
$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (macOS) ③
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
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 'Change delay to 3 seconds' ⑤
[slow-blink 5ca509d] Change delay to 3 seconds
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
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
```

① Клонираме fork-натото хранилище локално в работния компютър.

② Създаваме описателен topic клон.

③ Променяме кода както смятаме за добре.

④ Проверяваме дали промяната е наистина добра.

⑤ Къмитваме промяната в topic клона.

⑥ Качваме topic клона обратно в нашето GitHub копие на хранилището.

Сега, ако се върнем обратно в сайта с нашето копие, ще видим че GitHub е установил, че сме

публикували нов topic клон и ще ни предложи голям зелен бутон с чиято помощ да проверим промените и да създадем Pull Request към оригиналния проект.

Можете също така да отворите “Branches” страницата на адрес `https://github.com/<user>/<project>/branches` за да намерите вашия клон и да създадете Pull Request директно оттам.

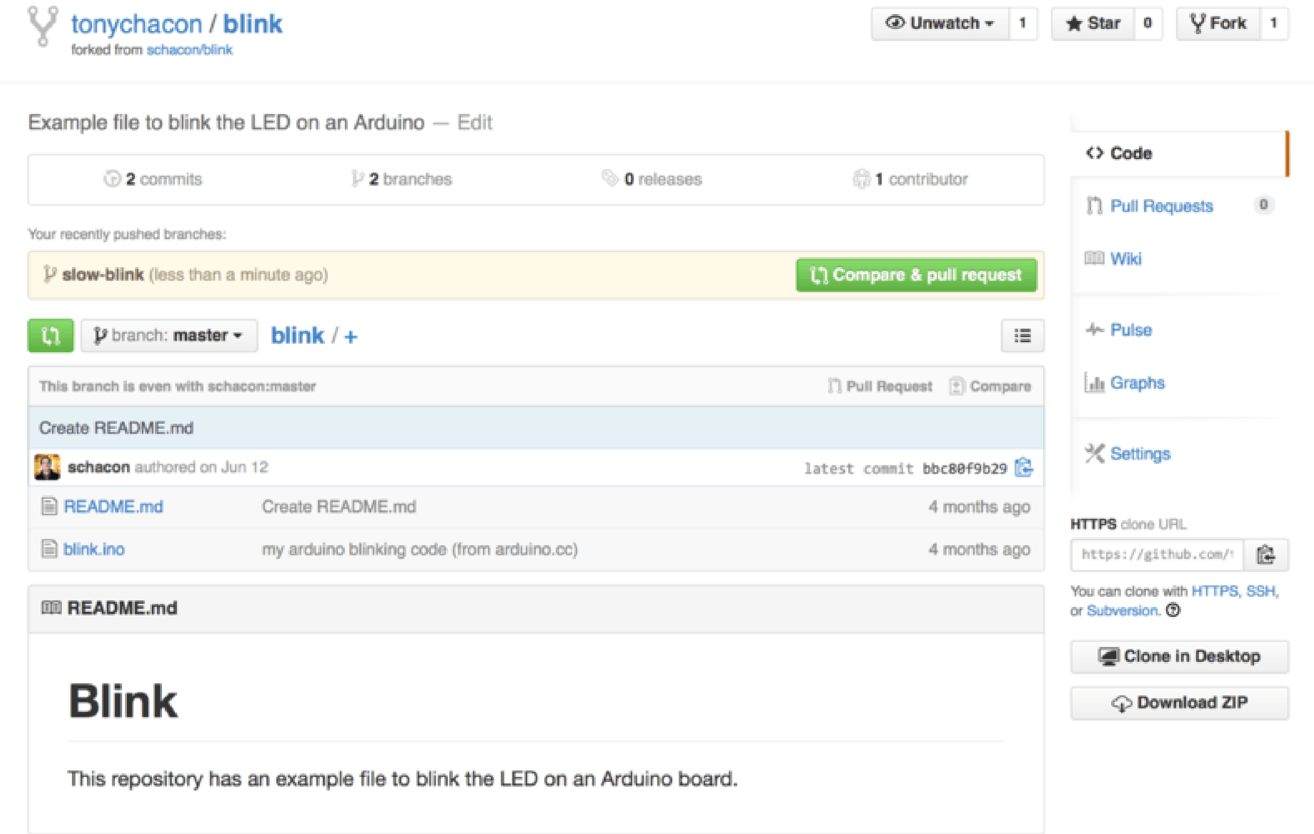


Figure 90. Pull Request бутон

Ако натиснем бутона, ще видим екран, който ни подканя да въведем заглавие и описание на нашия Pull Request. Почти винаги това усилие си струва, защото подходящото пояснение ще улесни собственика на проекта да разбере по-лесно целта на вашите промени, дали те са коректни и дали наистина подобряват проекта като цяло.

Също така, виждаме списък с къмитите на нашия topic клон, които са “ahead” (напред) в сравнение с `master` клона (в този случай, само един къмит) и унифициран diff със всички промени, които ще бъдат направени в проекта, ако собственикът му реши да направи сливането.

Figure 91. Страница за създаване на Pull Request

Когато натиснете бутона *Create pull request* на този екран, собственикът на оригиналния проект ще получи уведомление, че някой предлага промяна и линк към страница описваща всички промени.

Въпреки, че Pull Request заявките обикновено се ползват за публични проекти като този, когато сътрудникът е напълно готов с промените си, те също така често се използват във вътрешни проекти *в началото* на цикъла по разработка. Понеже можете да продължавате да променяте и публикувате topic клонове дори и **след** като Pull Request-ът от тях е пуснат, често той се създава много рано в процеса на работа и се използва като средство за постъпково следене на развитието на даден проект от целия екип, вместо да се създава в края на процеса.

Анализ на Pull Request

В този момент, собственикът на проекта може да погледне промените ви и да ги приеме, да ги отхвърли или да ги коментира. Нека допуснем, че той харесва идеята ви, но би

предпочел времето, в което светлината на платката е изключена, да е малко по-дълго от времето, в което е включена.

Докато цялата тази комуникация традиционно би се извършвала по имейл в работните процеси описани в Git в разпределена среда, то в GitHub тя се осъществява онлайн. Собственикът може да разгледа унифицирания diff и да постави коментар щраквайки директно върху кой да е ред от кода.

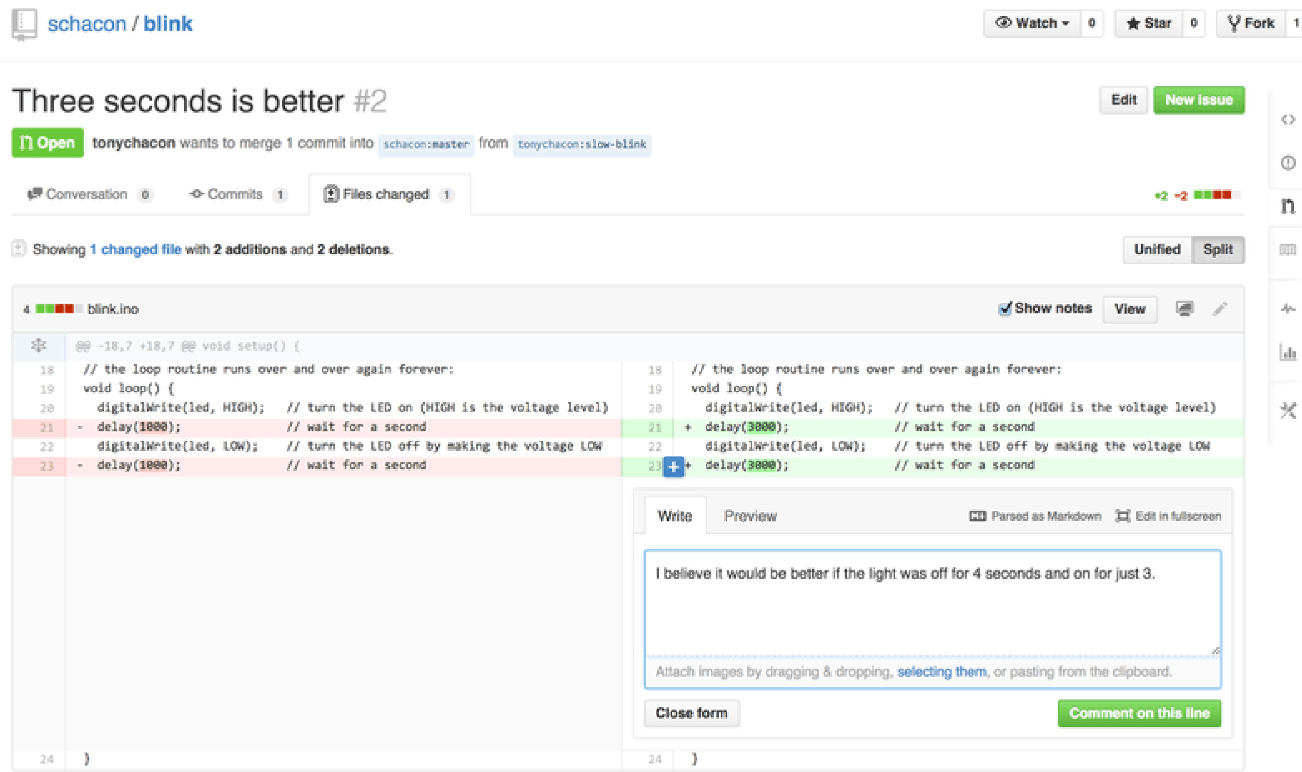


Figure 92. Коментар в специфичен ред от кода в Pull Request

Веднъж след като коментарът бъде направен, човекът отворил Pull Request-та (както и всички други следящи хранилището), ще получат уведомление за това. Ще видим това по-късно, но в общи линии, ако сме разрешили имейл нотификациите, Tony ще получи електронна поща с нещо подобно:

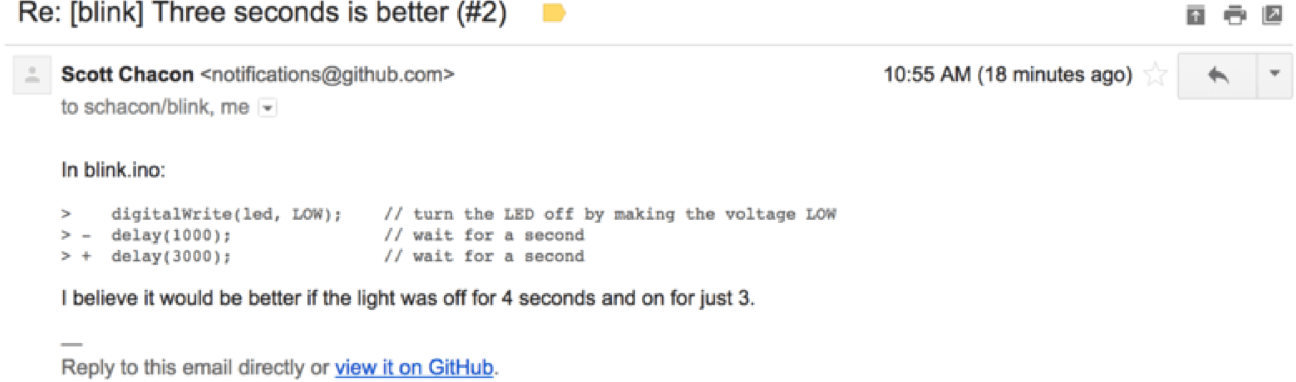


Figure 93. Коментарите изпратени като имейл уведомление

Също така, всеки може да остави общ коментар по Pull Request. В Страница за дискусии за Pull Request можем да видим пример, в който собственикът на проект коментира както

даден ред код, така и да оставя общ коментар в секцията за дискусии. Можете да видите, че code-коментарите също се показват в дискусията.

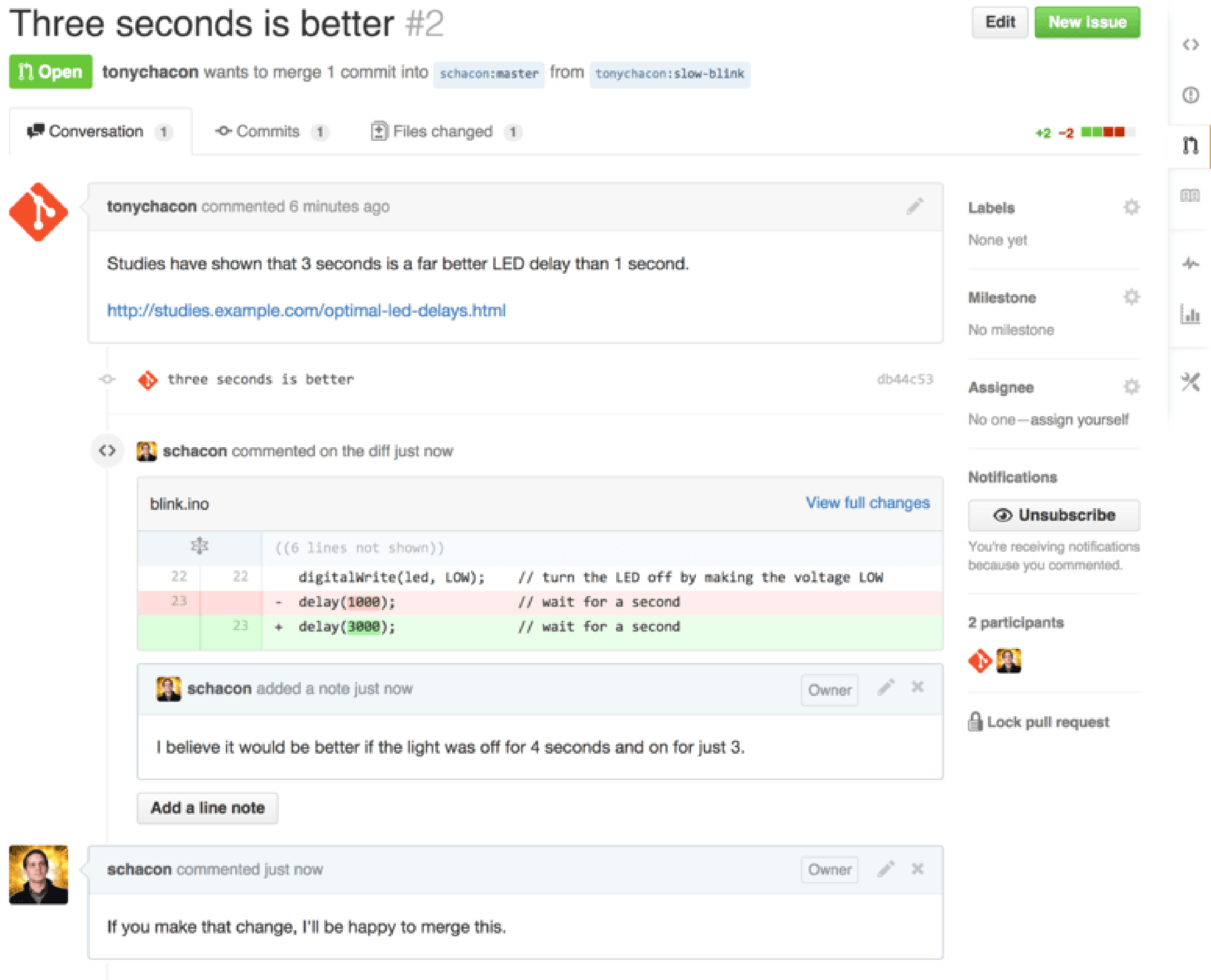


Figure 94. Страница за дискусии за Pull Request

Сега сътрудникът може да види какво трябва да коригира по кода си, за да бъде той приет за сливане. За щастие, това е доста лесно. Ако използвахте имейл, можеше да се наложи да съберете наново всичката информация и да я изпратите към мейлинг лист отново и отново. С GitHub обаче, вие просто къмитвате към topic клона, публикувате го в хранилището си и това автоматично обновява Pull Request-а. В Финален Pull Request можете също така да видите, че старият коментар в кода е свит в обновения Pull Request, тъй като е бил направен за ред, който после е бил променен.

Добавянето на къмити към съществуващ Pull Request не изпраща нови нотификации, така че след като Tony е публикувал корекциите си, той решава да остави коментар, за да информира собственика, че е направил исканите промени.

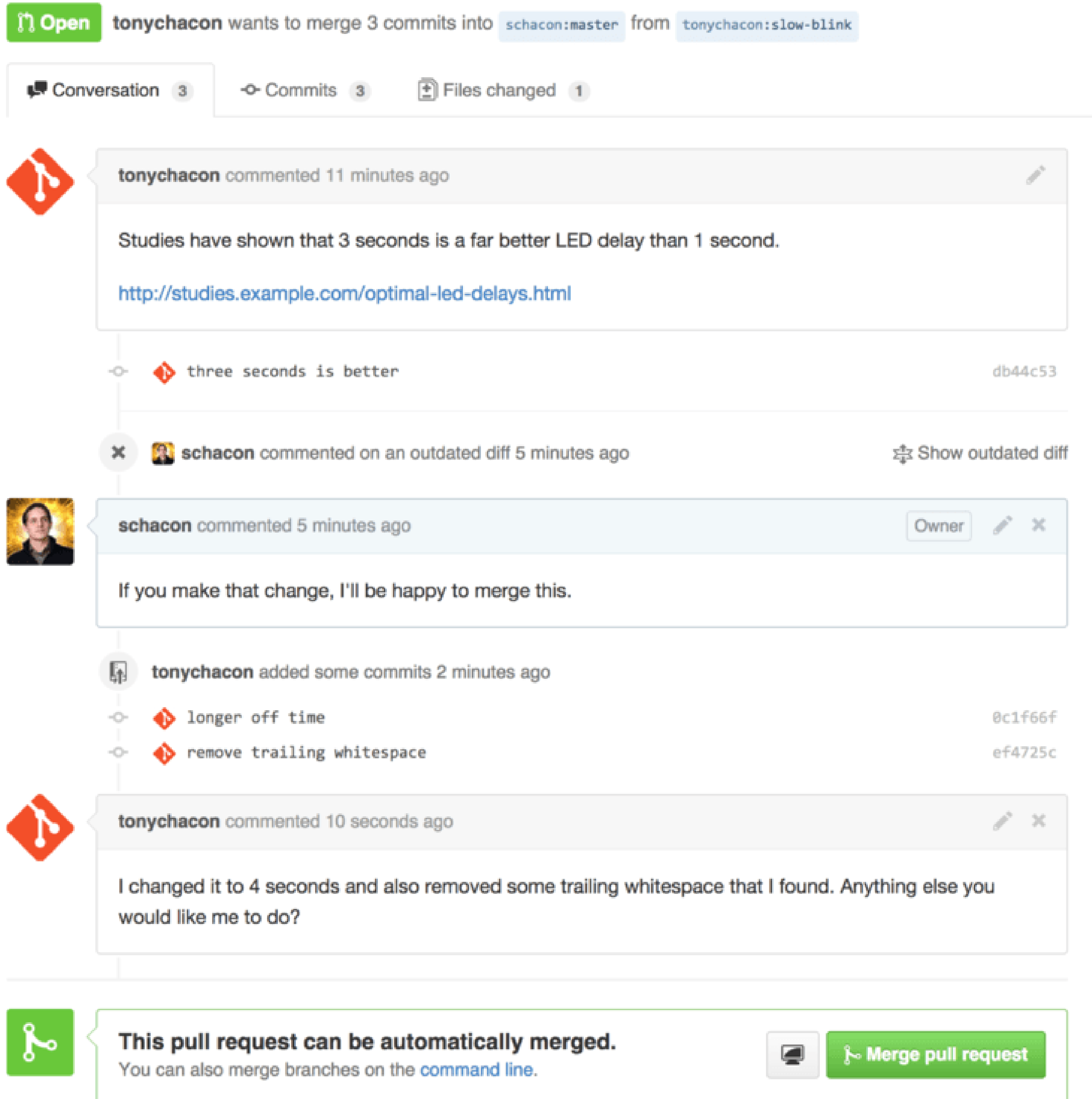


Figure 95. Финален Pull Request

Интересно за отбелязване, ако кликнете върху секцията “Files Changed” на този Pull Request, ще получите т. нар. “unified” diff — това са общите сумарни промени, които ще бъдат въведени в главния клон на проекта, ако този topic клон бъде слят. В смисъла на `git diff`, това автоматично ви показва `git diff master...<branch>` за клона, върху който е базиран този Pull Request. Вижте Изследване на промените за повече информация за този тип diff.

Другото, което ще забележите е, че GitHub проверява дали Pull Request-ът ви може да се слее чисто и ако е така - осигурява бутон да го направите директно на сървъра. Този бутон се показва само ако имате права за писане в хранилището и ако може да се направи тривиално сливане. Ако го натиснете, GitHub ще направи “non-fast-forward” сливане, което значи че дори сливането **да би могло** да бъде fast-forward, то все пак ще се създаде merge commit.

Ако предпочитате, бихте могли просто да изтеглите клона локално и да го слеете на компютър. Ако направите това и слеете клона с `master` клона и след това качите обратно в GitHub, то Pull Request-тът ще бъде автоматично затворен.

Това е базисният работен процес, който повечето GitHub проекти следват. Създават се topic клонове, по тях се отварят Pull Request-и, провежда се дискусия, евентуално се извършват допълнителни корекции и накрая Pull Request-ът или се приема или се затваря.

Не само Forks

Важно е да отбележим, че можете да създадете Pull Request между два клона в едно и също хранилище. Ако работите по дадена функционалност с някой друг колега и двамата имате права за писане в проекта, можете да публикувате topic клон в хранилището и да отворите Pull Request по него към `master` клона на същия проект. Така бихте могли да инициирате code review и дискусия. Не е необходимо да правите отделно копие на хранилището.

Pull Requests за напреднали

След като разгледахме основите на сътрудничеството в GitHub, нека погледнем някои интересни трикове и съвети за Pull Request-ите и как да ги използвате по-ефективно.

Pull Requests като пачове

Важно е да се разбере, че много проекти в действителност не разглеждат Pull Request-ите като поредици от перфектни пачове, които трябва да се прилагат чисто в определена последователност както примерно се гледа на тях в проектите базирани на комуникация през мейлинг листи. Повечето GitHub проекти гледат на Pull Request клоновете като на итеративна дискусия около дадена предложена промяна, кулминацията на която е унифицирания diff, който се прилага при сливането.

Това е важна разлика, понеже в общия случай промяната се предлага преди кодът да е възприет като перфектен - нещо което се случва далеч по-рядко при мейлинг-лист базираните серии от пачове. Това позволява контакт със собственика на проекта на много по-ранен етап и съответно достигането до финалното решение на даден проблем е много повече плод на съвместни усилия. Когато с Pull Request се предложи код и собствениците на проекта или трети страни предложат промяна по него, серията пачове не се излива наново а вместо това се публикуват само разликите под формата на нов къмит в клона като дискусията се премества напред в контекста на вече създадената работа.

Ако се върнете малко назад и погледнете отново Финален Pull Request, ще забележите, че сътрудникът не е пребазирал къмита си и не е създавал нов Pull Request. Вместо това, той е създал няколко нови къмита и ги е публикувал в същия клон. По този начин, ако по-късно във времето се върнете и разгледате отново този Pull Request, ще можете лесно да намерите контекста в който дадените решения са били взети. Натискането на бутона “Merge” в сайта умишлено създава merge къмит, който сочи към Pull Request-а, така че е лесно да се върнете назад и да изследвате оригиналната дискусия, ако е необходимо.

В тон с Upstream промените

Ако вашият Pull Request стане неактуален или по друга причина не се слива чисто, вероятно ще искате да го поправите, така че да може да бъде лесно слят от собственика на проекта на по-късен етап. GitHub ще тества това за вас и ще ви уведомява със съобщение в долната част на екрана на всеки Pull Request дали сливането е тривиално или не.

Figure 96. Pull Request, който не се слива чисто

Ако видите нещо като Pull Request, който не се слива чисто, ще трябва да поправите вашия клон, така че да стане отново "зелен" и да не се налага собственикът на проекта да извършва допълнителни дейности.

Имате два начина да се оправите в подобни случаи. Можете или да пребазирате вашия клон върху целевия такъв (нормално `master` клона на хранилището, което сте fork-нали), или да слеете целевия клон с вашия.

Повечето разработчици в GitHub правят второто по същите причини, които разгледахме в предишната глава. Това, което е важно е историята и финалното сливане, така че пребазирането не ви дава много повече от една идея по-чиста история - а за сметка на това е **много** по-труден и податлив на грешки процес.

Ако решите да слеете актуалния целеви клон, така че Pull Request-ът ви да може да се слива, ще добавите оригиналното хранилище като нова отдалечена референция, ще издърпате данните от нея, ще слеете основния ѝ клон във вашия topic клон, ще поправите евентуалните проблеми и накрая ще изпратите промените в същия клон, от който сте създали Pull Request-а.

Например, нека кажем, че в “tonychacon” случая, който вече използвахме, оригиналният автор е направил промяна, която ще доведе до конфликт с нашия Pull Request. Нека следваме тази последователност:

```
$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
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 ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
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
```

① Добавяме оригиналното хранилище като remote с име “upstream”.

② Издърпваме най-новата работа от това отдалечено хранилище.

③ Сливаме главния клон на това хранилище с нашия topic клон.

④ Оправяме възникналия конфликт.

⑤ Публикуваме обратно към същия topic клон.

След като направим това, Pull Request-ът ни автоматично ще бъде обновен и проверен дали се слива безконфликтно.

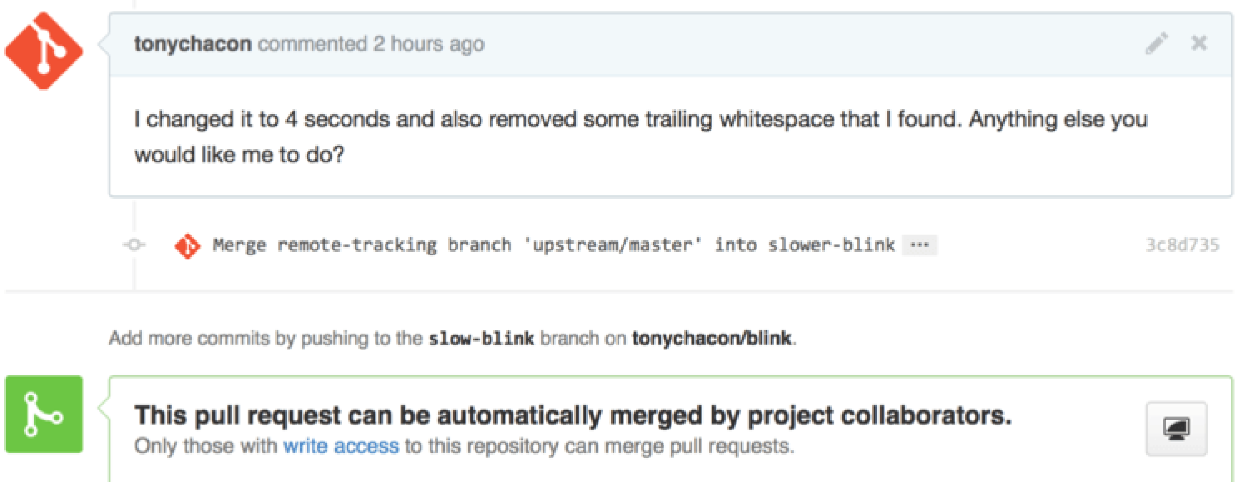


Figure 97. Pull Request-ът сега се слива чисто

Едно от най-добрите неща в Git е, че можете да правите това непрекъснато. Ако имате много дълго продължаващ проект, можете спокойно да издърпвате и сливате главния му клон много пъти и само трябва да се справяте с евентуално възникналите конфликти след последното ви сливане, което прави процеса лесно управляем.

Ако все пак категорично искате да пребазирате клона, за да го прочистите, със сигурност можете да направите това, но горещо се препоръчва да не се презаписва клон от който има пуснат Pull Request. Ако други хора вече са го изтеглили и са работили по него, ще се сблъскате с всички проблеми посочени в Опасности при пребазиране. Много по-малко болезнено би било да публикувате пребазирания клон като нов такъв в GitHub и да отворите нов Pull Request, който сочи към предишния и след това да закриете стария.

Указатели

Следващият ви въпрос би могъл да е "Как да се обръщам към стар Pull Request?". Оказва се, че съществуват голям брой начини да се обръщате към други неща почти навсякъде, където можете да пишете в GitHub.

Нека започнем с това как да направим cross-reference към друг Pull Request или Issue. Към всеки Pull Request или Issue се асоциира номер и тези номера са уникални в рамките на проекта. Например, не можете да имате Pull Request #3 *и* Issue #3. Ако искате да създадете указател към всеки Pull Request или Issue от кой да е друг такъв, можете просто да въведете `#<num>` във всеки ваш коментар или описание. Можете да бъдете и по-специфични, ако Pull Request-ът се намира някъде другаде; напишете `username#<num>` ако се обръщате към Issue или Pull Request във fork на хранилището, в което се намирате. Или дори `username/repo#<num>` за да укажете обект от съвсем друго хранилище.

Нека погледнем един пример. Да приемем, че сте пребазирали клона в предишния пример, създали сте нов Pull Request за него и сега искате да посочите стария Pull Request от новия такъв. Също така, искаме да укажем връзка към Issue което се намира във fork на хранилището и в заключение - Issue в съвсем различен проект. Можем да попълним описанието точно както Cross references в Pull Request

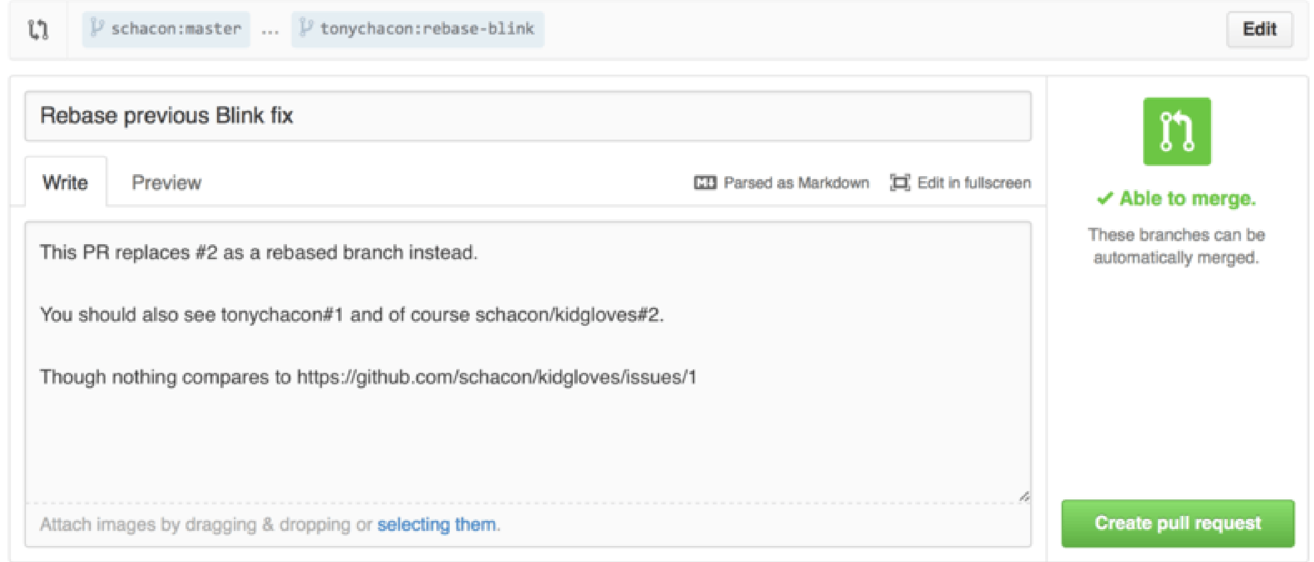


Figure 98. Cross references в Pull Request

Когато изпратим този Pull Request, ще видим всичко това рендерирано като на фигурата Cross references рендерирани в Pull Request.

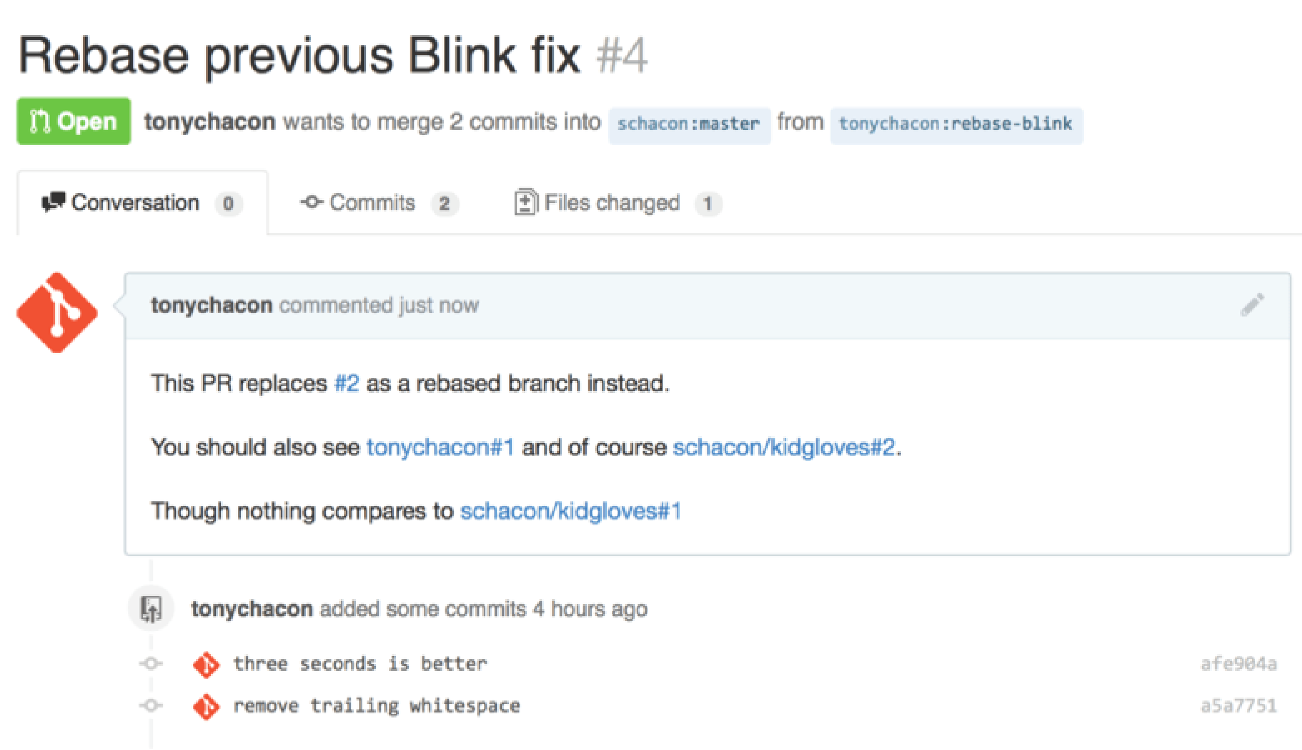


Figure 99. Cross references рендерирани в Pull Request

Забележете, как пълният GitHub URL, който поставихме, е съкратен така че да показва само нужното.

Ако сега Tony се върне и затвори оригиналния Pull Request, ще можем да видим това отбелязано в новия - GitHub автоматично е създал trackback събитие в Pull Request времевата линия. Това означава, че всеки който посети този Pull Request и види, че той е затворен, може лесно да направи връзка с този, който го замества. Линкът ще изглежда по подобен начин като на фигурата Линк към новия Pull Request в линията на времето на затворен Pull Request.

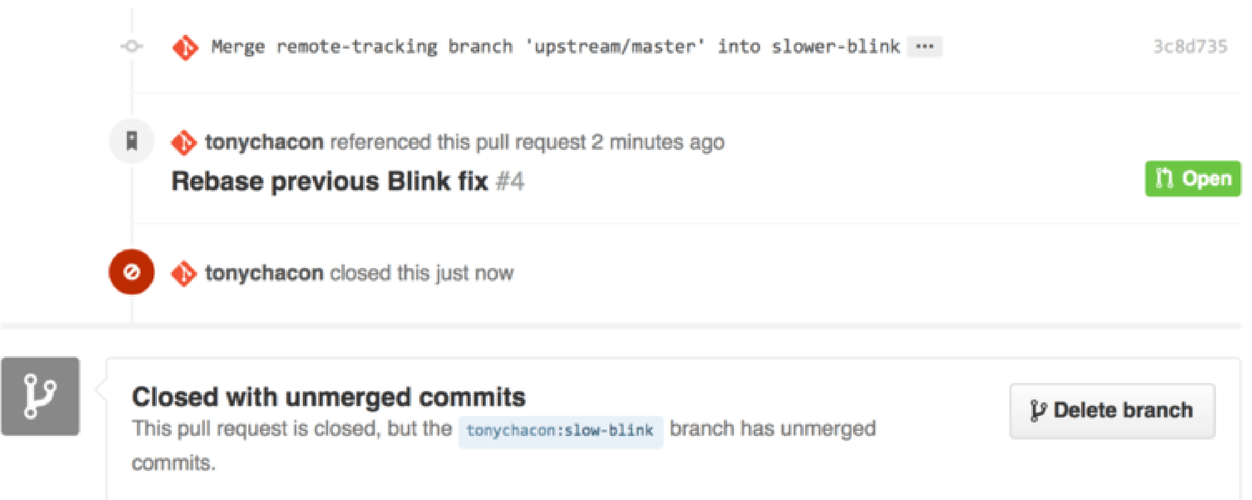


Figure 100. Линк към новия Pull Request в линията на времето на затворен Pull Request

Освен чрез Issue номерата, можете също така да сочите към определен къмит чрез SHA-1 сумата. Трябва да укажете пълните 40 символа и ако GitHub засече това в коментар, ще бъде създаден линк директно към къмита. Отново, можете да сочите към къмити във fork-нати хранилища или към такива в други хранилища по същия начин, по който го правехте с Issues.

Markdown в GitHub

Линкването към други Issues е само началото на интересните неща, които можете да правите в почти всяка текстова кутия в GitHub. В описанията на Issue и Pull Request-ите, коментарите, коментарите към кода и в други места, можете да използвате т. нар. “GitHub Flavored Markdown”. С Markdown езика пишете обикновен текст, който се форматира обогатен, когато се рендерира на страницата.

Вижте GitHub Flavored Markdown по време на писане и рендериран за пример как коментари или текст могат да се пишат и рендерират с Markdown.

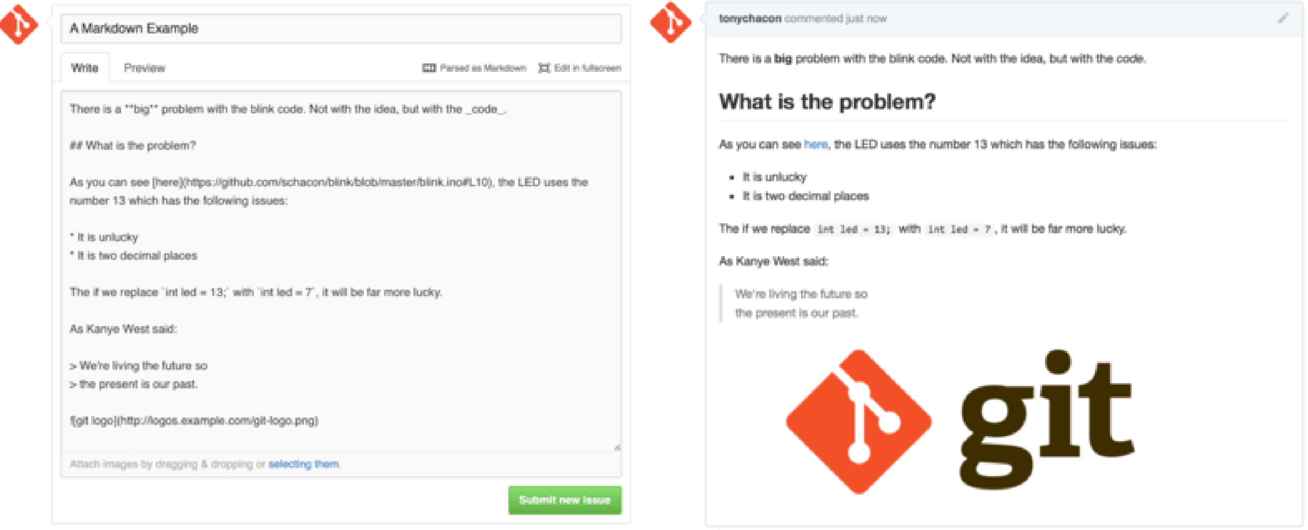


Figure 101. GitHub Flavored Markdown по време на писане и рендериран

Версията на Markdown, която ползва GitHub, добавя повече екстри към стандартния Markdown синтаксис, които подпомагат създаването на описателни и ясни инструкции към

Pull Request-и или Issues.

Списъци със задачи

Една много интересна възможност, специално за ползване с Pull Request-и, е списъкът със задачи (Task List). По същество това е поредица от чекбоксове, с които да отмятате нещата, които трябва да се свършат. Поставянето им в Issue или Pull Request обикновено индикира, че искате определени задачки да бъдат свършени преди съответния Pull Request/Issue да се приеме като завършен.

Списък се създава така:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

Ако въведем подобен код в описание за Pull Request или Issue, ще го видим рендериран като Списък задачи рендериран в Markdown коментар..

Figure 102. Списък задачи рендериран в Markdown коментар.

Това често се ползва в Pull Request-и за да се укажат нещата, които трябва да се свършат в клона код преди Pull Request-а да бъде одобрен за сливане. Истински готината част е, че можете да натискате чекбоксовете за да обновявате коментара — не е необходимо да редактирате Markdown текста директно за да маркирате задачите като свършени.

В допълнение, GitHub ще претърси вашите Pull Request и Issues за списъци със задачи и ще ги покаже като метаданни на страниците им. Например, ако имате Pull Request със задачи и погледнете overview страницата на всички Pull Request-и, можете да видите до каква степен задачите са изпълнени. Това позволява на потребителите да разделят Pull Request-ите на подзадачи и помага на другите да проследяват развитието на клона. Може да видите пример за това в Преглед на Task list статуса в списъка от Pull Request-и.

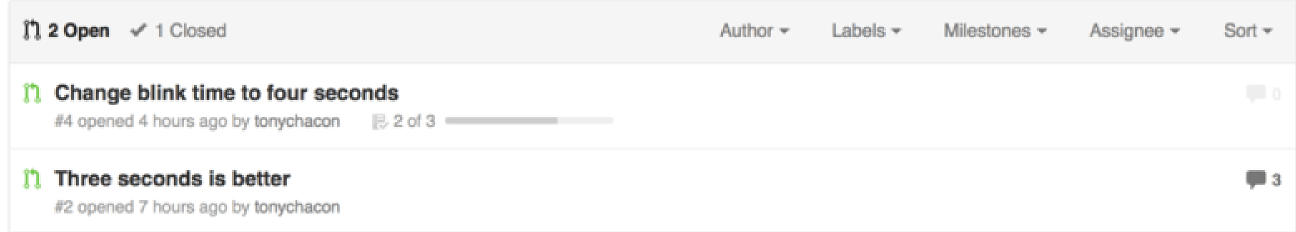


Figure 103. Преглед на Task list статуса в списъка от Pull Request-и

Това е особено полезно, когато отворите Pull Request рано във времето и го използвате да за следите как вървят дейностите по имплементация на дадена възможност.

Отрязъци код

В коментарите можете да добавяте и отрязъци от код. Това е особено полезно, ако искате да представите нещо, което *бихте могли* да направите преди в действителност да го имплементирате като къмит в клона. Често се използва и за добавяне на примерен код за нещо, което не работи или пък за това какво имплементира дадения Pull Request.

За да добавите част от код в текста, трябва да го оградите в обратни апострофи.

````
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
````

Ако добавите и името на езика за програмиране, както сме го направили в примера с *java*, то GitHub също така ще опита да оцвети синтактично отрязъка код. В горния пример, рендерираният текст ще изглежда като Рендериран програмен код

Figure 104. Рендериран програмен код

Цитати

Ако искате да отговорите на малка част от дълъг коментар, можете избирателно да го цитирате слагайки символа `>` преди редовете. В действителност, това се използва толкова често, че дори съществува клавиатурна комбинация. Ако селектирате текст в коментар, който искате директно да цитирате и натиснете клавиша `r`, това ще създаде автоматично

желания цитат в текстовото поле.

Цитатите изглеждат така:

```
> 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?
```

Веднъж рендериран, коментарът изглежда като Рендериран цитат.

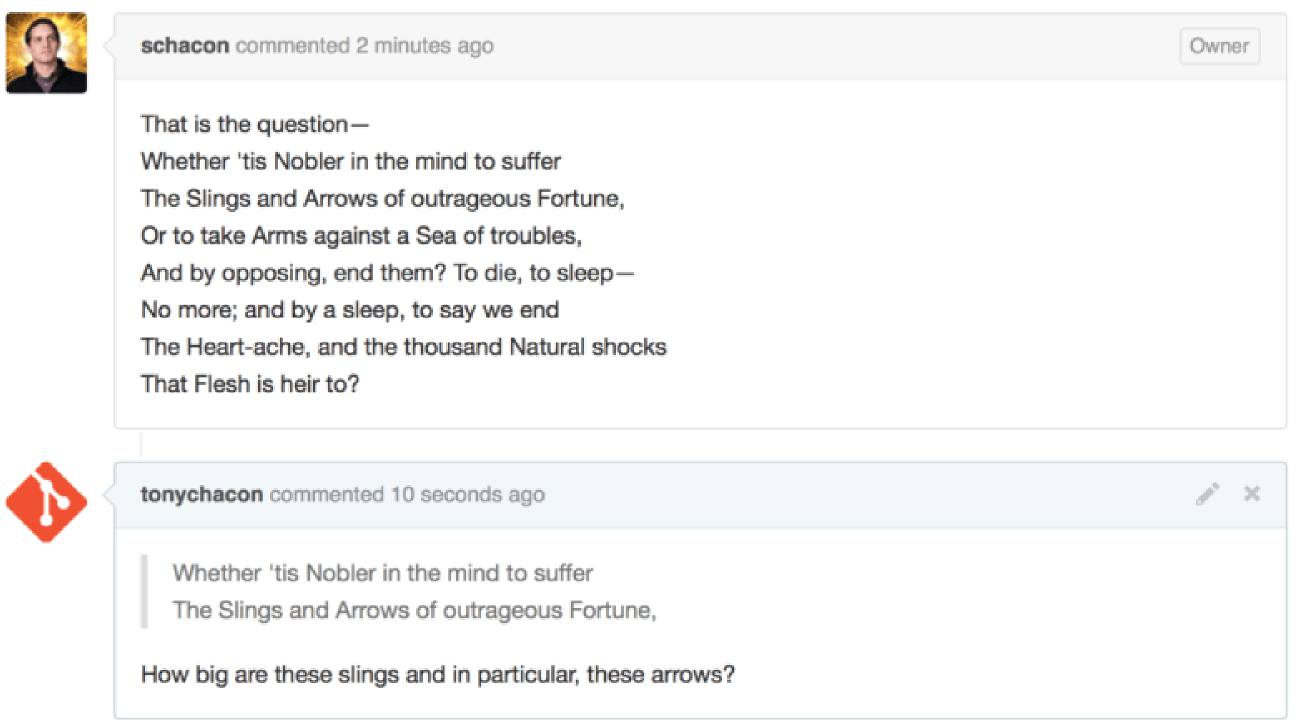


Figure 105. Рендериран цитат

Emoji

Накрая, можете също така да използвате и emoji. В действителност, тази възможност се ползва доста често в коментарите на Issues и Pull Request-ите като дори съществува emoji helper. Ако пишете коментар и започнете със символа `:`, то ще се покаже autocompleter за да ви помогне да изберете.

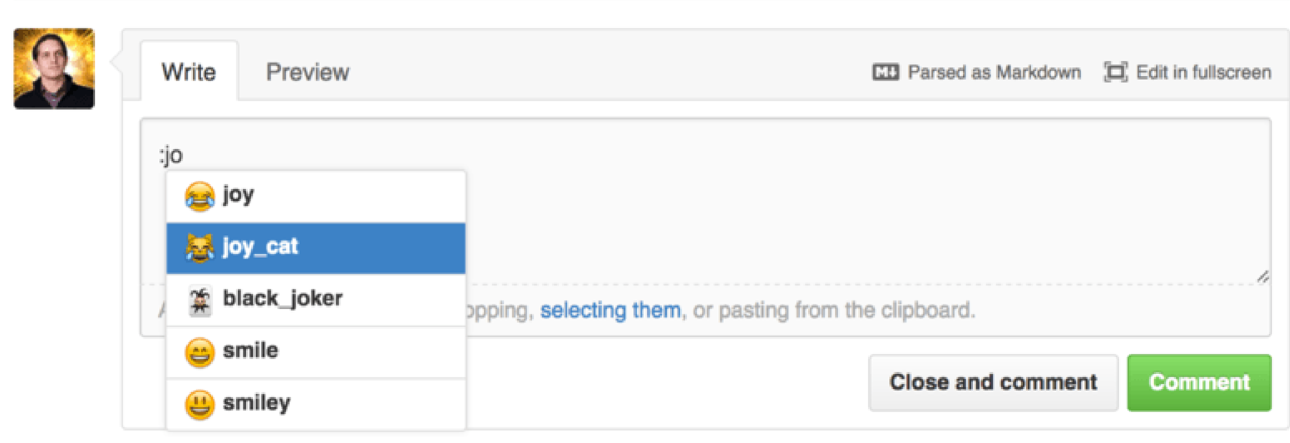


Figure 106. Emoji autocompleter в действие

Emoji-тата са под формата на `:<name>:` навсякъде в коментара. Например, можете да напишете това:

```
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:
```

И те ще изглеждат рендерирани като Интензивно ползване на emoji.

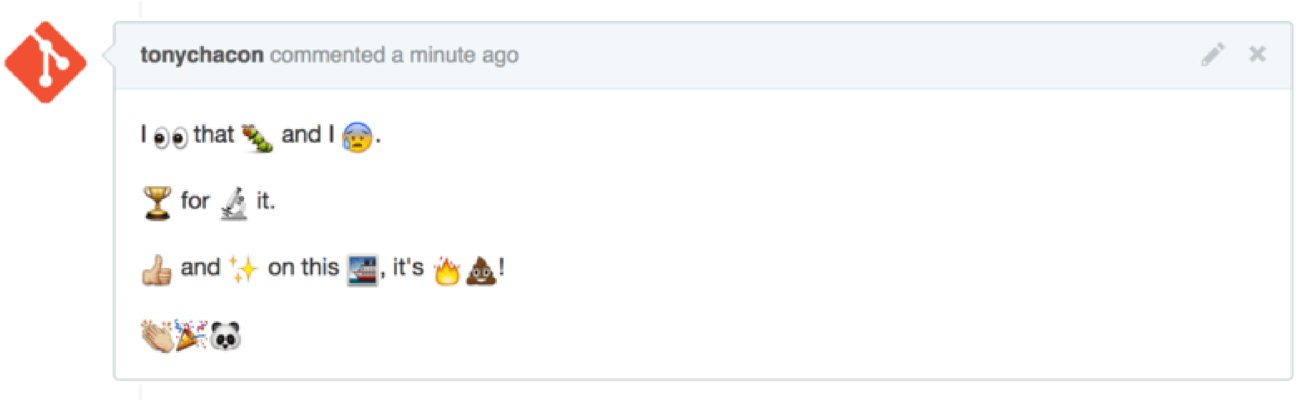


Figure 107. Интензивно ползване на emoji

Не че emoji-тата са нещо толкова полезно, но добавят елемент на забавление и разведряване в среда, в която това иначе е трудно постижимо.

Съществуват солиден брой уеб услуги, които осигуряват emoji символи. Можете да погледнете тук:

https://www.webfx.com/tools/emoji-cheat-sheet/

Изображения

Чисто технически, това не е част от GitHub Flavored Markdown, но възможността за лесно вмъкване на картинки е полезна функция в GitHub. Вместо да добавяте Markdown линкове към изображения, което може да е доста досадно, сайтът позволява да влачите и пускате изображенията директно в текстовите кутии.

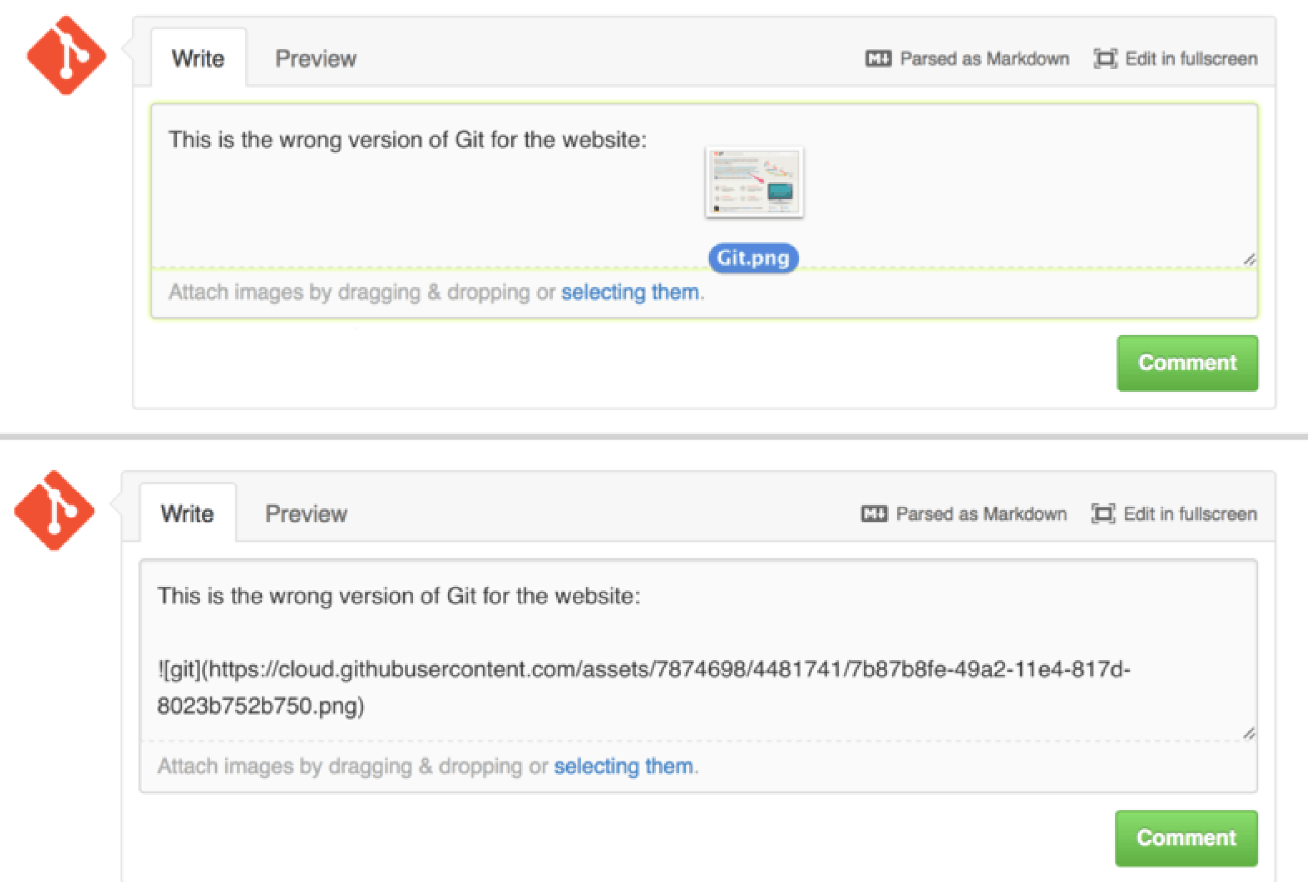


Figure 108. Вмъквайте изображения директно с Drag and drop и автоматично вграждане

Ако погледнете Вмъквайте изображения директно с Drag and drop и автоматично вграждане, може да видите малката подсказка “Parsed as Markdown” над текстовата област. Щракайки върху нея, можете да видите пълен спомагателен списък на всичко, което можете да правите с Markdown в GitHub.

Актуализиране на вашето публично GitHub хранилище

Веднъж след като fork-нете GitHub хранилище, вашето хранилище (или вашия "fork") съществува независимо от оригинала. В частност, когато в оригиналното хранилище се появят нови къмити, GitHub ви информира за това със съобщение от рода на:

```
This branch is 5 commits behind progit:master.
```

Но вашето клонирано копие никога няма да бъде автоматично обновено от GitHub - това е нещо, което трябва да направите сами. За щастие не е трудно.

Едната възможност да се направи не изисква конфигурация. Например, ако сте клонирали

от https://github.com/progit/progit2.git, можете да поддържате master клона актуален така:

```
$ git checkout master ①
$ git pull https://github.com/progit/progit2.git ②
$ git push origin master ③
```

① Ако сте на друг клон, върнете се към master.

② Изтеглете промените от https://github.com/progit/progit2.git и ги слейте в master.

③ Изпратете master клона към origin.

Това работи, но е малко досадно да въвеждате fetch URL-а всеки път. Може да автоматизирате процеса с малко конфигурация:

```
$ git remote add progit https://github.com/progit/progit2.git ①
$ git branch --set-upstream-to=progit/master master ②
$ git config --local remote.pushDefault origin ③
```

① Добавяте изходното хранилище и му давате име. Тук сме избрали то да е progit.

② Настройвате master клона си да тегли от отдалеченото хранилище progit.

③ Дефинирате origin като подразбиращо се push хранилище.

След като сте направили това, последователността става много по-проста:

```
$ git checkout master ①
$ git pull ②
$ git push ③
```

① Ако сте в друг клон се връщате в master.

② Изтегляте промените от progit и ги сливате в клона master.

③ Изпращате локалния master клон към origin.

Този подход може да е удобен, но си има и недостатъци Git с готовност ще свърши тази работа за вас, но няма да ви предупреди ако направите къмит в master, изтеглите от progit и след това публикувате в origin — всички тези операции са напълно валидни при такава ситуация. Така че, трябва да внимавате никога да не къмитвате директно в master, защото този клон ефективно принадлежи към upstream хранилището.

Управление на проект

След като вече сме готови да сътрудничим в проекти на други хора, ще погледнем и обратната страна: създаване, поддържане и администриране на собствен проект.

Създаване на хранилище

Нека създадем ново хранилище, чийто код да споделим с другите. Започваме натискайки

бутона “New repository” в дясната част на екрана или натискайки бутона + в горния тулбар до потребителското ни име както можем да видим в Падащият списък “New repository”.

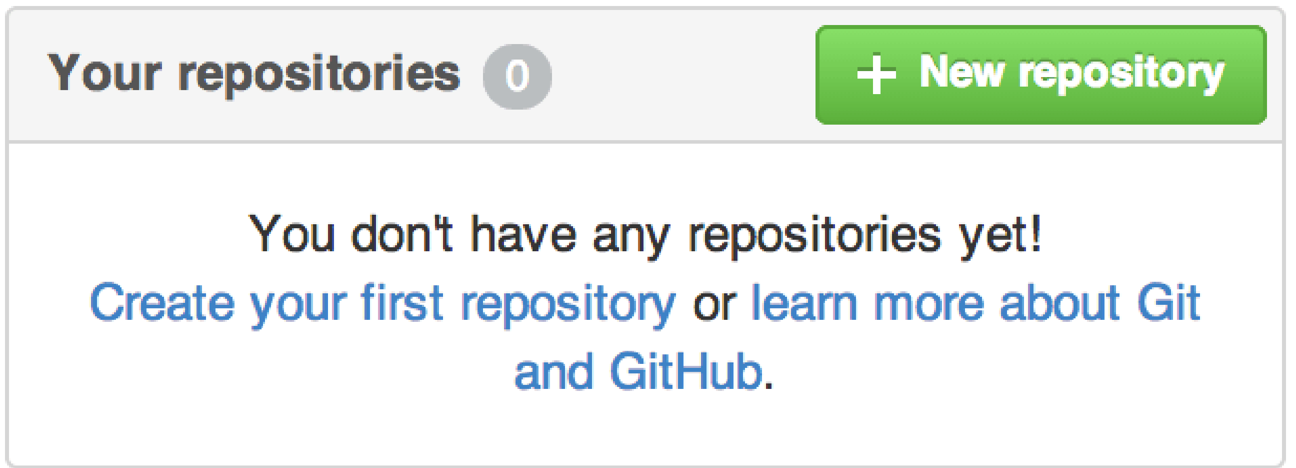


Figure 109. Частта “Your repositories”

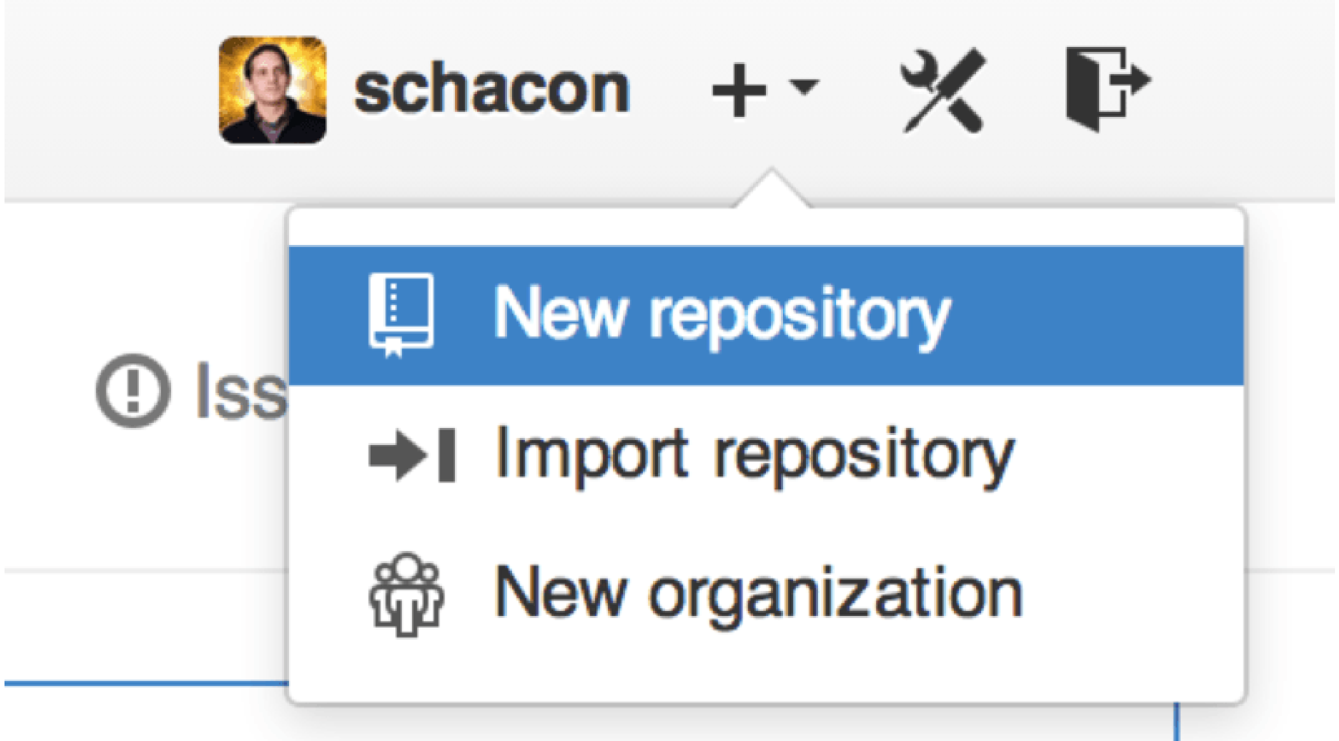


Figure 110. Падащият списък “New repository”

Това ни прехвърля към формата “new repository”:

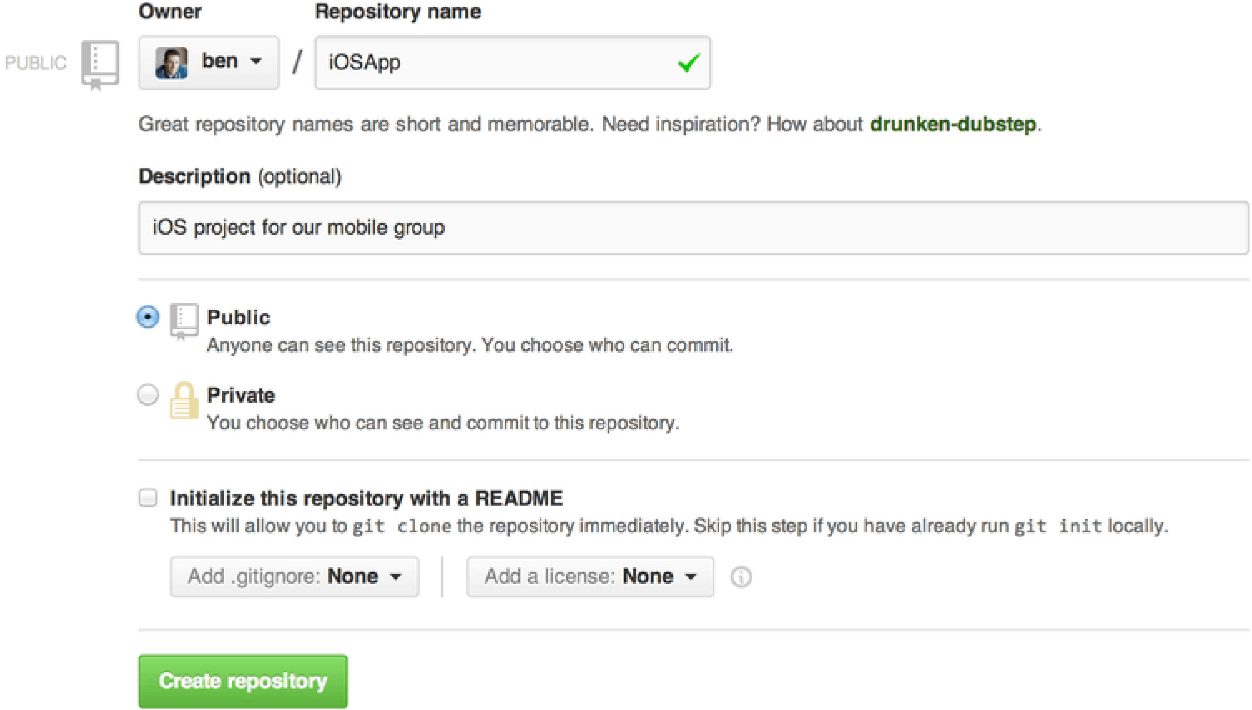


Figure 111. Формата "new repository"

На практика, единственото задължително поле е това с името на проекта, всички останали са незадължителни. Засега просто натиснете бутона "Create Repository" и вече разполагате с ново хранилище в GitHub, с име `<user>/<project_name>`.

Понеже все още нямате качен никакъв код, GitHub ще ви предложи инструкции как да създадете ново Git хранилище или да се свържете със съществуващ Git проект. Няма да навлизаме в детайли за това, ако имате нужда от припомняне, погледнете Основи на Git.

Сега проектът ви се хоства в GitHub и можете да изпратите URL-а на всеки, с който желаете да го споделите. Всеки GitHub проект е достъпен през HTTPS като `https://github.com/<user>/<project_name>`, а също и през SSH като `git@github.com:<user>/<project_name>`. Git може да тегли и да изпраща и по двата начина, а достъпът се контролира с правата на името и паролата на свързващия се потребител.

Често се предпочита HTTPS-базиран достъп, понеже по този начин външният потребител може да клонира проект и без да има GitHub акаунт. Ако някой потребител предпочита SSH достъп, то той трябва да има акаунт и качен SSH ключ. HTTPS адресът е същият, който потребителят би написал в браузъра си за уеб базиран достъп до проекта.

Добавяне на сътрудници

Ако работите с други хора по проекта си и желаете да им дадете възможност да правят къмити, трябва да ги добавите към проекта като "collaborators". Ако Ben, Jeff, и Louise имат GitHub акаунти и искате да има дадете Push достъп до вашето хранилище, можете да ги добавите към проекта, така че да могат както да четат, така и да пишат в кода.

Натиснете линка "Settings" в дъното на дясната част.

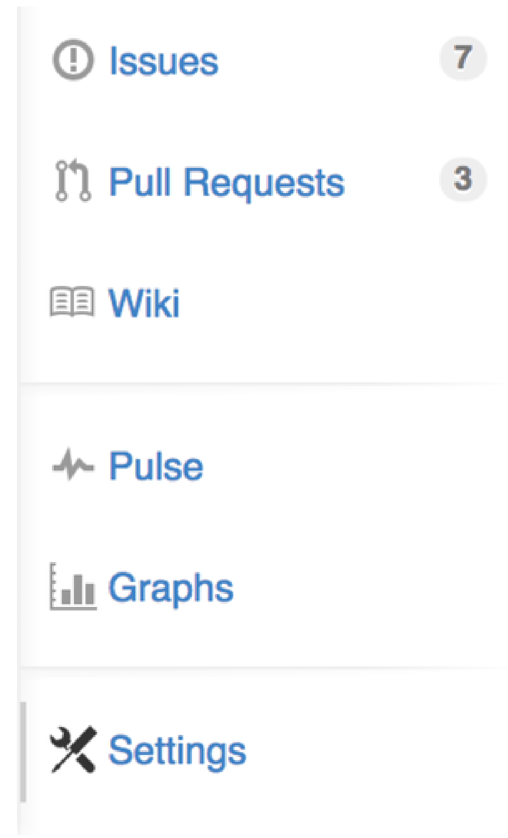


Figure 112. Препратката settings за хранилището

След това, изберете “Collaborators” от менюто вляво. Въведете потребителското име, което желаете и щракнете`‘Add collaborator.'' Можете да повторите това за колкото други потребителя желаете. Ако искате пък да отнемете достъпа, просто щракнете “X” иконата в дясната част на съответния ред.

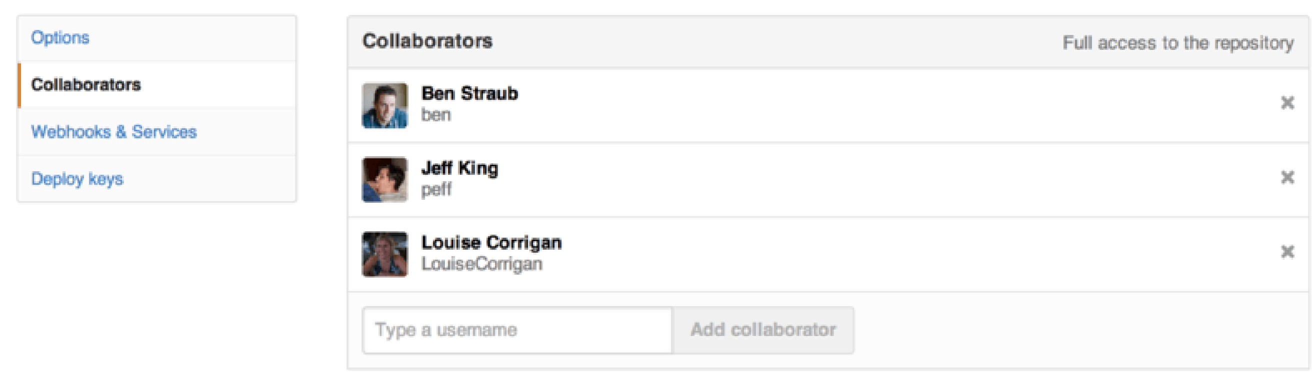


Figure 113. Сътрудници в хранилище

Управление на Pull Requests

Сега вече имате проект с код в него и може би няколко сътрудника с push достъп до хранилището - нека да видим какво да направите, когато получите Pull Request.

Pull Request-ите могат да дойдат от клон, който се намира във fork на проекта ви или пък от друг клон в същото хранилище. Единствената разлика е, че в клоновете на fork-натите хранилища нормално нямате достъп за писане (а и собствениците им нямат към вашите клонове), докато при вътрешните Pull Request-и обикновено и двете страни могат да пишат в клона.

За тези примери, нека приемем, че вие сте потребител “tonychacon” и сте създали нов Arduino проект наречен “fade”.

Email известяване

Някой се появява, променя част от кода ви и ви изпраща Pull Request. Ще получите електронна поща за новия Pull Request изглеждащ подобно на Email известяване за нов Pull Request.

Figure 114. Email известяване за нов Pull Request

Няколко неща са за отбелязване в този имейл. Първо, той съдържа малък diffstat — списък на файловете, в които има промени от Pull Request-а и в какво количество са те. След това, имате линк към Pull Request-а в GitHub. Предоставят ви се и няколко URL-а, които можете да ползвате от командния ред.

Ако виждате ред `git pull <url> patch-1`, това е прост начин да слеете отдалечен клон без да трябва да добавяте remote. Видяхме това в Извличане от отделечени клонове. Ако искате, можете да създадете и да превключите в topic клон и след това да изпълните тази команда за да слеете Pull Request-а.

Другите интересни URL-и са `.diff` и `.patch` URL-ите, които както можете да предположите, осигуряват unified diff и patch версии на Pull Request-а. Технически, можете да слеете работата в Pull Request-а примерно така:

```
$ curl https://github.com/tonychacon/fade/pull/1.patch | git am
```

Съвместна работа по Pull Request

Както видяхме в Работния процес в GitHub, сега можете да проведете дискусия с човека, който е пуснал Pull Request-а. Можете да коментирате специфични редове код, да коментирате цели къмити или целия Pull Request, използвайки GitHub Flavored Markdown където искате.

Всеки път, когато някой друг коментира Pull Request-а, ще продължавате да получавате имейл нотификации, така че да сте наясно какво се случва. Всеки от дискутиращите ще има линк към Pull Request-а и също така можете директно да отговорите на имейла пускайки автоматично коментар в Pull Request нишката в GitHub.

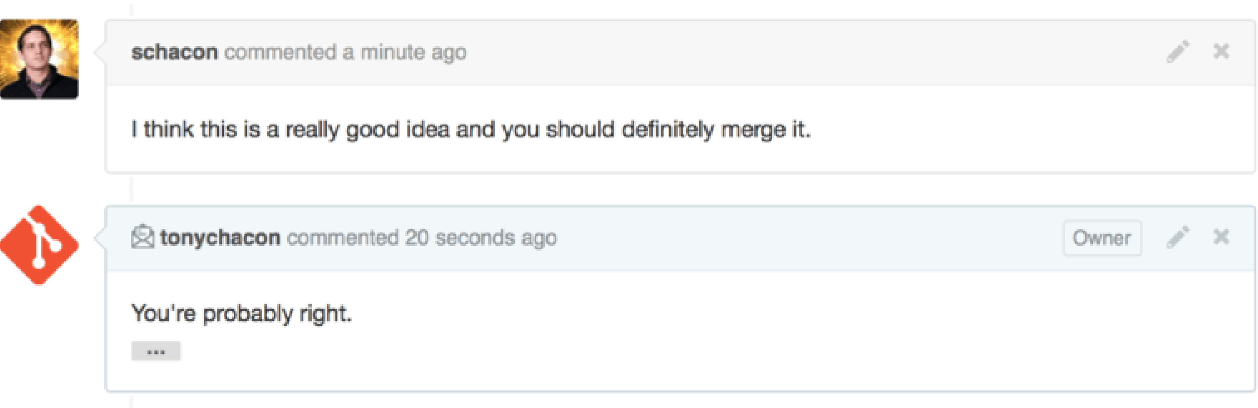


Figure 115. Отговорите на имейлите се показват в нишката

Веднъж след като кодът е одобрен и искате да го слеете, можете или да го издърпате и слеете локално с помощта на `git pull <url> <branch>` синтаксиса, който видяхме по-рано, или да добавите fork-хранилището като remote, да го изтеглите и слеете.

Ако сливането е просто, можете направо да натиснете бутона “Merge” в GitHub. Това ще направи “non-fast-forward” сливане с merge commit, дори и ако е възможно fast-forward сливане. Всеки път когато използвате бутона, винаги се създава merge commit независимо от обстоятелствата. Както можете да видите в Merge бутон и инструкции за сливането на Pull Request-а ръчно, GitHub ви дава цялата тази информация ако натиснете помощната препратка.

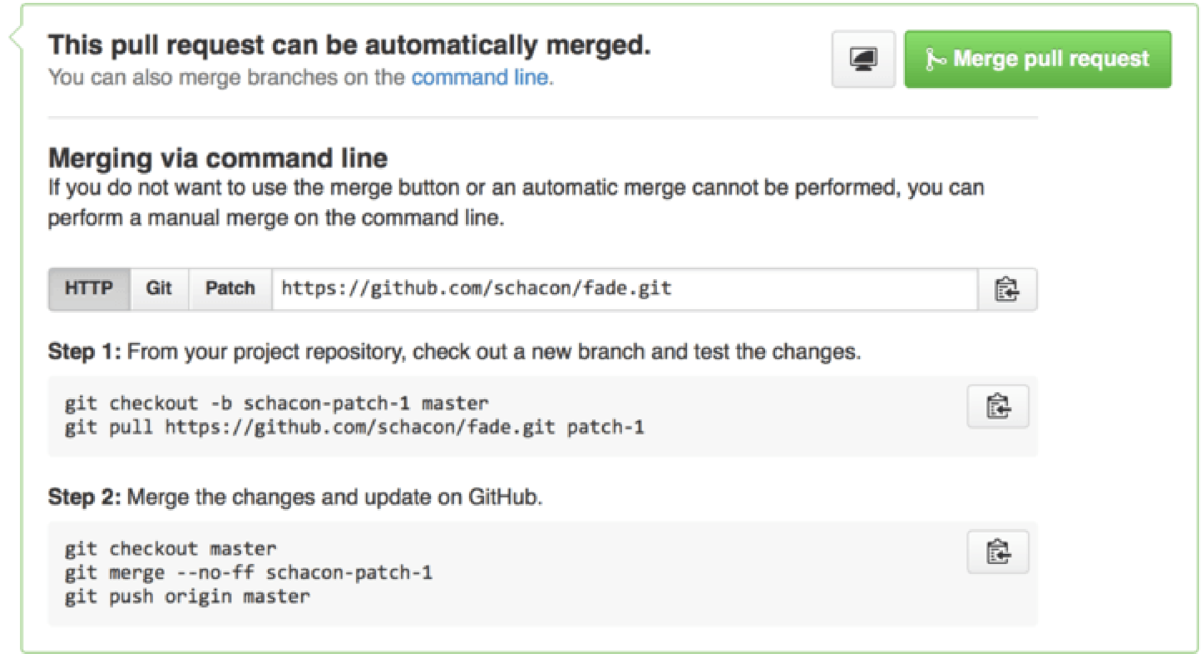


Figure 116. Merge бутон и инструкции за сливането на Pull Request-а ръчно

Ако решите, че не искате да слеете, можете просто да затворите Pull Request-а и човекът, който го е стартирал ще бъде уведомен.

Pull Request референции

Ако си имате работа с **много** Pull Request-и и не искате да добавяте цял куп remotes или да правите еднократни изтегляния всеки път, GitHub ви предоставя един хитър трик за улеснение в работата. Това е материал за напреднали и ще видим детайлите за него в Refspec спецификации, но може да е много полезен.

GitHub в действителност представя Pull Request клоновете за дадено хранилище като вид псевдо-клонове на сървъра. По подразбиране, вие не ги получавате при клониране, но те са там по един маскиран начин и можете да получите достъп до тях лесно.

За да демонстрираме това, ще използваме low-level команда (често наричана “plumbing” команда, за която ще научим повече в Plumbing и Porcelain команди) наречена `ls-remote`. Обикновено тази команда не се използва ежедневно в Git операциите, но е полезна защото ни показва какви референции съществуват на сървъра.

Ако стартираме тази команда за “blink” хранилището, което ползвахме по-рано, ще видим списък от всички клонове, тагове и други референции в него.

```
$ 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
```

Разбира се, ако сте във вашето хранилище и изпълните `git ls-remote origin` или кой да е друг remote, ще видите отпечатан изход подобен на този.

Ако хранилището е в GitHub и имате отворени Pull Request-и, ще получите тези референции с префикс `refs/pull/`. Това по същество са клонове, но понеже не са под `refs/heads/`, нормално не ги получавате когато клонирате или изтегляте от сървъра — fetching процесът по подразбиране ги игнорира.

Има по две референции на Pull Request - едната която завършва на `/head` сочи към точно същия къмит както и последния къмит в Pull Request клона. По този начин, ако някой отвори Pull Request в наше хранилище и клонът му се казва `bug-fix`, сочещ към къмит `a5a775`, тогава в **нашето** хранилище ние няма да имаме `bug-fix` клон (той е във fork-a), но *ще имаме* `pull/<pr#>/head` референция сочеща към `a5a775`. Това означава, че можем лесно да изтеглим всеки Pull Request клон в една стъпка без да трябва да добавяме множество remotes.

Сега можем да изтеглим референцията директно.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

Това инструктира Git да се свържи с `origin` адреса и да изтегли референцията наречена `refs/pull/958/head`. Git за щастие следва инструкцията и сваля всичко необходимо за конструирането на тази референция, след което поставя указател към къмита, който искате в `.git/FETCH_HEAD`. Можете да продължите с `git merge FETCH_HEAD` в клон, в който да тествате, но merge commit съобщението изглежда леко странно. Също така, ако разглеждате **много** Pull Request-и, това става досадно

Съществува също начин да изтеглите *всички* Pull Request-и и да ги актуализирате всеки път, когато се свързвате с отдалечения сървър. Отворете файла `.git/config` и потърсете `origin` секцията. Ще изглежда по подобен начин:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Редът , който започва с `fetch =` се нарича “refspec.” Това е начин за съотнасяне на имена в сървъра с имена в локалната ви `.git` директория. В примера тук, това казва на Git, "нещата на сървъра, които се намират в `refs/heads` трябва да се съхраняват в локалното ми хранилище в `refs/remotes/origin`." Можете да промените тази секция за да добавите друг refspec:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Това инструктира Git че, “всички refs изглеждащи като `refs/pull/123/head` трябва да се съхраняват локално като `refs/remotes/origin/pr/123`.” Сега, ако запишете файла и изпълните `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
# …
```

Сега всички отдалечени Pull Request-и се представят локално като refs, които работят като tracking клонове - те са само за четене и се обновяват когато теглите. Това прави много лесно изпробването на код от Pull Request локално:

```
$ 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'
```

Наблюдателните читатели ще забележат надписа `head` в края на remote частта на refspec. Съществува също и `refs/pull/#/merge` референция от страна на GitHub, която представлява къмита, който ще се създаде ако натиснете бутона “merge” в сайта. Това може да ви позволи да тествате сливането преди още да сте натиснали бутона.

Pull Requests на Pull Requests

Можете да създавате Pull Request-и насочени не само към главния (`master`) клон, а към всеки един клон в мрежата. В действителност, можете да ги насочите и към друг Pull Request.

Ако забележите Pull Request, който се развива в добра посока и имате идея за подобряване на кода в него, или пък просто нямате push достъп до желания клон, можете да отворите отделен Pull Request директно към него.

Когато започвате да правите Pull Request, в горната част на екрана има кутия, указваща от и към кой клон опитвате да го насочите. Ако натиснете бутона “Edit” вдясно от кутията,

можете да смените не само клоновете, но и fork-а.

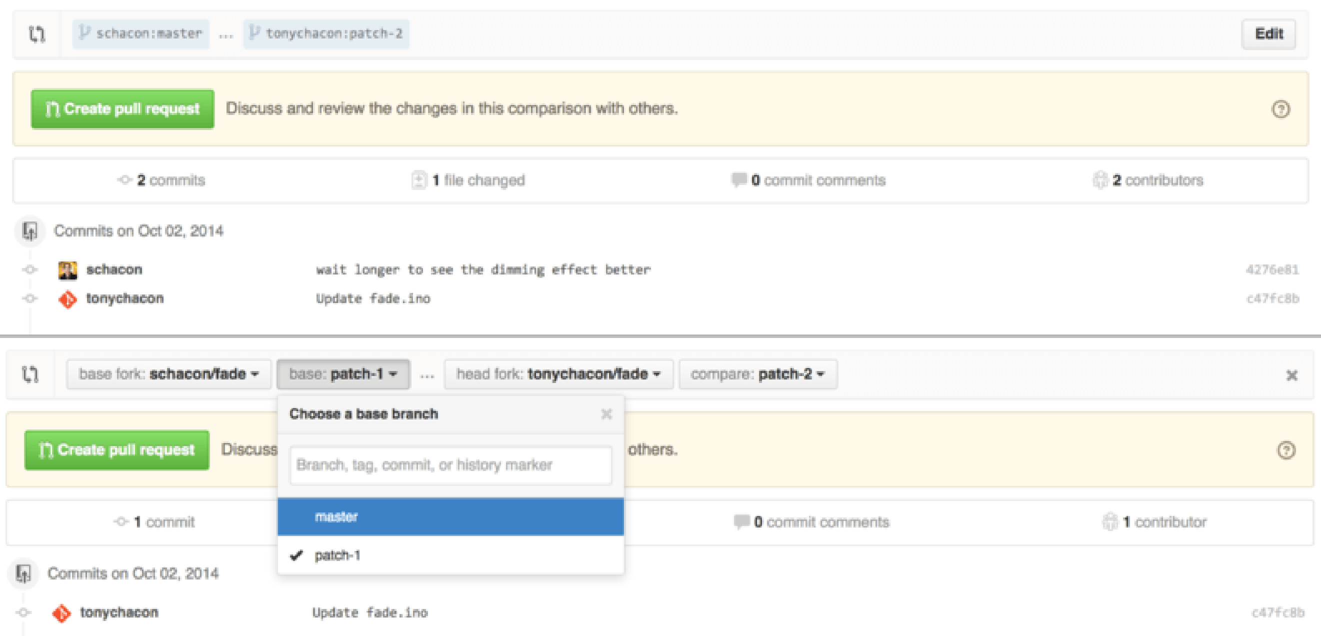


Figure 117. Ръчна смяна на целевия клон и fork на Pull Request

Тук можете сравнително лесно да слеете вашия нов клон в друг Pull Request или друг fork на проекта.

Бележки и уведомления

GitHub също така има удобна система за нотификации, която е полезна, когато имате въпроси или ви трябва мнение от специфични разработчици или екипи.

Във всеки коментар, можете да въведете символа `@` и ще получите autocomplete списък с имената и потребителските имена на сътрудниците в проекта.

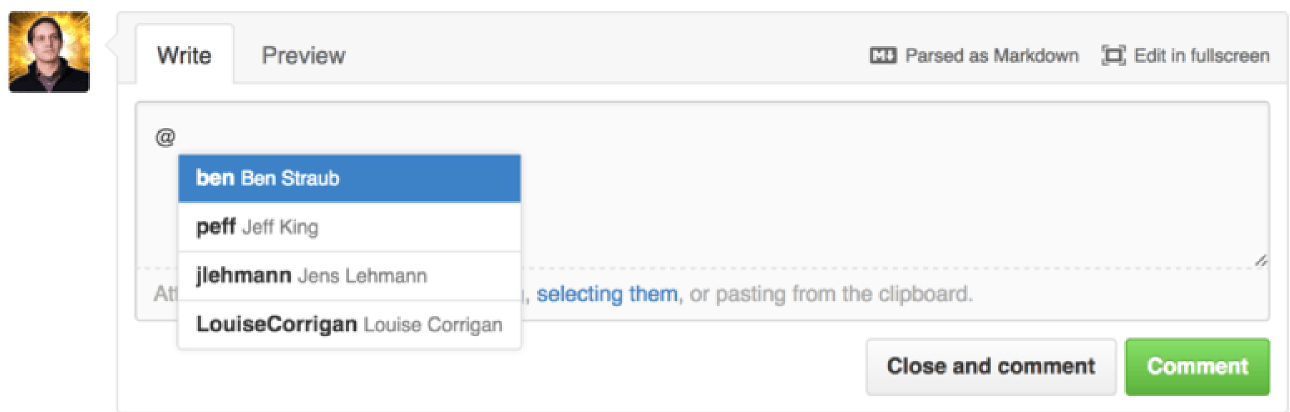


Figure 118. Въведете @ за да упоменете някого

Можете да упоменете и потребител, който не се показва в списъка, но често списъкът ускорява нещата.

След като веднъж публикувате коментар с упоменат по горния начин потребител, той ще бъде уведомен за това. Това е доста ефективен начин да въведете хора в дискусия, вместо да разчитате те да я наблюдават. Много често в Pull Request-ите в GitHub разработчиците

използват този похват, за да привлекат вниманието на колегите си към даден Issue или Pull Request.

Ако някой е бил упоменат в Pull Request или Issue, той ще бъде “абониран” към тях и ще бъде осведомяван своевременно за възникнала активност. Ще бъдете абонирани също така за всичко, което сте стартирали, за събития в наблюдавани хранилища или когато коментирате нещо. Ако не желаете да получавате нотификации, в съответната страница има бутон “Unsubscribe”, с който да се отпишете от уведомяванията свързани с нея.

Figure 119. Отписване от нотификации за Issue или Pull Request

Страницата за уведомления

Под “нотификации” в смисъла на GitHub имаме предвид специфичния подход, който сайтът използва за да поддържа връзка с вас при възникнали събития. Съществуват няколко различни начина за тяхната настройка. Ако отворите секцията “Notification center” от страницата с настройки можете да видите част от наличните опции.

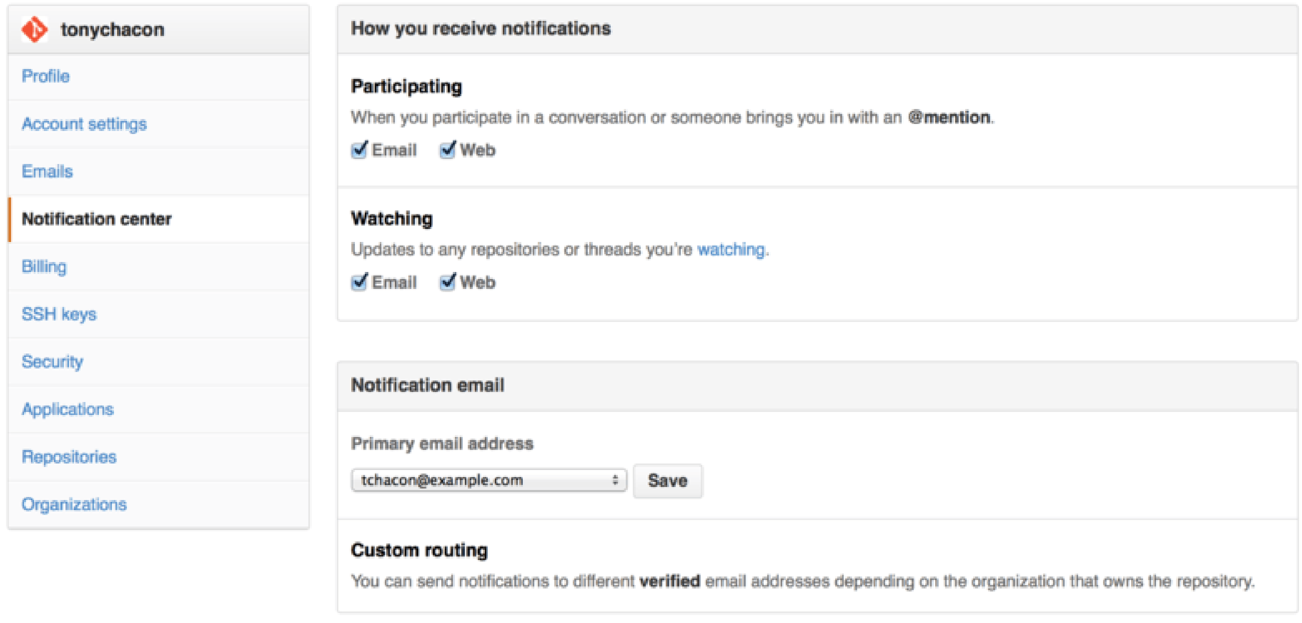


Figure 120. Опции за известяване

Двата избора за получаване на известия са през “Email” и “Web” и може да изберете кой да е от двата, двата едновременно или пък нито един

Web известяване

Web известяванията се отнасят само до сайта GitHub и можете да ги виждате само там. Ако в предпочитанията си сте избрали тази опция и възникне касаещо ви събитие, ще видите малка синя точка над иконата за нотификации в горната част на екрана както е показано в Център за известия.

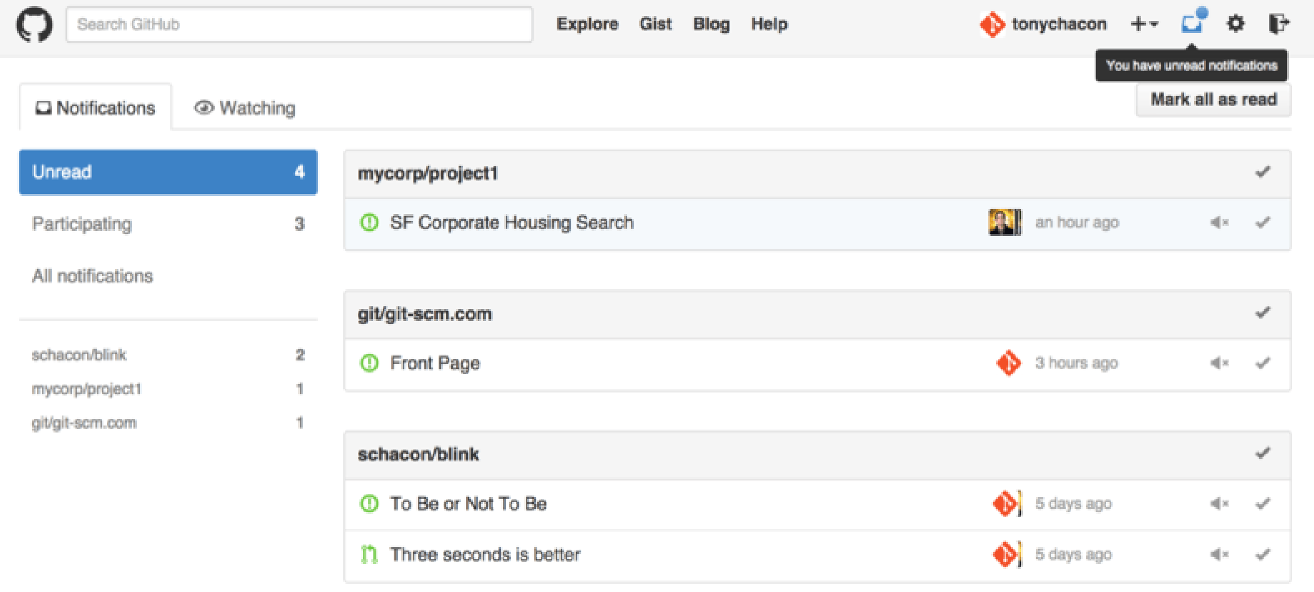


Figure 121. Център за известия

Ако щракнете върху нея, ще се покаже списък с всички уведомления за вас, групирани по проект. Можете да филтрирате по конкретен проект щракайки върху името му в лявата лента. Можете да маркирате уведомлението като прието с чекбокс иконата до него или да

маркирате *всички* като приети с чекбокса в горната част на групата. Съществува и бутон mute до всеки чекбокс, чрез който да укажете, че не желаете повече уведомления по съответната нотификация.

Всички тези инструменти са полезни при работа с голям брой известия. Много от опитните GitHub потребители изключват изцяло имейл уведомленията и управляват известията си през този екран.

Email известяване

Email опцията е алтернативен начин за управление на известяванията през GitHub. Ако я използвате, ще получавате поща за всяко известяване. Видяхме примери за това в Коментарите изпратени като имейл уведомление и Email известяване за нов Pull Request. Пощите също така ще бъдат правилно подредени в нишки, което е чудесно ако използвате threading съвместим пощенски клиент.

Хедърите на имейл съобщенията съдържат съответните метаданни, което е полезно за създаването на специфични правила и филтри за обслужването им от клиента.

Например, ако разгледаме хедърите на писмото до Tony показано в Email известяване за нов Pull Request, ще забележим следното:

```
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
```

Тук има няколко интересни неща. Ако искате да осветите или препратите имейлите от този конкретен проект или дори Pull Request, информацията в хедъра `Message-ID` ви дава всички данни във формата `<user>/<project>/<type>/<id>`. Ако например пощата се отнася до даден Issue, тогава `<type>` полето ще е "issues" вместо "pull".

`List-Post` и `List-Unsubscribe` хедърите помагат на съвместимите имейл клиенти бързо да изпратят отговор или да се отпишат от дискусията. По същество ефектът е същия както ако натиснете "mute" бутона в сайта или "Unsubscribe" в Issue или Pull Request страница.

Ако сте активирали и двата вида уведомявания и прочетете имейл версията за дадено събитие, то уеб версията му също ще бъде маркирана като прочетена (ако имейл клиентът ви разрешава картинки).

Специални файлове

Съществуват няколко специални файла, които GitHub ще забележи, ако присъстват в хранилището ви.

README

Първият е `README` файла, който може да се форматира във всеки формат, който GitHub би разпознал. Например, може да е `README`, `README.md`, `README.asciidoc`, и т.н. Ако GitHub види README файл, ще го рендерира на началната страница на проекта.

Много екипи ползват това за представяне на най-важните аспекти от проекта пред незапознатите с него потребители. Това би могло да включва:

- Каква е целта на проекта
- Как се конфигурира и инсталира
- Пример за това как се ползва и пуска
- Лицензът, под който се публикува
- Указания за сътрудничество в него

Понеже GitHub рендерира този файл, можете да вмъквате линкове (вкл. и към изображения) за допълнително улеснение на разглеждащия.

CONTRIBUTING

Друг специален файл, който GitHub разпознава е `CONTRIBUTING` файла. Ако имате файл с това име и произволно разширение, то GitHub ще покаже Създаване на Pull Request при наличен CONTRIBUTING файл, когато някой започне да създава Pull Request.

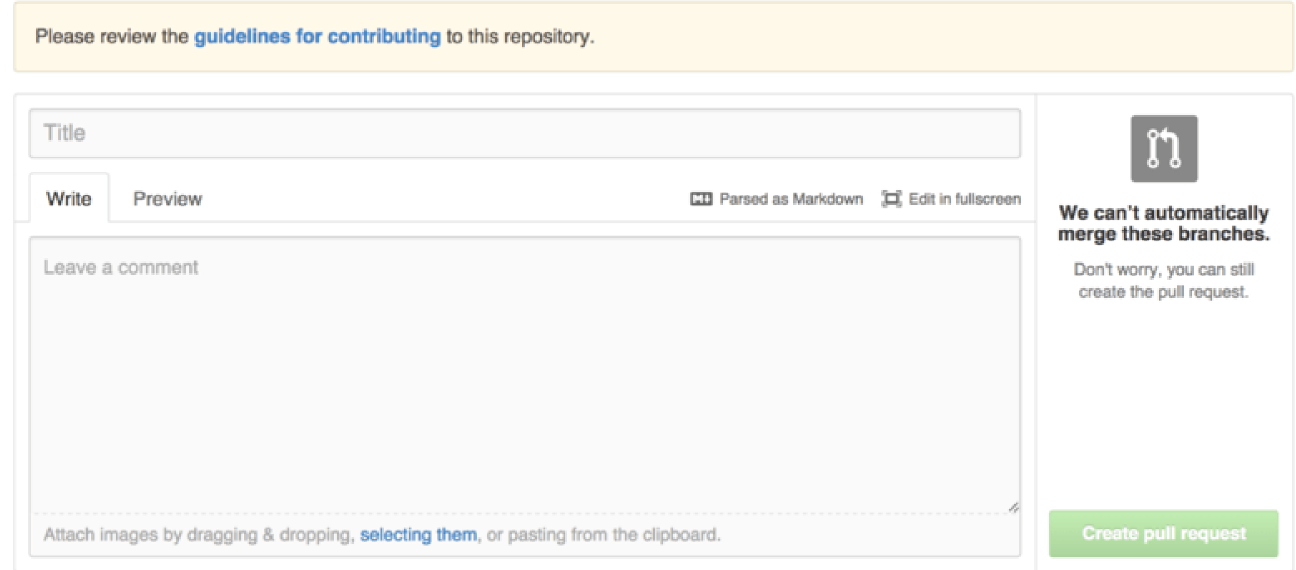


Figure 122. Създаване на Pull Request при наличен CONTRIBUTING файл

Идеята тук е да покажете специфичните неща, които искате или не желаете да присъстват в Pull Request-ите към проекта ви. Така потребителите могат да прочетат изискванията ви преди да създадат Pull Request.

Администриране на проект

В общи линии броят на административните действия за единичен проект не е голям, но

има няколко интересни опции.

Смяна на клона по подразбиране

Ако ползвате име на клон по подразбиране различно от “master” за основния клон, можете да укажете това в секцията “Options” в страницата с настройки за хранилището.

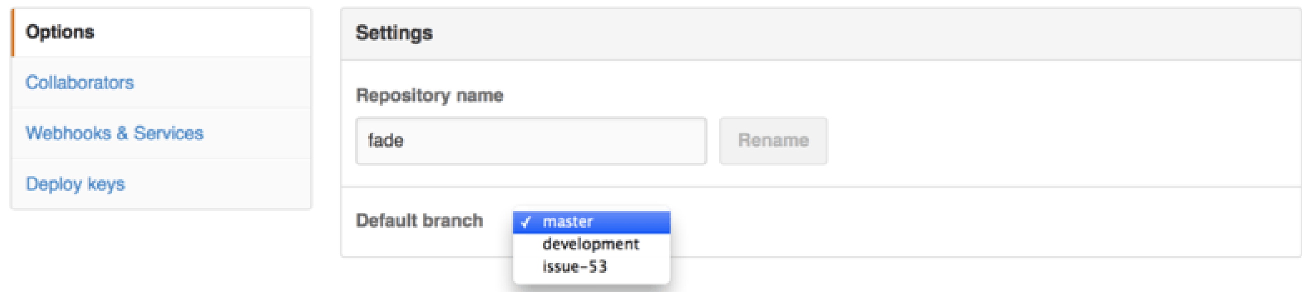


Figure 123. Смяна на клона по подразбиране за проект

Просто сменете клона по подразбиране от падащия списък и той ще стане такъв за всички главни действия занапред, включително за това от кой клон ще се извлекат файловете на проекта когато някой клонира хранилището.

Трансфер на проект

Ако искате да прехвърлите проекта си към друг потребител или организация в GitHub, разполагате с опция “Transfer ownership” в долната част на същата “Options” секция от настройките на проекта.

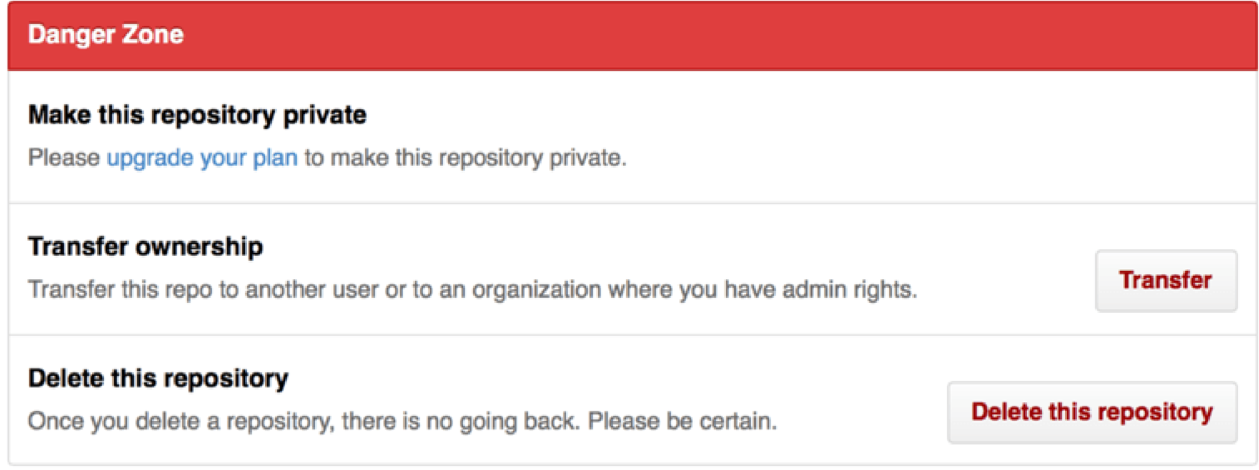


Figure 124. Трансфер на проект към друг GitHub потребител или организация

Това е полезно, ако прекратявате участието си в проект и някой друг иска да продължи поддръжката му или пък ако даден проект стане твърде мащабен и бихте желали да го преместите в организация.

Това не само ще премести хранилището заедно с всичката информация за него към друго място, но също така и ще бъде създаден redirect URL към новото място. Клониранията и изтеглянията от Git също ще бъдат пренасочени, а не само за уеб заявките.

Управление на организация

Освен акаунти за единични потребители, GitHub поддържа работа в организации. Подобно на персоналните акаунти, организациите също разполагат със собствен namespace в който се съхраняват проектите, само че много други неща са различни. Тези акаунти представят група от хора с поделен достъп до проекти и също така инструменти за управление на подгрупи от тези хора. Нормално подобни акаунти се използват от Open Source групи като "perl" или "rails" или пък от компании (като например "google" или "twitter").

Основи на организациите

Организация се създава лесно, просто натиснете иконата "+" в горната дясна част на всяка GitHub страница и изберете "New Organization" от менюто.

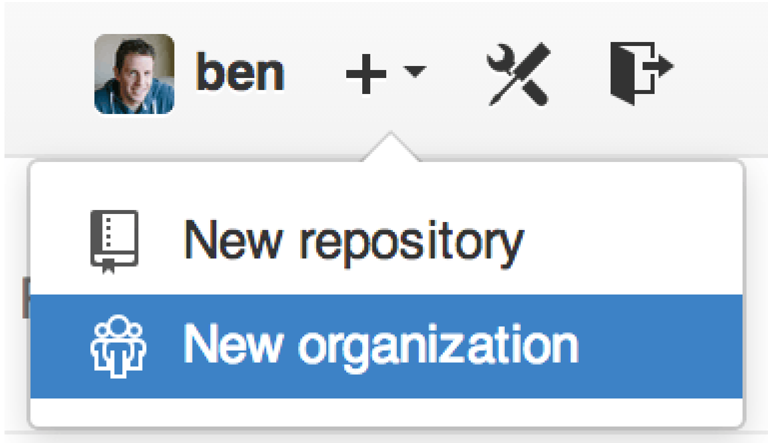


Figure 125. Командата "New organization"

Първо, ще трябва да дадете име за организацията и имейл адрес за контакт с групата. След това, можете да поканите други потребители като съсобственици на акаунта.

Следвайте тези стъпки и скоро ще сте собственик на чисто нова организация. Подобно на персоналните акаунти, организациите също са безплатни, ако планирате да съхранявате проекти с отворен код.

Като собственик на организация, когато после клонирате хранилище, ще ви бъде предоставена опция да изберете къде да го съхранявате - в namespace-а на организацията или на индивидуалния ви акаунт. Същият избор ще имате и при създаване на нови хранилища в GitHub. Също така, вие автоматично наблюдавате всички нови хранилища в организацията ви.

Както и в Вашият аватар, можете да качите аватар за организацията, за да я персонализирате. Както и при персоналните акаунти, организациите имат начална страница със списък на всички хранилища в нея.

Сега ще разгледаме няколко от разликите.

Екипи

Организациите са асоциирани с индивидуалните потребители посредством екипи, които

представляват логически групирани потребителски акаунти и хранилища в едно с правата, които всеки потребител има до съответните хранилища.

Например, да кажем че вашата организация има три хранилища: `frontend`, `backend`, and `deployscripts`. Може да искате вашите HTML/CSS/JavaScript разработчици да имат достъп до `frontend` и може би `backend`, а операторите ви да имат достъп до `backend` и `deployscripts`. Екипите правят тази организация на достъпа лесна, без да се налага да се управляват сътрудниците за всяко индивидуално хранилище.

Organization страницата показва опростен списък на всички хранилища, потребители и екипи, които организацията обединява.

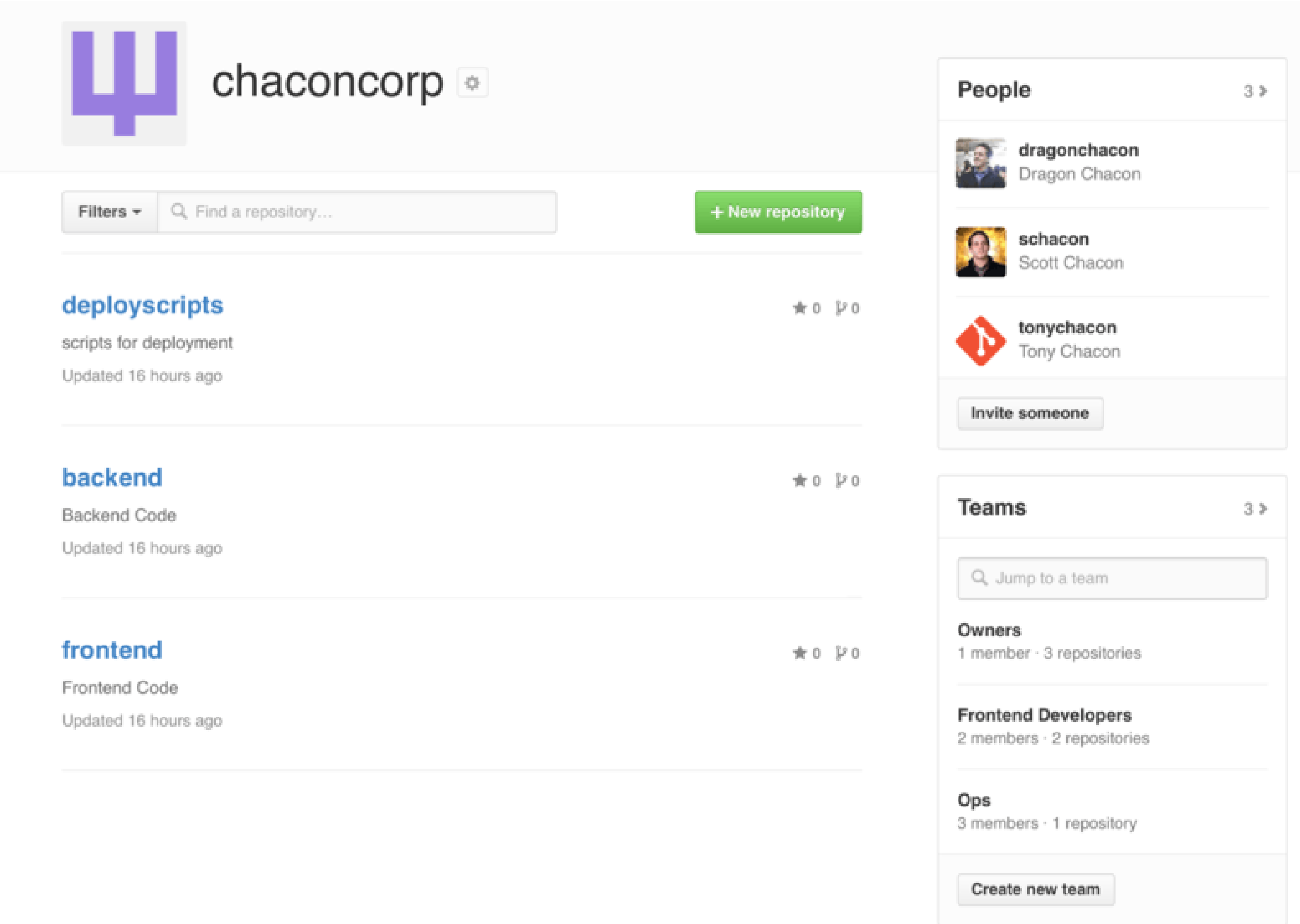


Figure 126. Organization страницата

За да управлявате вашите екипи, можете да използвате Teams лентата с инструменти вдясно на страницата в Organization страницата. Това ще ви покаже страница, в която можете да добавяте членове в екипа, да добавяте хранилища или да настройвате различни аспекти и нива на достъп за екипите. Всеки екип може да има права само за четене, за чете и писане, и за администрация на хранилищата. Можете да редактирате правата с бутона “Settings” в Страницата Team.

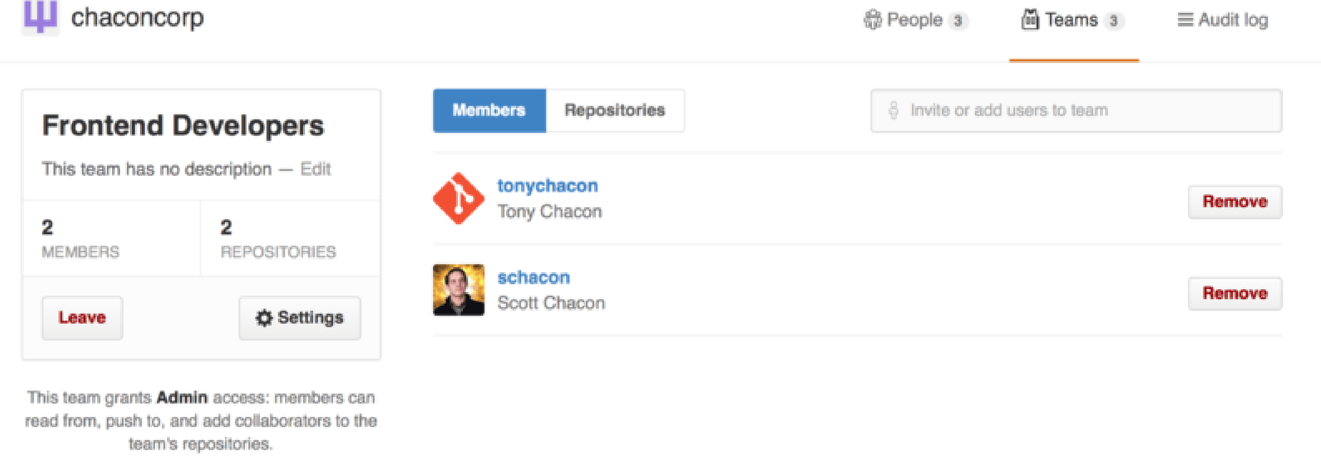


Figure 127. Страницата Team

Когато поканите някого в екип, той ще получи имейл с уведомление, че е поканен.

В допълнение, упоменаването в екипите (например `@acmecorp/frontend`) ще работи по същия начин както при индивидуалните потребители - с изключение на това, че **всички** членове на екипа са автоматично абонирани за дискусията. Това е полезно, ако се нуждаете от помощ от някого в екипа, но не знаете кого точно да питате.

Един потребител може да присъства в произволен брой екипи, така че не се ограничавайте само до access-control екипи. Специализираните екипи като `ux`, `css` или `refactoring` са полезни за определен вид въпроси, а други като `legal` и `colorblind` могат да ви помогнат по изцяло различна тема.

Audit Log

Организациите дават на собствениците си пълна информация за всичко, което се случва в тях. Можете да отворите секцията *Audit Log* и да видите какво се е случило на ниво организация, кой и откъде е извършвал промени.

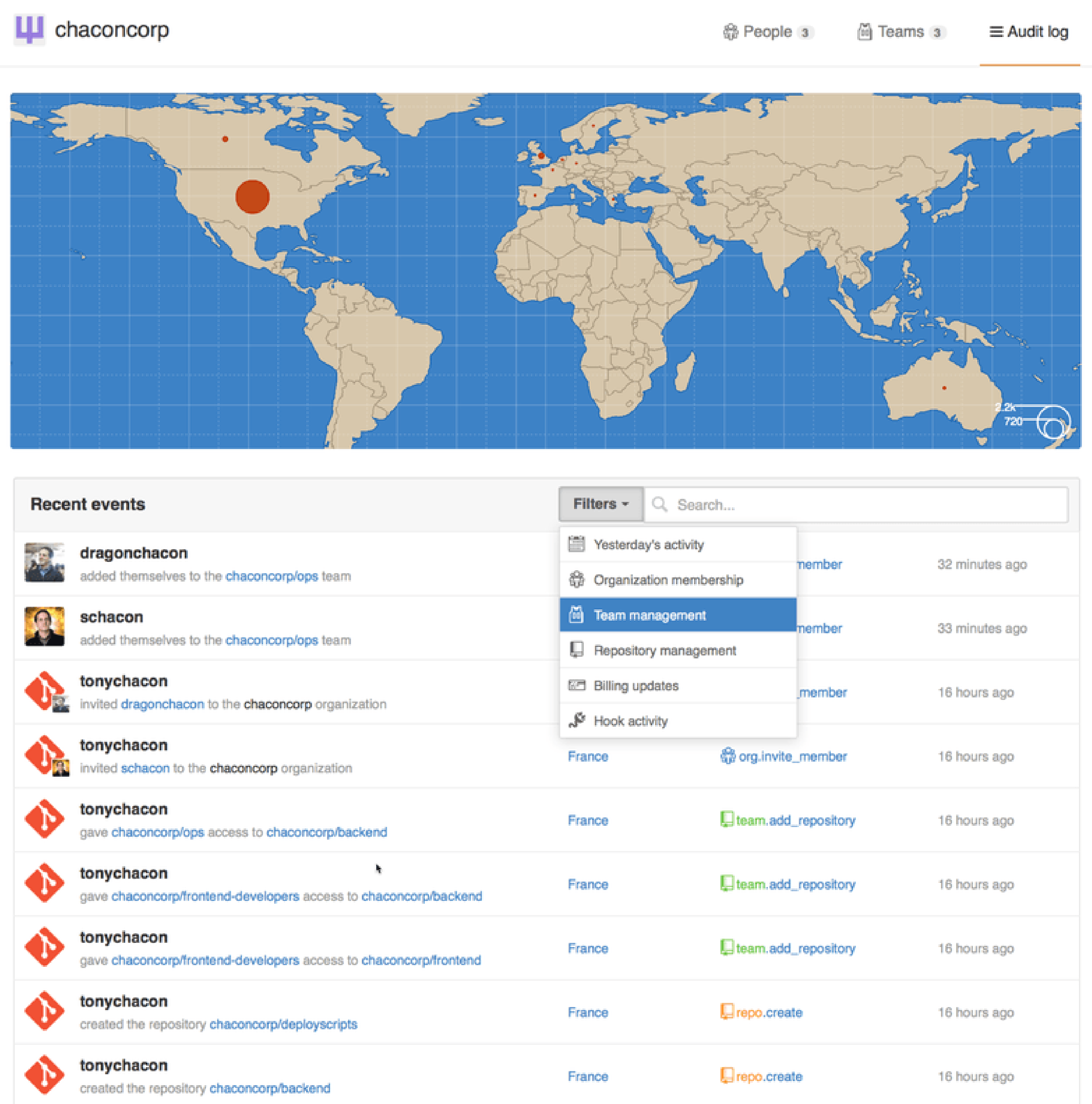


Figure 128. Audit log страницата

Можете да филтрирате по специфични типове събития, специфични места и хора.

Автоматизиране с GitHub

Разгледахме повечето функции и работни похвати в GitHub, но всеки голям проект или група от разработчици е твърде вероятно да искат да имат собствени настройки или интеграция с външни услуги.

За щастие, GitHub е доста гъвкав в това отношение. Тук ще разгледаме как се ползват hooks системата и API-то на GitHub.

Услуги и Hooks

Hooks and Services секцията в административната част на хранилище в GitHub е най-лесния

начин да накарате GitHub да комуникира с външни системи.

Services

Ще разгледаме първо Services. И Hooks и Services интеграциите могат да се намерят в Settings секцията на вашето хранилище, където преди гледахме при добавяне на сътрудници и смяна на клона по подразбиране за хранилището. В секцията “Webhooks and Services” ще видите нещо подобно на Services and Hooks конфигурационна секция.

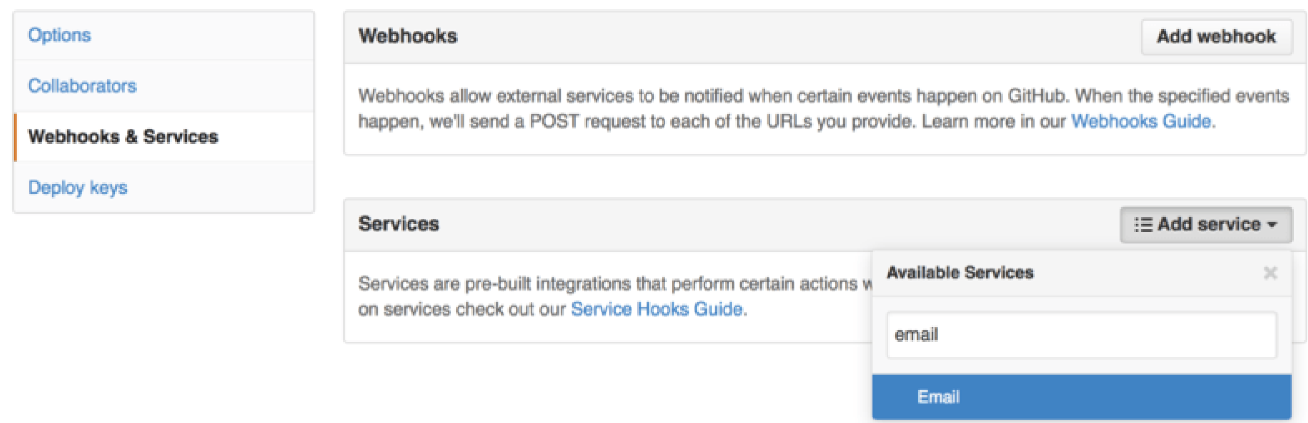


Figure 129. Services and Hooks конфигурационна секция

Може да избирате от десетки услуги, повечето от тях предлагащи интеграция с други комерсиални или open source системи. Сред тях са Continuous Integration услуги, услуги за проследяване на проблеми и грешки, chat room системи и системи за документация. Нека разгледаме настройката на една от най-простите, Email hook. Изберете “email” от падащия списък “Add Service” и ще попаднете на екран подобен на Email service конфигурация.

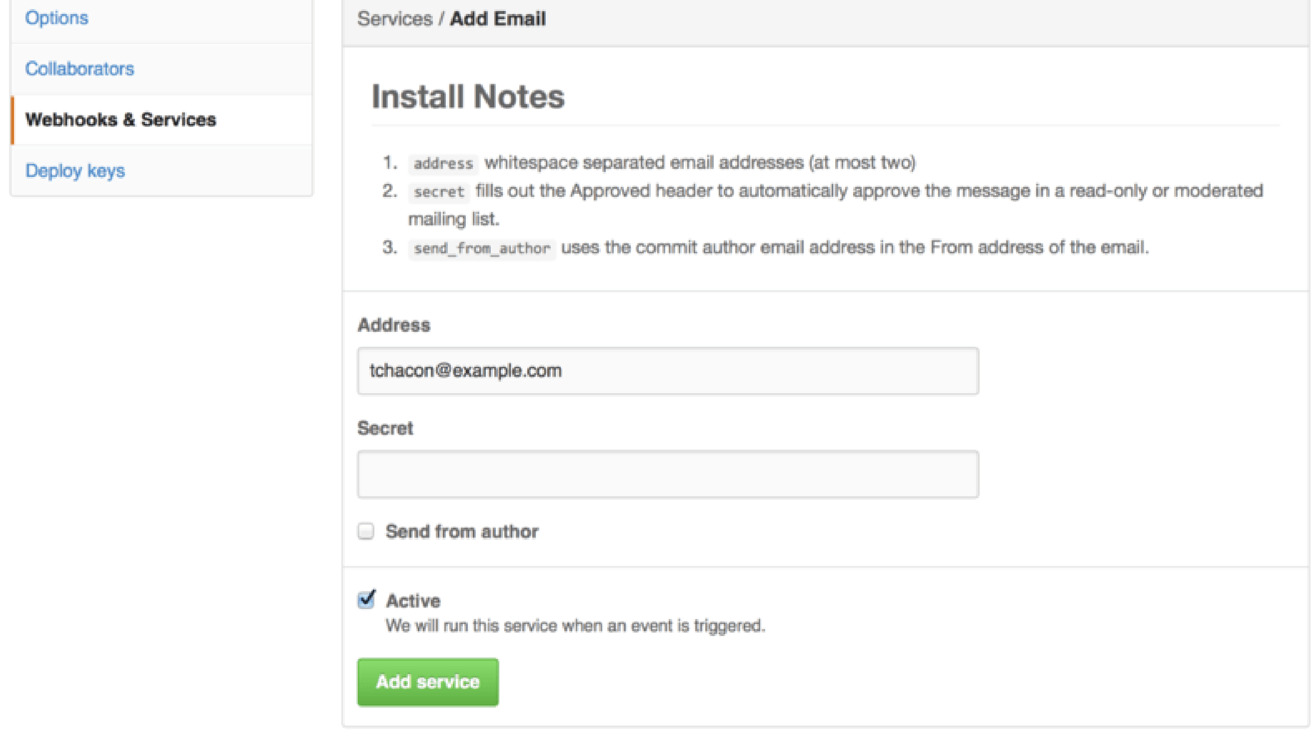


Figure 130. Email service конфигурация

В този случай, ако натиснете “Add service” бутона, имейл адресът който укажете ще получава съобщение всеки път, когато някой публикува код в хранилището. Услугите могат да слушат за много различни типове събития, но повечето от тях случат само за push събития и предприемат съответните действия с данните.

Ако ползвате система, която искате да интегрирате с GitHub, трябва да проверите тук дали не съществува налична интеграционна услуга. Например, ако ползвате Jenkins за тестване на код, можете да разрешите вградената Jenkins интеграция за да пуснете автоматично тест всеки път когато някой публикува код в хранилището ви.

Hooks

Ако ви трябва нещо по-специфично или желаете да интегрирате с услуга/сайт, които не присъстват в списъка на GitHub, можете да използвате по-общата hooks система. Hooks функционалността на GitHub е доста опростена. Указвате URL и GitHub ще изпрати HTTP заявка към него при всяко желано от вас събитие.

В общи линии, можете да пуснете малка уеб услуга, която да следи за hook-заявките от GitHub и да прави нещо с получените данни.

За да разрешите hook, натиснете “Add webhook” бутона в Services and Hooks конфигурационна секция. Ще видите страница подобна на Web hook конфигурация.

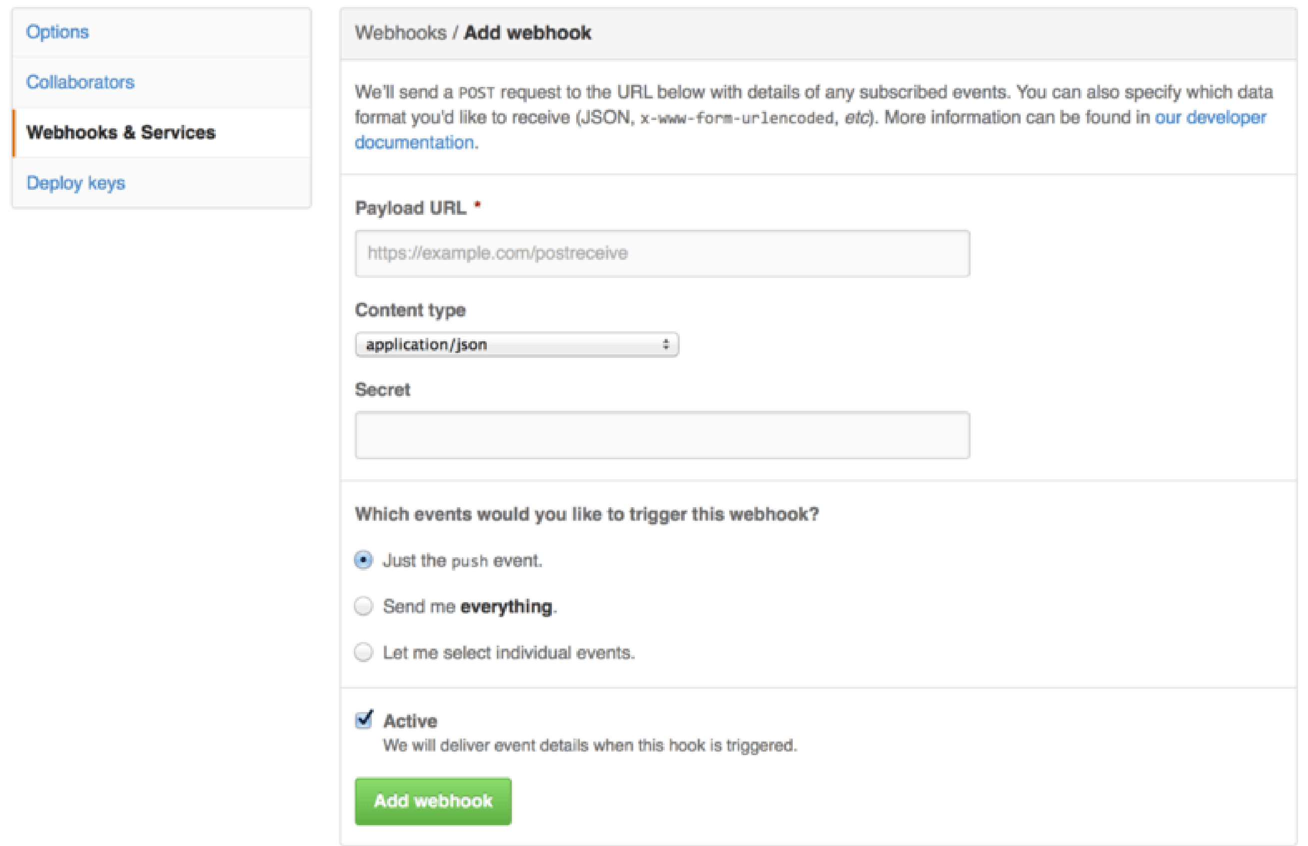


Figure 131. Web hook конфигурация

Конфигурацията е много опростена. В повечето случаи просто въвеждате URL и секретен ключ и натискате “Add webhook”. Има доста опции за това при кои събития искате GitHub да ви изпраща данни — по подразбиране това се случва само при `push` събития, когато някой

публикува промени в произволен клон от хранилището.

Нека видим малък пример за уеб услуга, която бихте могли да напишете за обслужване на данните от web hook. Ще използваме уеб framework системата Sinatra за Ruby, понеже тя е сравнително компактна и би следвало лесно да разберете какво вършим.

Нека приемем, че искаме да получаваме поща, ако специфичен потребител публикува код в конкретен клон и промени конкретен файл. Можем сравнително лесно да го направим така:

```
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
```

Тук прочитаме JSON данните от GitHub и търсим кой е направил публикуването, в кой клон и кои файлове са били модифицирани във всички публикувани къмити. След това проверяваме информацията според нашия критерий и изпращаме имейл, ако срещнем съвпадение.

За да разработите и тествате нещо от този род, разполагате с удобна developer конзола на същия екран, в който настройвате вашия hook. Можете да видите последните няколко съобщения, които GitHub се е опитал да направи за този webhook. За всеки hook може да

видите в детайли кога е бил изпратен, дали изпращането е успешно, както и тялото и хедърите за заявката и отговора при операцията. Това много улеснява тестването и дебъгването на вашите hooks.

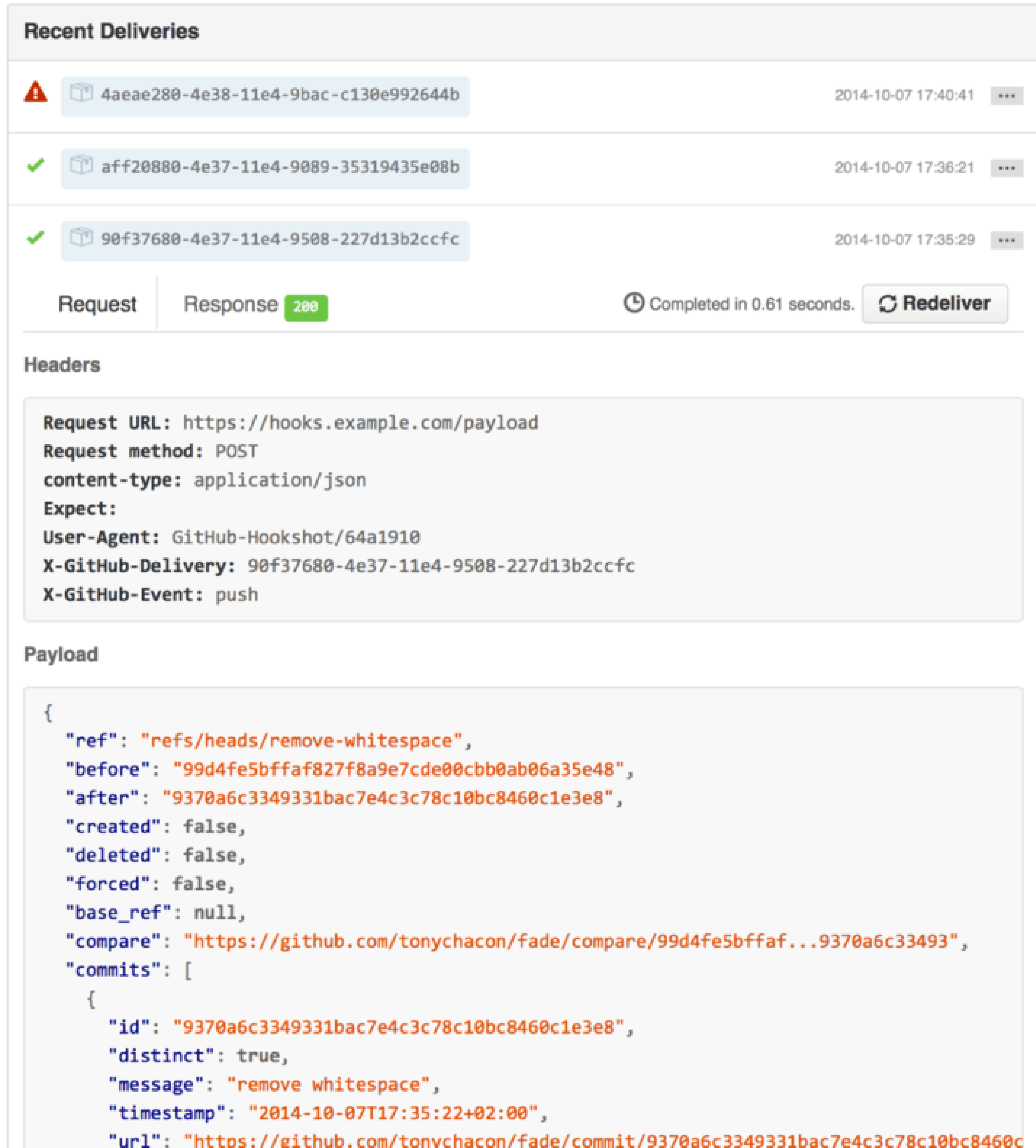


Figure 132. Web hook debugging данни

Друга чудесна особеност е, че можете да повтаряте изпращането на всяка заявка за да тествате уеб услугата си лесно.

За повече информация как да пишете webhooks и за всички възможни типове GitHub събития, погледнете документацията за разработчици на GitHub на адрес https://developer.github.com/webhooks/.

GitHub API

Services and hooks функционалността ви дава инструменти за получаване на push нотификация за събитията, които се случват във вашите хранилища, но какво да правите, ако искате повече информация за съответните събития? Например, бихте искали да автоматизирате неща като добавянето на сътрудници или маркирането на issues.

Тук е мястото да се възползвате от GitHub API. GitHub има голямо количество API endpoints позволяващи да правите почти всичко, което можете да правите в сайта, но по автоматизиран начин. В тази част ще видим как да се автентикираме и да се свържем с API интерфейса, как да коментираме по issue и как да променим статуса на Pull Request през API.

Основни действия

Най-простото нещо, което може да направите е да изпратите проста GET заявка към endpoint, който не изисква автентикация. Това може да е потребителска или read-only информация за open source проект. Например, ако искаме да знаем повече за потебителя "schacon", можем да изпълним нещо такова:

```
$ 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"
}
```

Има безброй подобни endpoints предоставящи информация за организации, проекти, issues, къмити — почти за всичко, което можете да видите публично в GitHub. Можете дори да използвате API за рендериране на специален Markdown или за да намерите `.gitignore` шаблон.

```
$ 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
https://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Коментар по Issue

Обаче, ако искате да направите определено действие в сайта, като коментар по Issue или Pull Request, или искате да погледнете или комуникирате с частно съдържание, ще се наложи да се автентикирате.

Съществуват няколко начина за това. Можете да ползвате базова автентикация с име и парола, но по-добра идея е да ползвате персонален token за достъп. Можете да го генерирате в секцията “Applications” на страницата с настройки.

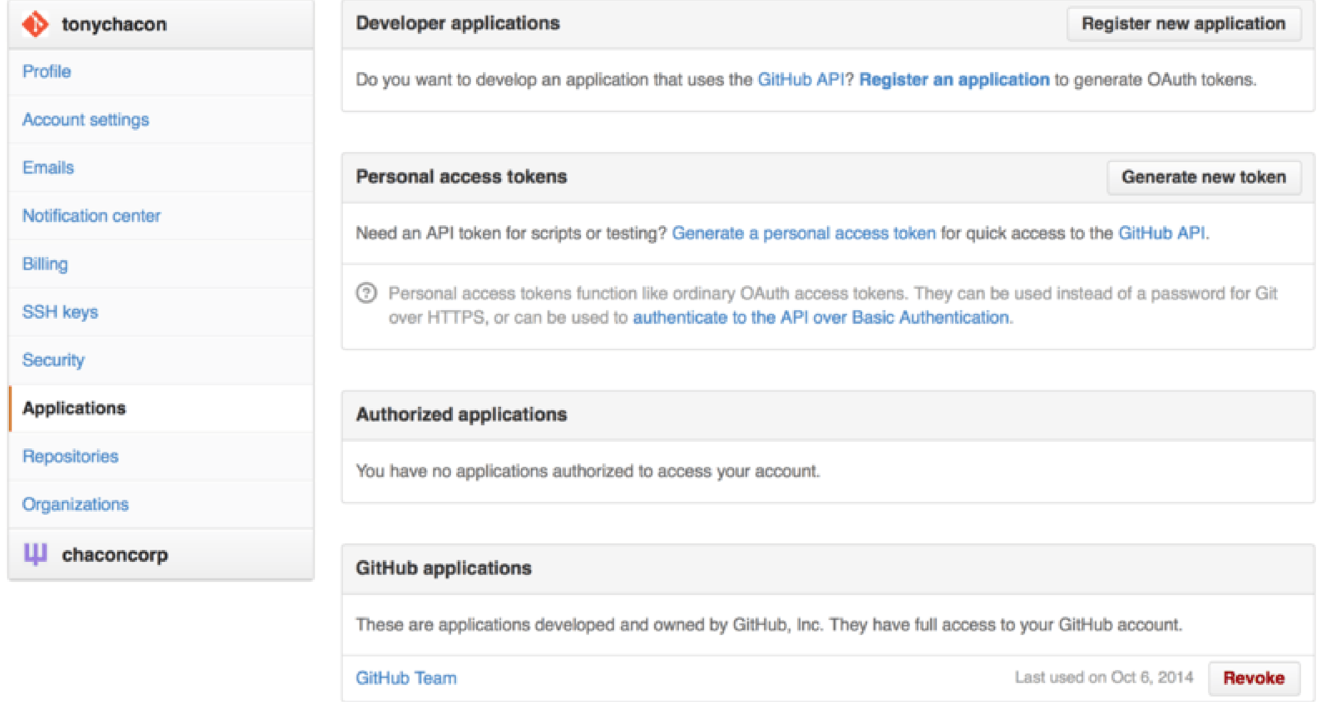


Figure 133. Генерирайте token за достъп в “Applications” секцията от страницата с настройки

Ще бъдете попитани за обхвата на token-а и за кратко описание. Уверете се, че използвате добро описание, така че да сте спокойни по-късно, когато изтривате token-а след като вече не се нуждаете от него.

GitHub ще ви покаже token-а само веднъж, така че се уверете, че сте го копирали. Можете вече да го използвате за автентикация вместо име и парола. Това е чудесно, защото можете да ограничите обхвата на това, което желаете да манипулирате и също така, token-ът може да се отмени.

Получавате и допълнително предимство. Без автентикация, можете да правите до 60 заявки на час. Ако се автентикирате, броят им нараства до 5,000 на час.

Нека го използваме за да пуснем коментар по един от нашите Issues. Ще коментираме Issue #6. За да направим това, изпращаме HTTP POST заявка към `repos/<user>/<repo>/issues/<num>/comments` с token-а, който току що генерирахме като Authorization хедър.

```
$ 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:"
}
```

Сега, ако отидем в този issue, можем да видим коментара ни там, както е показано в Коментар изпратен с GitHub API.

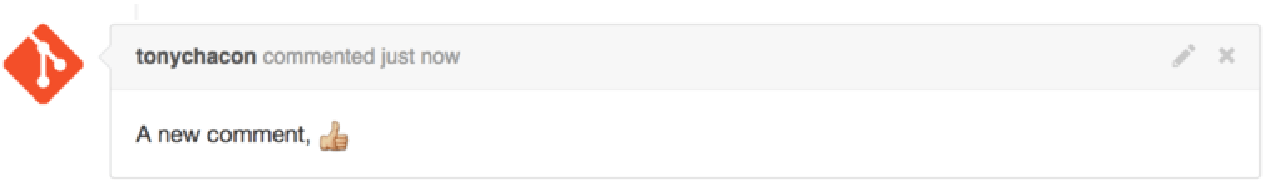


Figure 134. Коментар изпратен с GitHub API

Може да ползвате API интерфейса за да правите почти всичко, което правите и през сайта — създаване и настройка на milestones, асоцииране на хора към Issues и Pull Request-и, създаване и промяна на етикети, достъп до данните на къмит, създаване на нови къмити и

клонове, отваряне, затваряне и сливане на Pull Request-и, създаване и редакция на екипи, коментари по редове на код в Pull Request, търсене в сайта и т.н.

Промяна на статуса на Pull Request

Разглеждаме последния пример, понеже е наистина полезен, ако работите с Pull Request-и. Всеки къмит може да има един или повече статуси асоциирани с него и разполагате с API за добавяне и запитване към статус.

Повечето Continuous Integration и testing услуги използват този API за да реагират на публикувания теставайки кода, който е бил пратен и след това рапортувайки дали конкретния къмит е преминал всички тестове. Можете също да ползвате това за проверка дали къмит съобщенията са коректно форматирани, дали изпратилия къмита е следвал указанията ви за сътрудничество в проекта, дали къмитът е с валиден подпис — и безброй други неща.

Да приемем, че сте създали webhook във ваше хранилище, който се свързва с малка уеб услуга, която проверява за стринга `Signed-off-by` в къмит съобщението.

```ruby
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
```

Надяваме се, че кодът е лесно проследим. Преглеждаме всеки изпратен къмит, търсим за стринга *Signed-off-by* в къмит съобщението и накрая изпращаме HTTP POST заявка към `/repos/<user>/<repo>/statuses/<commit_sha>` API endpoint-а със съответния статус.

В този случай, можете да изпратите статус (*success*, *failure*, *error*), за описание на това какво се е случило, URL адрес за повече информация и “context” в случай, че има няколко статуса за единичен къмит. Например, една testing услуга може да предостави статус и една валидизираща услуга също може да предостави статус — “context” полето е това, което ги диференцира.

Ако някой отвори нов Pull Request в GitHub и този hook е бил настроен, може да видите нещо като Commit status през API.

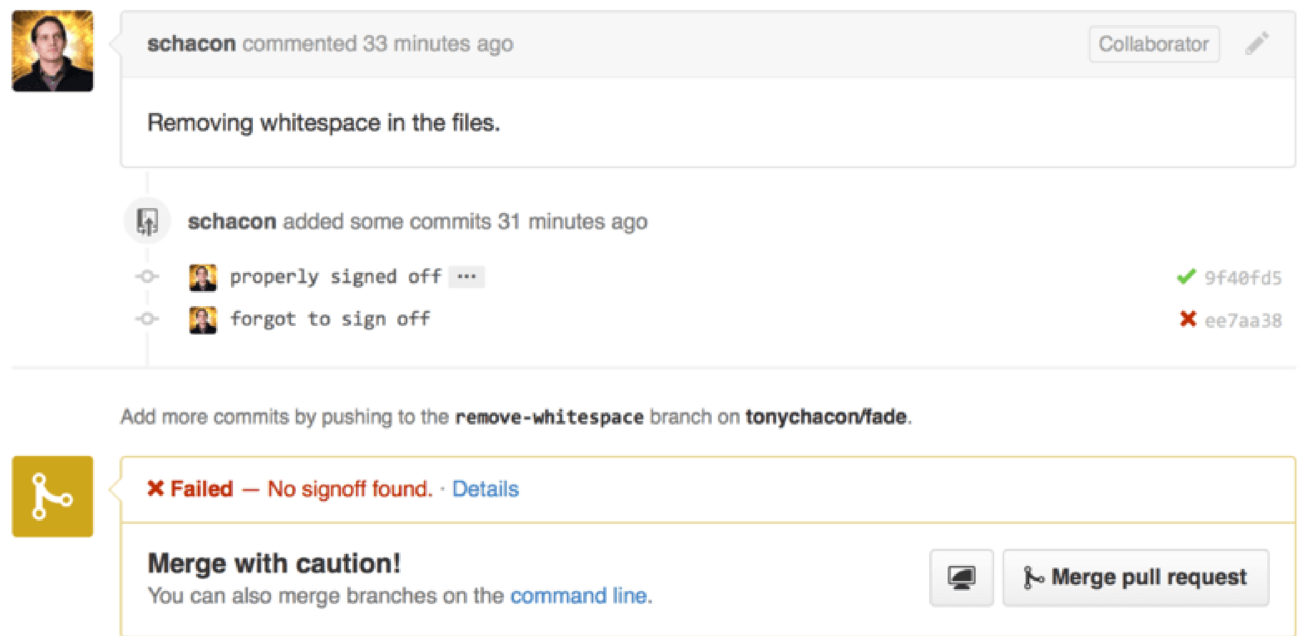


Figure 135. Commit status през API

Сега може да видите малка зелена отметка до къмита, който има стринга “Signed-off-by” в съобщението си и червено кръстче до този, който авторът е забравил да подпише. Може да видите също, че Pull Request-ът приема статуса на последния къмит в клона и ви предупреждава, ако резултатът е грешка. Това е наистина удобно, ако ползвате това API за тестове, така че да не слеете без да искате някой къмит, който не е минал през теста успешно.

Octokit

Въпреки, че правим всичко през `curl` и прости HTTP заявки в нашите примери, налични са няколко open-source библиотеки, които могат да ви помогнат да ползвате API интерфейса в по-идиоматичен стил. По време на писането на книгата, поддържаните езици включваха Go, Objective-C, Ruby, и .NET. Проверете https://github.com/octokit за повече информация, понеже те поемат повечето HTTP действия вместо вас.

Надяваме се, че тези инструменти ще ви помогнат да настроите поведението на GitHub така, че да комуникира с вашите системи и работни похвати така като желаете. За пълна документация на целия API интерфейс, както и за упътвания за често извършвани задачи, посетете https://developer.github.com.

Обобщение

Сега вече сте GitHub потребител. Знаете как се създава акаунт, как се управлява организация, как се създават и актуализират хранилища, как да допринасяте към проектите на друго потребители и да приемате код от тях във вашите проекти. В следващата глава ще разгледаме някои мощни инструменти и съвети за справяне със сложни ситуации, които ще ви направят майстори на Git.

Git инструменти

Дотук научихте за повечето ежедневни команди и работни похвати, от които имате нужда в управлението и поддръжката на Git хранилище. Овладяхте основните начини за проследяване и къмитване на файлове, за манипулация на индексната област и ползата от topic клоновете и сливането.

Сега ще разгледаме множество полезни трикове на Git, които може и да не ползвате често в ежедневната работа, но могат да се окажат полезни в един момент.

Избор на къмити

Git позволява да се обръщате към единичен къмит, серия или множество от къмити по няколко начина. Те не са непременно очевидни, но е полезно да се знаят.

Единични къмити

Очевидно можете да се обърнете към единичен къмит по неговия пълен, 40-символен SHA-1 хеш, но съществуват и по-лесни начини за това. Тази секция представя няколко такива.

Скъсен SHA-1

Git е достатъчно гъвкав да определи кой къмит имате предвид, ако укажете първите няколко символа от хеша, стига да подадете поне 4-символен, недвусмислен стринг. Това значи, че в базата данни с обектите няма друг такъв, който да започва със същия префикс.

Например, за да изследвате специфичен къмит, в който знаете че сте добавили дадена функционалност, бихте могли първо да изпълните `git log`, за да го намерите:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update 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

    Add some blame and merge stuff
```

В този случай, нека кажем, че се интересувате от къмита, чийто хеш започва с `1c002dd....`. Можете да инспектирате къмита с всеки от следните варианти на `git show`:

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git може да установи късите, уникални абревиатури на вашите SHA-1 стойности. Ако подадете опцията `--abbrev-commit` към `git log`, ще бъдат отпечатани съкратените версии; по подразбиране Git използва седем символа:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial commit
```

В общи линии, 8 до 10 символа са повече от достатъчни за гарантирана уникалност в рамките на един проект. Например, към февруари 2019, Linux ядрото (доста мащабен проект) има над 875 хиляди къмита и почти 7 милиона обекта, като не съществуват два такива с повтарящи се първи 12 символа в своя SHA-1 хеш.

КРАТКО УТОЧНЕНИЕ ЗА SHA-1

Много хора се притесняват, че в даден момент, по случайност биха могли да имат два отделни обекта в едно хранилище с една и съща SHA-1 стойност. Какво следва тогава?

Ако се случи да къмитнете обект, който да има същия SHA-1 хеш като предишен, *различен* такъв в хранилището, Git ще види предишния обект в базата данни, ще приеме че той вече е бил записан и просто ще го използва наново. Ако се опитате да извлечете този обект наново в даден момент, винаги получавате данните от първия обект.

Обаче, трябва да сте наясно колко супер малко вероятно е това. SHA-1 хеш сумата е с дължина 20 байта или 160 бита. Броят на случайно хешираните обекти, необходими за осигуряване на 50% вероятност от единично повторение е приблизително 2^{80} (формулата за определяне на вероятност за конфликт е `p = (n(n-1)/2) * (1/2^160))`. 2^{80} е 1.2 x 10^{24} или 1 милион милиарда милиарда. Това е 1200 пъти по-голямо число от броя песъчинки на земята.

Ето пример, който дава идея за това какво е нужно да получите SHA-1 повторение. Ако всички 6.5 милиарда човека на земята програмираха, и всяка секунда всеки един от тях произвежда код еквивалентен на цялото Linux ядро (6.5 милиона Git обекта) и го изпраща в едно грамадно общо Git хранилище, биха били необходими около 2 години докато това хранилище получи толкова обекти, че да има 50% шанс от единично SHA-1 повторение. Така че, SHA-1 колизията е по-малко вероятна от това всеки член на екипа ви да бъде атакуван и убит от вълци в различно място в една и съща нощ.

Ако заделите няколко хиляди долара за изчислителна мощ, възможно е да синтезирате два файла с един и същи хеш, както беше демонстрирано от https://shattered.io/ през февруари 2017г. Git постепенно преминава към SHA256 като хеширащ алгоритъм по подразбиране, който е много по-устойчив на collision атаки и има помощен код за смекчаване на този тип атаки (въпреки че, не може да ги елимиира изцяло).

Референции към клонове

Един прост начин да се обърнете към специфичен къмит, ако той е на върха в даден клон, е директно да използвате името на клона във всяка Git команда, която очаква референция към къмит. Например, ако искате да изследвате последния къмит в клон, следните две команди са еквивалентни, ако името на клона е topic1 и клонът сочи към къмита с хеш `ca82a6d...`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Ако искате да видите към кой специфичен SHA-1 сочи клона или да разберете как се представят тези примери по отношение на SHA-1 хешовете, можете да ползвате plumbing

инструмент на Git наречен `rev-parse`. Можете да видите Git на ниско ниво за повече информация за plumbing инструментите; в общи линии, `rev-parse` се използва за операции на по-ниско ниво и не е предназначен за ежедневни дейности. Обаче, понякога той може да е полезен, ако искате да знаете какво се случва зад кулисите. Ето как можете да стартирате `rev-parse` във вашия клон.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog съкратени имена

Едно от нещата, които Git прави на заден план докато вие работите, е да пази т. нар. "reflog" — дневник на това къде са били вашите HEAD и branch референции за последните няколко месеца.

Можете да го видите с командата `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.
1c002dd HEAD@{2}: commit: Add 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
```

Всеки път, когато върхът на клона се обнови по каква да е причина, Git съхранява тази информация за вас в тази временна история. Можете да я използвате за да се обръщате към по-стари къмити. Например, ако искате да видите петата предишна стойност на HEAD на вашето хранилище, можете да използвате референцията `@{5}`, която се вижда в историята:

```
$ git show HEAD@{5}
```

Можете да ползвате същия синтаксис за да видите къде е бил клона преди определено време. Ако искате примерно да разбере в какво състояние е бил `master` клона вчера, можете да изпълните:

```
$ git show master@{yesterday}
```

Това ще ви покаже къде е сочил върха на `master` клона вчера. Тази техника обаче работи само за данни, които все още са в reflog историята, така че не можете да я използвате за къмити по-стари от няколко месеца.

За да видите reflog информацията форматирана като `git log` изход, изпълнете `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update 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'
```

Важно е да се подчертае, че reflog информацията е строго локална — това е дневник на действията, които *само вие* сте извършили във *вашето* хранилище. Референциите няма да са същите в копието на хранилището на някой друг. Също така, веднага след като сте клонирали хранилище, ще имате празна reflog история, понеже се подразбира, че все още не са правени никакви промени по него. Ако изпълните `git show HEAD@{2.months.ago}`, ще видите съответния къмит само ако сте клонирали проекта поне два месеца преди това — ако сте го клонирали по-късно, ще видите само първия ви локален къмит.

Мислете за reflog историята като за Git версия на историята на вашия шел

Ако имате опит с UNIX или Linux, можете да гледате на reflog дневника ви като за Git версия на вашата история на шела, която представлява списък на това, което само вие сте правили във вашата "сесия" и няма нищо общо с това, което друг потребител на операционната система би могъл да е вършил на същата машина.

Escaping на скоби в PowerShell

В PowerShell, скобите като `{` и `}` са специални символи и трябва да бъдат escaped. Можете да го направите с backtick ` или да поставите къмит обръщението в кавички:

```
$ git show HEAD@{0}     # will NOT work
$ git show HEAD@`{0`}   # OK
$ git show "HEAD@{0}"   # OK
```

Йерархични референции

Другият основен начин да укажете къмит е през неговото родословие. Ако поставите символа `^` в края на референция към къмит, Git ще намери неговия родител. Да допуснем, че историята на проекта ви изглежда така:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
* 9b29157 Add open3_detach to gemspec file list
```

Сега, можете да видите предишния къмит указвайки `HEAD^`, което означава "родителят на 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'
```

Префикс на символа в Windows

В Windows и `cmd.exe`, `^` е специален символ и е необходимо да се третира по различен начин. Можете или да го напишете два пъти или да напишете къмит референцията в кавички:

```
$ git show HEAD^     # няма да работи в Windows
$ git show HEAD^^    # OK
$ git show "HEAD^"   # OK
```

Можете да въведете и число след `^` за да укажете *кой* родител искате – например, `d921970^2` означава "вторият родител на d921970." Този синтаксис е полезен само за merge къмити, които имат повече от един родител — *първият* родител на merge къмит е от клона, в който сте били по време на сливането (често това е `master`), докато *вторият* родител идва от клона, който е бил слят (примерно, `topic`):

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add 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
```

Символът `~` (тилда) също се използва за референция на унаследявания. Той също указва първия родител, така че `HEAD~` и `HEAD^` са еквивалентни. Разликата идва, когато укажете и число. `HEAD~2` означава "първият родител на първия родител," или "дядото" — това трасира първите родители толкова пъти, колкото е указано с числото. Например, в историята от преди малко, `HEAD~3` ще бъде:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

Това може да се напише и като `HEAD~~~`, резултатът ще е същия:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

Можете също да комбинирате тези символи — за да укажете втория родител на предишната референция (приемаме, че е бил merge къмит), използвайте `HEAD~3^2` и т.н.

Обхвати от къмити

Видяхме как се указват единични къмити, нека да видим как може да реферираме и обхвати от къмити. Това е полезно при управлението на клонове — ако имате множество клонове код, можете да използвате подобен вид обръщения, за да отговорите на въпроси от рода на "Каква работа от този клон все още не съм слял в главния ми клон?"

Две точки

Най-обикновеният начин за указване на обхват е с две точки. Това указва на Git да намери множество къмити, които са достъпни от единия, но недостъпни от другия клон. Нека приемем, че имате история подобна на Примерна история за избор на обхват.

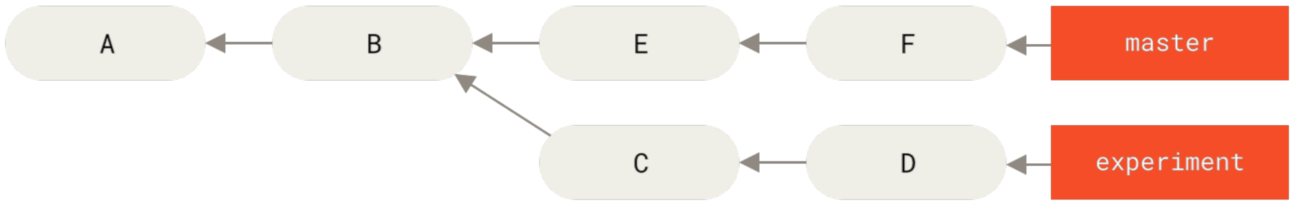


Figure 136. Примерна история за избор на обхват

Да кажем, че искате да видите коя част от работата в клона `experiment` все още не е интегрирана в `master` клона. Можете да укажете на Git да покаже списък на само тези къмити с `master..experiment` — това означава "всички къмити достъпни през `experiment`, които не са достъпни през `master`." За по-голяма яснота, в диаграмата къмитите са показани със символи:

```
$ git log master..experiment
D
C
```

Ако пък желаете да покажете обратното — всички къмити в `master`, които не са интегрирани в `experiment` — просто разменете имената на клоновете. `experiment..master` ви показва всичко от `master`, което е недостъпно през `experiment`:

```
$ git log experiment..master
F
E
```

Това е много полезно, ако държите да обновявате `experiment` с възможно най-новите данни и да видите какво предстои да сливате. Друго често използване на този синтаксис е когато искате да прегледате какво предстои да публикувате в отдалечено хранилище:

```
$ git log origin/master..HEAD
```

Тази команда ще отпечата всички къмити в текущия ви клон, които липсват в `master` клона на отдалеченото хранилище `origin`. Ако изпълните `git push` и текущият локален клон следи `origin/master`, то къмитите изведени от командата `git log origin/master..HEAD` ще са тези, които ще бъдат изпратени към сървъра. Можете също така да пропуснете името на клона от едната страна на двете точки и в такъв случай Git ще подразбира `HEAD`. Например, ще получите същия резултат като в предишния пример ако просто напишете `git log origin/master..` — Git замества празното пространство в края с `HEAD`.

Много точки

Синтаксисът с две точки е удобен за съкращение, но може да искате да укажете повече от два клона за вашето търсене, така че да намерите кои са къмитите във всеки от произволен брой клонове, които не присъстват в текущия клон. Git позволява това със символа `^` или фразата `--not` изписани преди всяка референция от която не искате да виждате достъпни къмити. Така следните три команди са еднакви:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Това е хубаво, защото с този тип синтаксис можете да укажете повече от две референции в заявката, което не става при използване на две точки. Например, ако искате да видите всички къмити достъпни от клоновете `refA` или `refB` но не и през `refC`, можете да използвате една от следните команди:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Това вече ви дава много мощна система за запитвания, която да ви помогне да видите какво съдържат клоновете ви.

Три точки

И последният основен начин за извличане на набор от къмити е т.нар. triple-dot синтаксис, който указва на Git да ви потърси всички къмити, които са достъпни от *кой да* е от двата клона, но не и от двата едновременно. Погледнете отново историята на къмитите от диаграмата Примерна история за избор на обхват. Ако искате да видите какво има в `master` или `experiment` но не и в двата клона, можете да изпълните:

```
$ git log master...experiment
F
E
D
C
```

Отново, това ви дава нормален `log` изход, но извежда информация само за тези 4 къмита в реда на датите им.

Често използван параметър към тази команда е `--left-right`, който в допълнение ще ви покаже от коя страна на обхвата е всеки от намерените къмити. Това прави примера ни една идея по-полезен:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

С тези инструменти можете по-лесно да накарате Git да ви покаже къмита или множеството от къмити, които искате да обследвате.

Интерактивно индексиране

В тази секция ще разгледаме няколко интерактивни Git команди, позволяващи да настроите къмитите си така, че да включват само определени файлове, комбинации от файлове или части от файлове. Това е полезно в случай, че сте направили промени по много файлове и след това искате промените да бъдат въведени като няколко организирани логически къмита, а не наведнъж в един голям и объркващ къмит. По този начин улеснявате вашите колеги, които евентуално биха искали да разгледат и разберат вашия принос.

Ако изпълните `git add` с параметър `-i` или `--interactive`, Git минава в интерактивен шел режим, показвайки нещо такова:

```
$ 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: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now>
```

Може да видите, че тази команда показва съвсем различен изглед на индексната ви област — в общи линии получавате резултатите от `git status`, но по един по-кратък и информативен начин. Тя извежда промените, които сте индексирали отляво, както и неиндексираните отдясно.

След това идва "Commands" секцията, която ви позволява множество действия като индексиране и деиндексиране на файлове, индексиране на част от файлове, добавяне на нови файлове и показване на diffs на индексираните данни.

Добавяне и изваждане от индекса

Ако въведете `u` или `2` (за обновяване) в промпта `What now>`, ще бъдете запитани кои файлове искате да вкарате в индексната област:

```
What now> u
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

За да индексирате `TODO` и `index.html`, може да въведете номерата им в списъка:

```
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>>
```

Символът `*` до всеки от файловете сега индикира, че съответния файл е избран за индексиране. Ако сега натиснете Enter в промпта `Update>>` без да въвеждате друго, Git взема всички избрани файлове и ги вкарва в индекса:

```
Update>>
updated 2 paths

*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Сега може да видите, че `TODO` и `index.html` файловете са индексирани, докато `simplegit.rb` все още не е. Сега може да извадите от индекса файла `TODO` избирайки опцията `r` или `3` (за revert):

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> r
           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
```

Поглеждайки отново във вашия Git status, може да видите, че `TODO` вече е извън индекса:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Ако искате diff на индексираното, използвайте командата `d` или `6` (за diff). Тя показва списък на индексираните файлове и може да изберете тези, за които искате да видите индексирания diff. Това е съвсем същото като да изпълните `git diff --cached` в нормалния команден ред:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> d
           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">
```

С тези основни команди може да улесните работата си с индексната област.

Индексиране на пачове

Също така може да вкарате в индекса само определени *части* от файлове, вместо всички промени. Например, ако сте направили две промени във файла `simplegit.rb`, но искате да индексирате само едната, можете да го направите лесно с Git. От същия интерактивен промпт, въведете `p` или `5` (за patch). Git ще ви попита кои файлове искате да индексирате частично. След това, за всяка секция от избраните файлове, ще ви бъдат показани големи парчета diff информация от всеки файл (hunks) и ще бъдете попитани дали искате да ги индексирате, едно по едно:

```
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,?]?
```

В този момент имате цял куп опции как да продължите. Въведете `?` за помощна

информация:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - индексира текущото парче код (hunk)
n - пропуска този hunk
a - индексира този и всички останали hunks от файла
d - отменя индексирането на този и всички останали hunks от файла
g - избор на hunk към който да се премине
/ - търсене на hunk по регулярен израз (regex)
j - оставя текущия hunk без решение, показва следващия нерешен hunk
J - оставя текущия hunk без решение, показва следващия hunk
k - оставя текущия hunk без решение, показва предишния нерешен hunk
K - оставя текущия hunk без решение, показва предишния hunk
s - разделя текущия hunk на по-малки части
e - ръчна редакция на текущия hunk
? - показва помощната информация
```

Обикновено, ще избирате опциите `y` или `n`, ако искате да индексирате всеки hunk, но индексирането им наведнъж в определени файлове или отлагането на решението за даден hunk за по-късно, също може да е полезно. Ако индексирате една част от файл и оставите друга извън индекса, статуса може да изглежда така:

```
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
```

Интересната част е в реда на `simplegit.rb`. Той показва, че част от редовете код във файла са индексирани, а други не. Така имате частично индексиран файл. Сега можете да излезете от интерактивния промпт и да изпълните `git commit` за да къмитнете частично индексираните файлове.

Използването на интерактивния промпт не е единствения начин да постигнете това — може да стартирате същата процедура използвайки `git add -p` или `git add --patch` от командния ред.

Освен това, можете да използвате patch режима за частично възстановяване на файлове с командата `git reset --patch`, за изваждане в работната област на част от файлове с `git checkout --patch`, а също и за stashing на части от файлове с `git stash save --patch`. Ще видим повече подробности за тези команди по-натам в книгата.

Stashing и Cleaning

Често докато работите по дадена част от вашия проект, нещата стават заплетени и в недовършен статус и може да поискате да спрете работата по текущия проблем за известно време за да свършите нещо друго, в друг клон на хранилището. Проблемът е в това, че не

желаете да къмитвате половинчати работи само за да можете да се върнете към моментното им състояние по-късно. Отговорът на този проблем идва с командата `git stash`.

Stashing-ът е процес, при който Git взема вашия моментен статус на извършена работа (това са модифицираните проследявани файлове в едно с индексираните такива) — и го съхранява в нещо като стек от недовършени промени, които след това може да се приложат обратно по всяко време, дори и върху различен клон. Правейки това, работната директория се маркира като чиста и можете да превключите към друг клон. На stashing-а може да гледате като временно маскиране/скриване на промените.

Миграция към `git stash push`

Към октомври 2017 в мейлинг листата на Git имаше оживена дискусия, в резултат на което `git stash save` сега се счита за отхвърлена (deprecated) и се препоръчва алтернативната `git stash push`. Основната причина за това е, че `git stash push` предоставя опцията за stashing на избрани *pathspecs*, нещо което `git stash save` не поддържа.

Обаче, имайте предвид, че `git stash save` няма да изчезне скоро като команда. Добре е да опитате да преминете към `push` алтернативата просто за да сте в тон с новата функционалност.

Stashing на промените

За да демонстрираме процеса по маскирането, ще поработим в нашия проект по няколко файла и можем да индексираме някои от промените. Ако изпълним `git status`, можем да видим нашия недовършен (dirty) статус:

```
$ 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
```

Сега имаме модифицирани файлове както и индексирани промени и искаме да превключим клоновете, но без да къмитваме. Ще се наложи да ги маскираме. За да вкараме нов stash в стека ни, изпълняваме `git stash` или `git stash push`:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

Вече можем да видим, че работната ни директория е чиста, в статус clean:

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

В този момент, вече можем да превключим към друг клон, а промените ни ще се пазят в стека. За да видим какви stash-ове има там, изпълняваме `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

В този случай можем да видим, че имаме два stash-а съхранени преди последния, така че имаме достъп до три предишни състояния на проекта. Можете да възстановите обратно последната маскирана работа използвайки командата от помощния изход отпечатан в момента, в който направихме маскирането: `git stash apply`. Ако искате да възстановите по-стара версия, може да я укажете с името ѝ, например така: `git stash apply stash@{2}`. Ако не укажете име, Git подразбира най-новия stash и опитва да приложи него:

```
$ git stash apply
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:   index.html
    modified:   lib/simplegit.rb

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

Ще видите, че Git модифицира файловете и ги връща до състоянието им, в което са били по време на съхранението на stash-а. В този случай имахме чиста работна директория в момента, в който се опитахме да възстановим stash-а и се опитахме да го приложим върху същия клон, от който го записахме. Обаче, чистата работна директория и същия клон не са задължително условие за успешно прилагане на stash. В действителност можете да запишете stash в един клон, да превключите към друг по-късно и да опитате да възстановите stash-а в него. Също така можете да имате модифицирани и некъмитнати

файлове в работната директория при опита да приложите stash — Git ще ви даде merge конфликти, ако нещо не се прилага чисто.

Може би сте забелязали, че файловете ви са възстановени, но файлът който беше индексиран преди вече не е индексиран. Ако искате да е, ще трябва да изпълните `git stash apply` с параметъра `--index` и Git ще се опита да възстанови и промените по индекса. Така че, ако бяхте изпълнили тази последна команда, ще се върнете в изцяло оригиналната позиция:

```
$ git stash apply --index
On branch master
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
```

Опцията apply само се опитва да приложи съхранената по-рано работа — но вие продължавате да я имате в стека. За да я махнете от там, може да изпълните `git stash drop` с името на stash-а, който искате да изтриете:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Друг вариант е да изпълните `git stash pop`, което прилага stash-а и веднага след това го изтрива от стека.

Креативен Stashing

Има няколко stash варианта, които могат да бъдат полезни. Първият от тях, който е доста популярен, е опцията `--keep-index` към командата `git stash`. Това указва на Git не само да включи в stash-а индексираното съдържание, но едновременно с това да го остави в индекса.

```
$ 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 file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Друга често използвана опция е да се stash-нат едновременно проследяваните и непроследяваните от Git файлове. По подразбиране, `git stash` ще съхрани в stash стека само променените и индексирани *проследявани* файлове. Ако укажете `--include-untracked` или `-u`, Git ще включи в новия stash и файловете, които не следи Обаче, включването на непроследяващи се файлове в stash-а все пак няма да добави изрично *игнорираните* файлове. За да присъедините и тях, използвайте флага `--all` (или само `-a`).

```
$ 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 file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Накрая, ако подадете параметъра `--patch` към командата, Git няма да включи всичко променено, а ще ви пита интерактивно кои от промените искате да включите в stash-а и кои да оставите в работната директория.

```
$ 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
```

Създаване на клон от Stash

Ако stash-нете някаква ваша работа, изоставите я за повече време и продължите да работите по клона, от който сте го направили — може да се окаже, че обратното ѝ прилагане причинява проблеми. Ако процесът по възстановяването се опита да промени файл, който е бил редактиран след stash-а, ще получите merge конфликт и ще трябва да го разрешите. Обаче, ако търсите по-лесен начин да тествате маскираните промени отново, можете да изпълните `git stash branch <new branchname>`, което ще създаде нов клон с избраното име, ще извлече в работната директория къмита в който сте били, когато сте направили stash-а, ще приложи обратно работата ви там и ще изтрие stash-а, ако се приложи успешно:

```
$ git stash branch testchanges
M	index.html
M	lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
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

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Това е лесен и удобен начин за възстановяване на stash-ната работа.

Почистване на работната директория

Накрая, може да не искате да маскирате определени файлове от работната директория а просто да се отървете от тях. Командата `git clean` ще ви помогне за това.

Чести случаи за подобна необходимост са генерирани временни файлови от външни инструменти или компилация, които искате да махнете, за да остане чист вариант на вашата работа.

Трябва да внимавате с тази команда, защото тя е проектирана да изтрива непроследявани файлове от работната директория. Ако решите по-късно, че те ви трябват, често няма връщане назад. По-безопасен вариант е `git stash --all` за да изтриете всичко, но да си го запазите за всеки случай в stash.

Ако решите да чистите въпреки тези съображения, използвайте `git clean`. Изпълнението на `git clean -f -d` премахва всички непроследявани файлове от работната директория, както и всички поддиректории, които могат да се окажат празни в резултат. Флагът `-f` означава *force* или "really do this" и се изисква, ако конфигурационната променлива на Git `clean.requireForce` не е изрично указана като false.

Ако все пак искате да видите предварително какъв ще е резултата от изпълнението на командата, добавете флага `-n` (или `--dry-run`) за да направите "dry run" и да видите какво ще се изтрие преди да е станало късно.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

По подразбиране, `git clean` ще изтрие само файловете, които не се следят и не се игнорират. Всеки файл, който съвпада с маска от `.gitignore` или други ignore файлове няма да бъде изтрит. Ако искате да премахнете и тези файлове (като например `.o` файлове генерирани от компилатора) така, че да имате чиста версия на проекта, добавете флага `-x` към командата.

```
$ 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/
```

Ако не сте сигурни в крайния резултат, добре е винаги да използвате `-n` флага преди да почистите окончателно. Командата също така може да се изпълнява интерактивно с флага `-i` или "interactive", което също може да е полезно като предпазно средство.

Така ще може лично да потвърждавате изтриването на всеки обект.

```
$ 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             5: quit
    6: help
What now>
```

По този начин минавате през всеки файл постъпково или може да укажете шаблон за търсене интерактивно.

Съществуват специфични ситуации, в които трябва да сте по-настойчиви за да накарате Git да ви почисти работната директория. Ако сте в работна директория, под която сте копирали или клонирали други Git хранилища (вероятно като под-модули), дори `git clean -fd` ще откаже да ги изтрие. В случаи като този, трябва да добавите втори `-f` флаг за да потвърдите.

Подписване на вашата работа

Git е криптографски сигурен, но не и дуракоустойчив. Ако вземате код от други хора в Интернет и искате да проверите дали къмитите са действително от надежден източник, Git

разполага с начини за подписване и проверка през GPG.

За GPG

Преди всичко, ако искате да подписвате каквото и да е, нуждаете се от конфигуриран GPG и персонален 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
```

Ако нямате такъв инсталиран ключ, може да си генерирате с командата `gpg --gen-key`.

```
$ gpg --gen-key
```

След като веднъж имате частен ключ, можете да настроите Git да го използва за подпис на нещата ви посредством конфигурацията `user.signingkey`.

```
$ git config --global user.signingkey 0A46826A
```

Сега Git по подразбиране ще използва този ключ за да подписва тагове и къмити, ако желаете това.

Подписване на тагове

Вече имате GPG частен ключ, да видим как можете да подписвате тагове. Всичко, от което се нуждаете е да използвате флага `-s` вместо `-a`:

```
$ 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
```

Ако пуснете `git show` на този таг, може да видите прикрепена вашата GPG сигнатура:

```
$ 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

    Change version number
```

Проверка на тагове

За да проверите подписан таг, изпълнете `git tag -v <tag-name>`. Тази команда използва GPG за проверка на сигнатурата. За да работи това коректно, имате нужда от публичния ключ на подписващия във вашия keyring:

```
$ 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
```

Ако не разполагате с публичния ключ, ще получите вместо горния резултат нещо такова:

```
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'
```

Подписване на къмити

В по-новите версии на Git (v1.7.9 и нагоре), можете да подписвате също и индивидуални къмити. Ако искате да подписвате директно къмитите вместо само таговете, трябва да добавите флага `-S` към командата `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
```

За да видите и проверите тези сигнатури, на разположение е аргумента `--show-signature` за командата `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    Signed commit
```

В допълнение, можете да конфигурирате `git log` да проверява всички открити сигнатури и да ги показва в изхода си с формата `%G?`.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  Signed commit
ca82a6d N Scott Chacon  Change the version number
085bb3b N Scott Chacon  Remove unnecessary test code
a11bef0 N Scott Chacon  Initial commit
```

Тук можем да видим, че само последния къмит е подписан и успешно валидиран, докато предишните три не са.

В Git 1.8.3 и по-новите версии, `git merge` и `git pull` могат да се инструктират да проверяват и отхвърлят сливането на къмити, които не носят в себе си trusted GPG сигнатура с опцията `--verify-signatures`.

Ако използвате тази опция по време на сливането на клон и той съдържа къмити, които не са подписани и валидни, сливането ще бъде отказано.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Ако сливането съдържа само валидно подписани къмити, merge командата ще ви покаже всички проверени сигнатури и ще продължи нататък с процеса на сливане.

```
$ 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(+)
```

Можете също да ползвате `-S` флага с `git merge` за да подпишете сами получения merge къмит. Следващият пример проверява, че всеки къмит в клона, който ще бъде слят е подписан и едновременно с това подписва получения merge къмит.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>

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(+)
```

Всеки трябва да подписва

Подписването на тагове и къмити е хубаво нещо, но ако решите да ползвате този принцип в нормалния си работен процес, трябва да се уверите, че всеки в екипа разбира как да го прави. Ако това не е така, ще се озовете в ситуация, в която изразходвате доста усилия и време разяснявайки на колегите си как да преработят техните къмити в подписани версии. Затова е добре да се уверите, че всички в екипа познават добре GPG и ползите от подписването на работата преди да въведете подхода в нормалния работен процес.

Търсене

Независимо от размера на даден проект, често ще ви се налага да потърсите къде дадена функция се вика или къде е дефинирана, или пък да покажете историята на метод. Git осигурява полезни инструменти за търсене в кода и къмитите в базата данни бързо и лесно. Ще разгледаме някои от тях.

Git Grep

Git идва с командата `grep`, позволяваща ви лесно да търсите по стринг или регулярен израз във всяко къмитнато дърво, в работната директория и дори в индексната област. За примерите, които следват ще търсим в сорс кода на самия Git.

По подразбиране, `git grep` ще търси само във файловете от работната директория. Като първи вариант, можете да използвате флага `-n` или `--line-number` за да отпечатете номерата на редовете, в които Git намира съвпадения:

```
$ 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:826:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:482:             if (gmtime_r(&now, &now_tm))
date.c:545:             if (gmtime_r(&time, tm)) {
date.c:758:             /* gmtime_r() in match_digit() may have clobbered it */
git-compat-util.h:1138:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:1140:#define gmtime_r git_gmtime_r
```

В допълнение към тази основна функционалност, `git grep` поддържа множество интересни опции.

Например, вместо да отпечатвате всички попадения, може да инструктирате командата да съкрати изхода и да покаже само кои файлове съдържат търсения стринг и колко пъти чрез флага `-c` или `--count`:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:3
git-compat-util.h:2
```

Ако се интересувате от *контекста* в който се използва търсения стринг, можете да покажете цялата функция/метод с опцията `-p` или `--show-function`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(timestamp_t num, char c, const char *date,
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)) {
date.c=int parse_date_basic(const char *date, timestamp_t *timestamp, int *offset)
date.c:         /* gmtime_r() in match_digit() may have clobbered it */
```

Както се вижда, `gmtime_r` се извиква от функциите `match_multi_number` и `match_digit` във файла `date.c` (третият намерен резултат съдържа търсения стринг в коментар).

Можете също да търсите за сложни комбинации от стрингове с флага `--and`, което указва че на един ред трябва да има повече съвпадения. Например, нека потърсим всички редове код, които дефинират константа, чието име съдържа *кой да е* от подстринговете "LINK" или "BUF_MAX" специфично в по-стара версия на Git сорса маркирана с тага `v1.8.0` (ще подадем флаговете `--break` и `--heading`, които помагат да разделим изхода в по-четим формат):

```
$ 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 */
```

Командата `git grep` има няколко предимства пред нормалните команди за търсене като `grep` и `ack`. Първо, тя е наистина много бърза и второ - позволява търсене във всяко дърво в Git, не само в работната директория. Както видяхме в предния пример, ние търсехме в контекста на по-стара версия на сорс кода на Git, а не в текущата версия извлечена в работната ни област.

Търсене в Git Log

Може би търсите не *къде* съществува дадено нещо, а *кога* е съществувало или въведено. Командата `git log` има много мощни инструменти за намиране на специфични къмити по съдържанието на техните съобщения и дори по съдържанието на diff информацията, която въвеждат.

Ако например искаме да разберем кога константата `ZLIB_BUF_MAX` е била първоначално въведена в кода, можем да използваме флага `-S` (разговорно позната като Git "pickaxe" опция) за да укажем на Git да ни изведе само тези къмити, които са променили броя на срещанията на този стринг.

```
$ git log -S ZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Ако погледнем diff-а на тези къмити, можем да видим, че константата е била въведена в `ef49a7a` и променена в `e01503b`.

Ако се нуждаете от по-голяма прецизност, можете да подадете регулярен израз с `-G` опцията.

Line Log търсене

Друга особено полезна възможност е опцията за търсене на историята на ред от код. Просто пуснете `git log` с флага `-L` и ще получите историята на функция или ред от код в сорса.

Например, ако искаме да видим всяка промяна направена във функцията `git_deflate_bound` от файла `zlib.c`, можем да изпълним командата `git log -L :git_deflate_bound:zlib.c`. Тя ще опита да установи кои са границите на дефиницията на функцията и след това, гледайки през историята на промените, ще ни покаже всяка една редакция в кода като серия от пачове — чак до момента на дефинирането ѝ за първи път.

```
$ 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);
+}
+
```

Ако все пак Git не съумее да установи дефиницията на функция или метод във вашия програмен език, можете да подпомогнете търсенето с регулярен израз (*regex*). Например, същото нещо от примера по-горе можем да получим с командата `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. Можете също да подадете набор от редове или единичен ред и ще получите същия тип резултат.

Манипулация на историята

Често в работата си с Git може да поискате да ревизирате историята на локалните къмити. Една от чудесните страни на Git е, че ви позволява да вземате решения в последния възможен момент. Вие решавате кои файлове в кои къмити да отидат непосредствено преди къмитване с индексната област, вие може да решите, че в момента отлагате работата по даде проблем с `git stash` и също така, можете да презапишете предишни къмити така че

да изглеждат сякаш са се случили по различен начин. Това може да включва смяна на реда на къмитите, смяна на съобщенията или модифициране на файлове в къмит, комбиниране на няколко в един или разделяне на къмит на части или пък изтриване на къмит — всичко това преди да споделите работата си с колегите.

В тази част от книгата ще видим как се осъществяват такива задачи, така че да накарате историята на работата ви да изглежда по начина, по който вие искате преди да я споделите с другите.

Не публикувайте работата си преди да сте я довършили докрай

Едно от кардиналните правила на Git е това, че понеже вършите повечето работа локално във вашето копие на хранилището, вие разполагате с почти неограничена свобода да преправяте *локалната* си история. Обаче, след като веднъж сте публикували работата си, нещата значително се променят и е редно да гледате на нея като на финална версия, освен ако нямате наистина добра причина да промените това. Накратко, би следвало да избягвате да публикувате работата си до момента в който решите, че тя е достатъчно добра за да бъде споделена с другите.

Промяна на последния къмит

Промяната на най-последния къмит вероятно е най-често срещания вид манипулация на историята. Две неща ще искате да правите най-често: просто да редактирате къмит съобщението или да промените съдържанието на къмита добавяйки, изтривайки или променяйки файлове.

Ако става дума само за съобщението на последния къмит, това е лесно:

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

Командата зарежда съобщението на къмита в текстовия редактор, където можете да го редактирате, да запишете промените и да излезете. Когато затворите редактора, той записва нов къмит с редактираното съобщение и го прави последен къмит.

Ако, от друга страна, искате да промените нещо по действителното *съдържание* на къмита, процесът в общи линии работи по същия начин — първо направете желаните промени, индексирайте ги и след това `git commit --amend` командата ще *замени* последния записан къмит с вашите нови, коригирани данни.

Трябва да внимавате с тази техника, понеже тя променя SHA-1 хеша на къмита. Това е като много малко пребазиране — не променяйте последния къмит, ако вече сте го публикували!

Редактиран къмит може да се нуждае, но може и да не се нуждае от amended къмит съобщение

Когато редактирате къмит, имате възможност да смените както съобщението, така и съдържанието на данните в къмита. Ако променяте данните по същество, почти винаги е добре да обновите и съобщението, така че да отразява корекциите.

От друга страна, ако промените са тривиални (например поправка на правописна грешка или добавяне на файл, който сте забравили да индексирате) и оригиналното съобщение си е съвсем на място, можете просто да направите промените и да прескочите стъпката с редактора изцяло:

```
$ git commit --amend --no-edit
```

Смяна на повече къмит съобщения

За да промените къмит, който е назад в историята, ще ви трябват по-сложни инструменти. Git не разполага с инструмент за модифициране на историята, но можете да използвате rebase за да пребазирате серия от къмити върху HEAD-а, на който са били първоначално базирани, вместо да ги премествате към друг. С интерактивния инструмент за пребазиране след това можете да спирате на всеки къмит, който искате и да редактирате съобщението му, да добавяте файлове и т.н. Можете да стартирате rebase в интерактивен режим с флага `-i` на командата `git rebase`. Трябва да посочите колко назад искате да презаписвате къмити указвайки на командата върху кой къмит да пребазира.

Например, ако искате да промените последните три къмит съобщения или кое да е съобщение в тази група, вие подавате като аргумент на `git rebase -i` родителя на последния къмит, който искате да редактирате, което е `HEAD~2^` или `HEAD~3`. Може да е по-лесно да използвате `~3`, понеже опитвате да редактирате последните три къмита, но имайте предвид, че всъщност посочвате четири къмита назад, родителя на последния къмит, който искате да промените:

```
$ git rebase -i HEAD~3
```

Подчертаваме отново, че това е пребазираща команда — всеки къмит в обхвата `HEAD~3..HEAD` с променено съобщение *и всички негови наследници* ще бъдат презаписани. Не включвайте никакъв къмит, който вече сте изпратили на централния сървър — правейки това ще смутите другите разработчици, защото осигурявате алтернативна версия на една и съща промяна.

Изпълнявайки командата, получавате списък къмити в текстовия си редакто, подобно на това:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Важно е да кажем, че къмитите се изброяват в обратен ред на този, в който се виждат с `log` командата. Ако изпълните `log`, виждате нещо такова:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit
```

Забележете обърнатия ред. Интерактивното пребазиране ви дава скрипта, който ще изпълни. То ще започне от къмита, който указвате на командния ред (`HEAD~3`) и ще извърши промените въведени във всеки от тези къмити отгоре надолу. Най-старият се показва най-отгоре, защото той е първият, който ще бъде приложен.

Трябва да редактирате скрипта така, че да спре на къмита, който желаете да редактирате. За да направите това, променете думата 'pick' на 'edit' за всеки от къмитите, след които искате скрипта да спре. Например, за да промените само третото къмит съобщение, променяте файла да изглежда така:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Когато запишете и излезете от редактора, Git ви превърта обратно до последния къмит в този списък и ви връща в командния ред със следното съобщение:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

Инструкциите посочват точно какво да направите. Напишете:

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

Променете съобщението на къмита и затворете редактора. След това изпълнете:

```
$ git rebase --continue
```

Тази команда ще приложи останалите два къмита автоматично и след това сте готови. Ако смените думата pick с edit на повече редове, можете да повторите тези стъпки за всеки съответен къмит. Всеки път Git ще спира, ще ви позволи да промените къмита и ще продължи, когато сте готови.

Пренареждане на къмити

Можете да използвате интерактивното пребазиране за да размествате или изцяло да премахвате къмити. Ако искате да премахнете къмита “added cat-file” и да смените реда, в който останалите два се прилагат, може да смените rebase скрипта от това:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

към това:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

Когато запишете и затворите редактора, Git превърта клона назад до родителя на тези къмити, прилага `310154e` следван от `f7f3f6d` и спира. Сега редът на двата останали къмита е сменен, а "added cat-file" е изцяло премахнат.

Обединяване на къмити

Възможно е също така да вземете няколко къмита и да ги обедините в един единичен. Процесът е известен като Squashing и също може да се направи с инструмента за интерактивно пребазиране. Скриптът ви дава напътствия в rebase съобщението:

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

В този случай, ако вместо "pick" или "edit", укажете "squash", Git прилага текущата промяната и промяната непосредствено преди нея и ви позволява да слеете къмит съобщенията заедно. Така че, ако искате да направите единичен къмит от горните три, може да редактирате скрипта си така:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

Когато затворите редактора, Git прилага всичките три промени и пуска редактора още веднъж, за сливане на трите съобщения:

```
# This is a combination of 3 commits.
# The first commit's message is:
Change my name a bit

# This is the 2nd commit message:

Update README formatting and add blame

# This is the 3rd commit message:

Add cat-file
```

Записвайки това, вие получавате единичен къмит, който съдържа промените от всичките три предишни.

Разделяне на къмит

Разделянето ще отмени даден къмит и след това частично ще индексира и къмитне толкова пъти, колкото укажете. Например, решавате да разделите втория от трите къмита по-горе. Вместо "Update README formatting and add blame", искате да го разделите в два къмита със съобщения "Updated README formatting" за първия и "Add blame" за втория. Можете да го постигнете с `rebase -i` скрипта променяйки инструкцията на втория къмит на "edit":

```
pick f7f3f6d Change my name a bit
edit 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

След това, когато скриптът ви върне в командния ред, вие reset-вате къмита, вземате промените, които са били отменени и създавате няколко къмита от тях. Когато запишете и затворите редактора, Git превърта назад до родителя на първия къмит в списъка, прилага първия къмит (`f7f3f6d`), прилага втория (`310154e`) и ви връща в конзолата. Там можете да направите mixed reset на този къмит с командата `git reset HEAD^`, което на практика отменя къмита и оставя модифицираните файлове извън индекса. Сега можете да индексирате и къмитвате файлове докато получите колкото желаете къмита и след това да изпълните `git rebase --continue` за да завършите процеса:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'Update README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'Add blame'
$ git rebase --continue
```

Git ще приложи и последния къмит от скрипта (`a5f4a0d`) и сега историята ви изглежда така:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

Този процес променя SHA-1 стойностите за последните 3 най-нови къмита в списъка, така че се убедете, че никой от тях не е бил публикуван в споделено хранилище. Отбележете също така, че последният къмит в списъка (`f7f3f6d`) е непроменен. Въпреки, че е показан в скрипта, Git го оставя непроменен, понеже той е маркиран като "pick" и е приложен преди всички rebase промени.

Изтриване на къмит

Ако искате да изтриете къмит, можете да го направите със скрипта `rebase -i`. В списъка с къмити, поставете думата 'drop'' пред този, който искате да изтриете (или просто изтрийте реда от rebase скрипта):

```
pick 461cb2a This commit is OK
drop 5aecc10 This commit is broken
```

Поради начина, по който Git построява къмит обектите, изтриването или промяната на къмит предизвиква презапис на всички следващи след него. Колкото по-назад в историята се връщате, толкова повече къмити ще трябва да се създадат наново. Това може да предизвика купища merge конфликти, ако по-късно в историята има много къмити, зависещи от изтрития.

Ако в един момент в rebase процеса установите, че той не е бил добра идея, можете винаги да спрете. Изпълнете `git rebase --abort` и хранилището ще се върне в статуса, в което е било преди да стартирате пребазирането.

Ако в края на пребазирането решите, че резултатът не е какъвто очаквате, можете да използвате `git reflog` за да възстановите по-ранна версия на клона. Вижте Възстановяване на данни за повече информация за командата `reflog`.

Drew DeVault е създал практическо hands-on упътване с упражнения за използването на `git rebase`. Достъпно е на адрес: https://git-rebase.io/

Мощната алтернатива: filter-branch

Съществува и друга опция за презапис на историята, която се използва за модифициране на голям брой къмити в в скриптов маниер — например ако искате да си смените имейл адреса глобално или да премахнете файл от всеки къмит. Командата е `filter-branch` и може да променя огромни порции от вашата история, така че вероятно не би следвало да я ползвате — освен ако проектът ви все още не е публично достъпен или пък никой ваш

колега не е базирал работата си на някой от вашите къмити, които ще бъдат пренаписани. Обаче, командата може да бъде много полезна. Ще покажем няколко от най-честите ѝ приложения, така че да получите идея какво може да прави.

`git filter-branch` има много недостатъци и вече не се препоръчва като начин за презапис на историята. Вместо това, използвайте `git-filter-repo`, която е скрипт на Python и върши по-добра работа в ситуациите, в които нормално бихте използвали `filter-branch`. Документацията и сорс кода ѝ могат да се намерят на https://github.com/newren/git-filter-repo.

Изтриване на файл от всеки къмит

Това се налага доста често. Някой по невнимание къмитва голям двоичен файл с `git add .` и се налага да го махнете навсякъде. Или пък, без да искате сте къмитнали файл, съдържащ парола, а проектът трябва да стане с отворен код. В такива случаи `filter-branch` е инструментът, който вероятно ще искате да използвате, за да ремонтирате историята из основи. За да премахнете файла `passwords.txt` от цялата история използвайте параметъра `--tree-filter` на `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Опцията `--tree-filter` изпълнява указаната команда след всяко извличане на съдържанието на проекта и след това къмитва резултатите обратно. В този случай, вие изтривате файла `passwords.txt` от всеки един snapshot без значение дали присъства или не. Ако искате да премахнете всички случайно къмитнати backup файлове от вашия редактор, може да изпълните нещо като `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Ще видите как Git пренаписва дърветата и къмитите и след това премества указателя на клона в края. Добра идея е да направите това пробно в тестов клон и ако резултатите ви устройват, да го приложите и в master клона като му направите hard-reset. Можете да пуснете `filter-branch` и върху всички клонове с опцията `--all` към командата.

Превръщане на под-директория в Root директория

Да допуснем, че сте импортирали проект от друга source control система и имате под-директории, които са излишни (`trunk`, `tags` и т.н.). Ако искате да направите директорията `trunk` корен за проекта ви за всеки къмит, също може да използвате `filter-branch`:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Сега `trunk` е новата основна директория. Git автоматично ще премахне къмитите, които не касаят тази директория.

Смяна на имейл адрес глобално

Друга възможна ситуация е да сте забравили да пуснете `git config` за да настроите вашето име и имейл адрес преди да започнете същинската работа. Или пък, решавате даден проект от вашата компания да стане с отворен код и искате да смените служебните имейл адреси с персоналния ви такъв. Инструментът `filter-branch` може да помогне в автоматичната смяна на информацията в множество къмити. Трябва да внимавате и да смените само вашите адреси, така че използвайте `--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
```

Тази сложничка за изписване команда ще препише всеки къмит с новия адрес, който посочите. Понеже къмитите съдържат SHA-1 данни на своите родители, това също означава, че командата ще смени хешовете на всички къмити в историята, включително и на тези, които не съдържат вашия имейл.

Мистерията на командата Reset

Преди да преминем към по-специализираните инструменти на Git, нека поговорим още малко за `reset` и `checkout` командите. Тези команди са два от най-смущаващите аспекти в Git, когато за първи път се сблъскате с тях. Правят толкова много неща, че изглежда безнадеждно да бъдат разбрани и използвани ефективно. Ето защо, препоръчваме една проста метафора.

Трите дървета

Един по-лесен подход да мислите за `reset` и `checkout` е да гледате на Git като мениджър на съдържание за три различни дървета. Като казваме "дърво", в действителност разбираме "колекция от файлове", а не структурата от данни. Има няколко ситуации, където индексът на практика не работи като дърво, но за нашите цели е по-лесно да го възприемаме като такова.

Git като система управлява три дървета в нормалната си работа:

Дърво	Роля
HEAD	Snapshot на последния къмит, родител на следващия
Index	Snapshot за следващия къмит
Работна директория	Работна област

Дървото HEAD

HEAD е указателят към референцията на текущия клон, която от своя страна сочи към последния къмит направен в този клон. Това означава, че HEAD ще бъде родител на следващия създаден къмит. Най-лесно е да гледаме на HEAD като на snapshot на **последния ни къмит в този клон**.

В действителност, лесно е да видим как изглежда този snapshot. Ето пример за извличане на реалния листинг на директория и SHA-1 чексумите за всеки файл в HEAD snapshot-а:

```
$ 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
```

Командите на Git `cat-file` и `ls-tree` са “plumbing” команди използвани за неща на по-ниско ниво и рядко се използват в ежедневната работа, но ни помагат да видим какво се случва тук.

Индексът

Индексът е **очаквания следващ къмит**. Наричаме го още “Staging Area” понеже това е мястото, от което Git взема данни, когато изпълните `git commit`.

Git попълва индекса със списък от съдържанието на всички файлове, които последно са били извлечени в работната директория и как са изглеждали те когато първоначално са били извлечени. Вие след това замествате част от файловете с техни актуализирани версии и `git commit` конвертира това в дървото за нов къмит.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0	README
100644 8f94139338f9404f26296befa88755fc2598c289 0	Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0	lib/simplegit.rb
```

Отново, тук използваме `git ls-files`, която е задкулисна команда, показаща ви как изглежда текущия ви индекс.

Технически, индексът не е дървовидна структура — реално той е имплементиран като плосък манифест — но за нашите цели можем да кажем, че прилича на дърво.

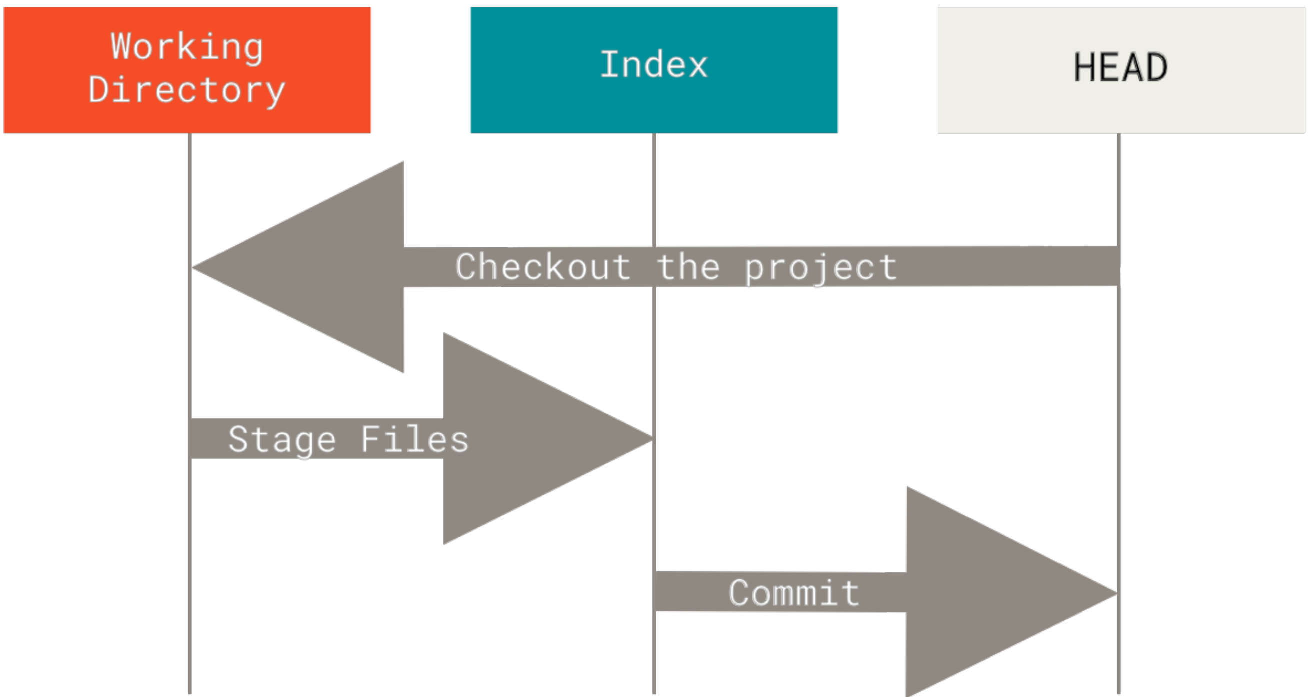
Работната директория

Накрая идва третото Git дърво, *работната ви директория*, известно още като “working tree”. Другите две съхраняват съдържанието си в ефективен, но неудобен за разглеждане вид, в директорията `.git`. Работната директория, от своя страна, разпакетира съдържанието в действителните файлове, с които работим. Можем да гледаме на нея като на **опитно поле**, в което пробваме промените си преди да ги изпратим в индексната област и след това в историята на проекта.

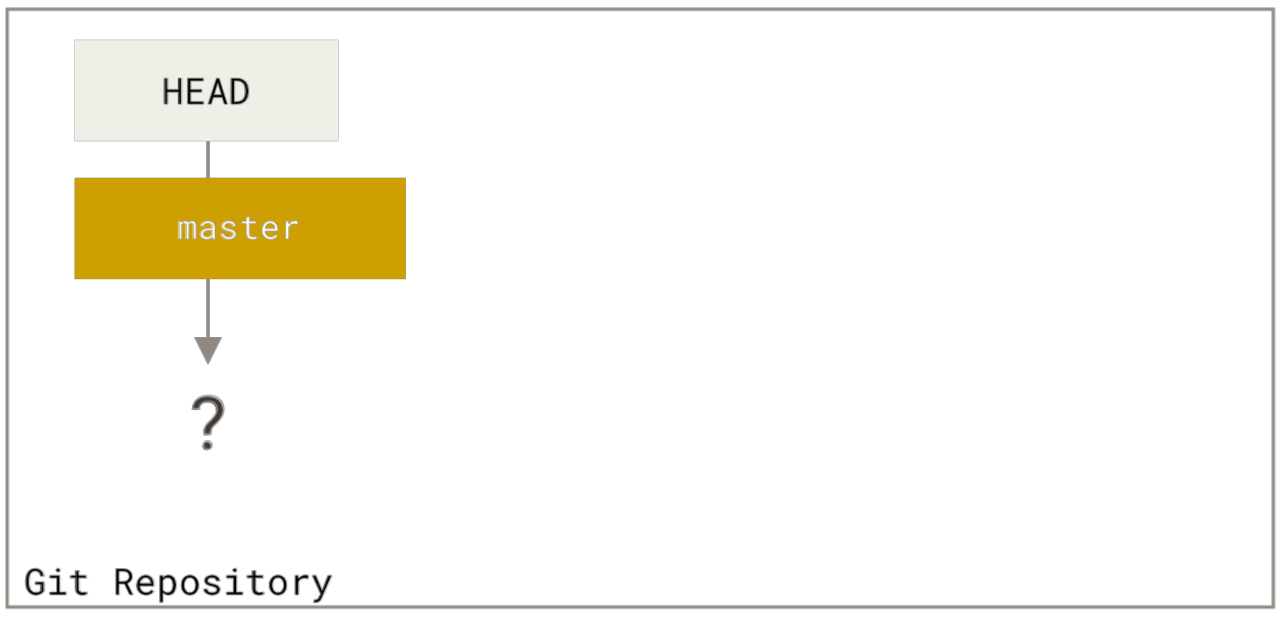
```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

1 directory, 3 files
```

Работният процес

Основната работна последователност на Git е да записва snapshot-и на проекта ни в последователни серии, манипулирайки тези три дървета.

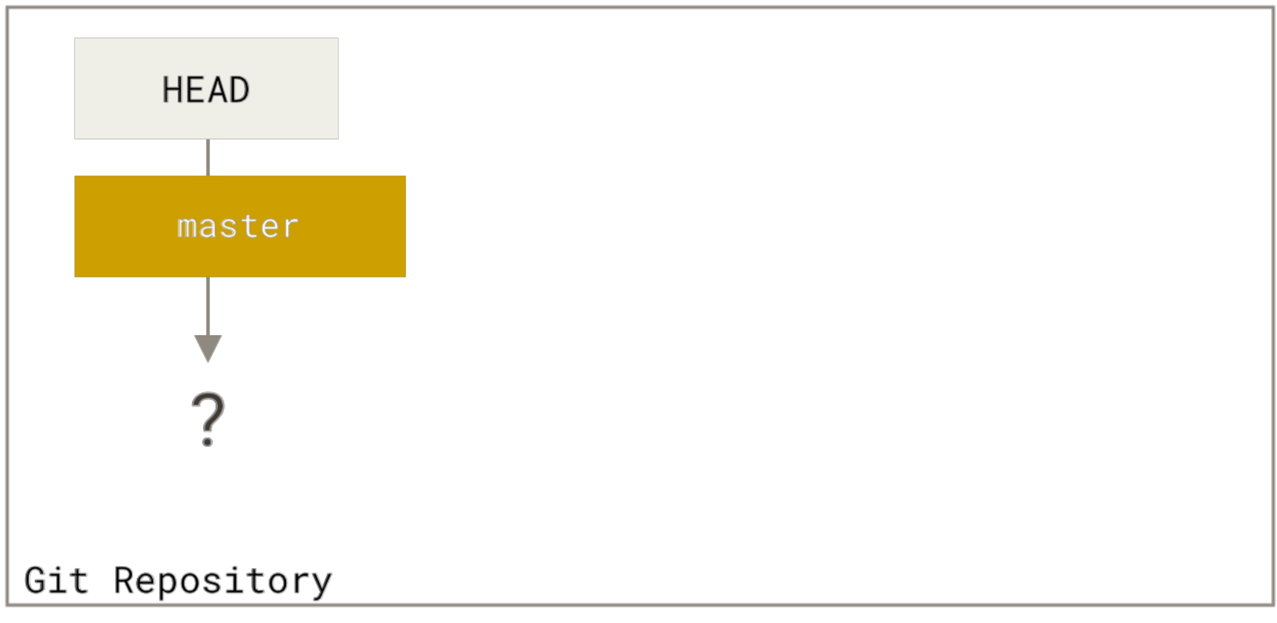


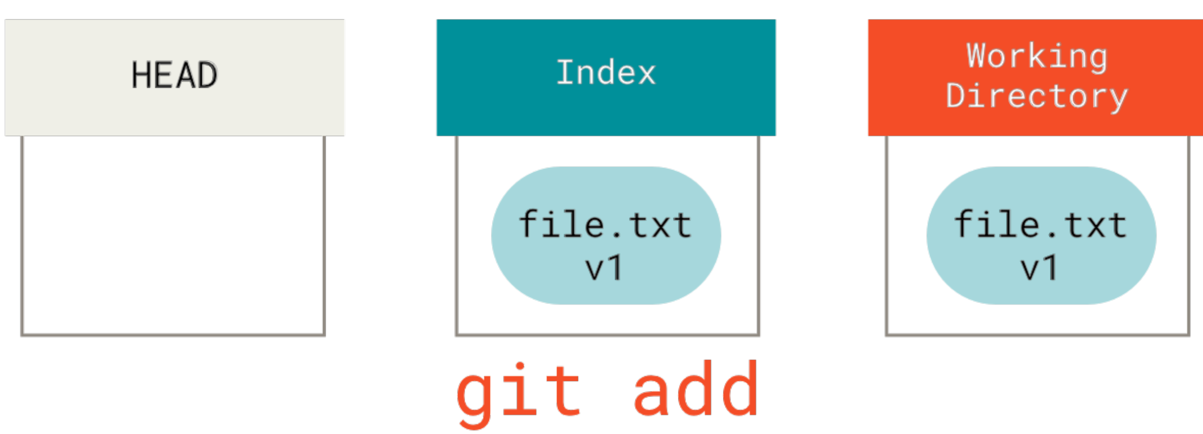
Нека онагледим процеса: да кажем, че отиваме в нова директория с един файл в нея. Ще наречем това **v1** на файла и ще го маркираме в синьо. Сега изпълняваме `git init`, което ще създаде ново Git хранилище с HEAD референция, която сочи към все още несъществуващ клон (`master` все още не е създаден).



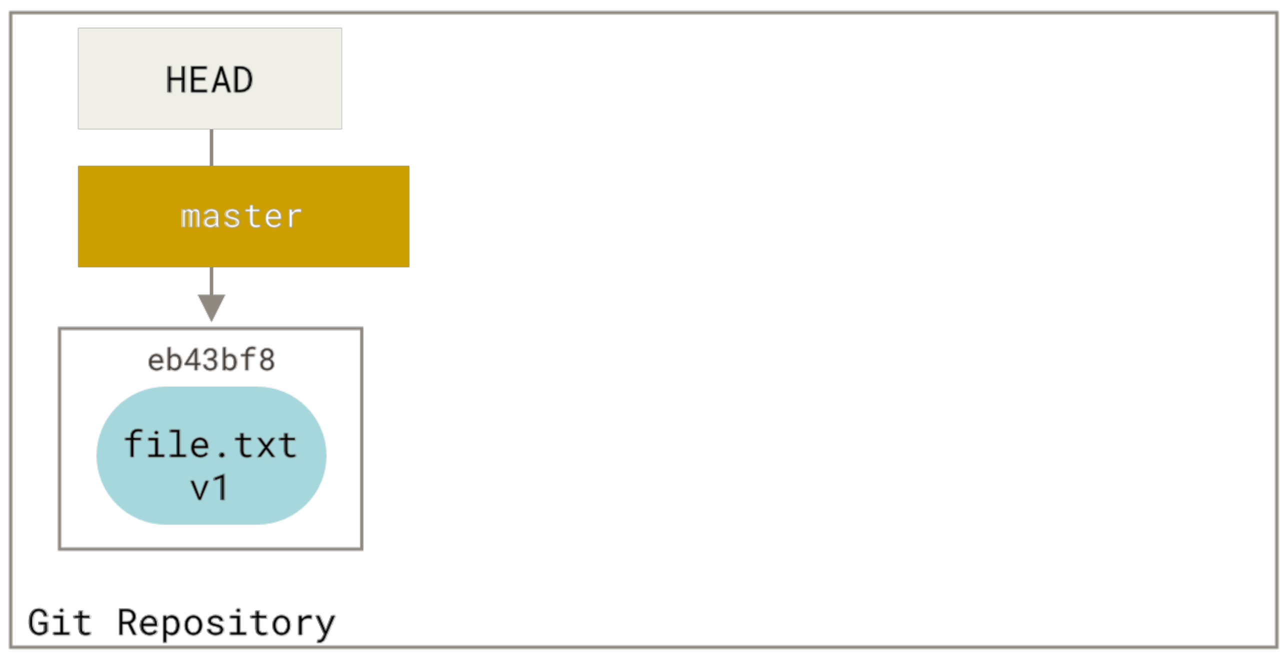
На този етап, единствено работната ни директория има някакво съдържание.

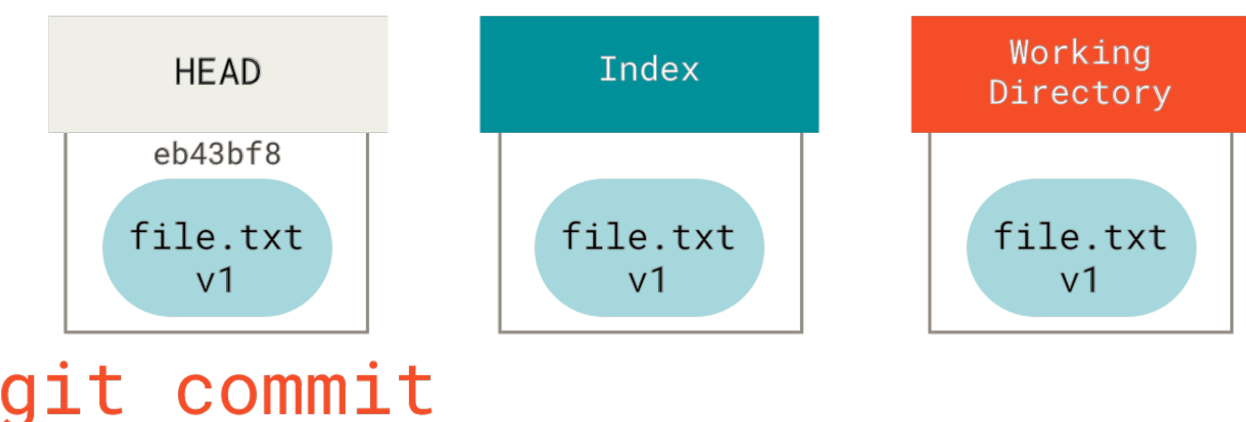
Сега ще искаме да индексираме този файл, така че използваме `git add` за да вземем съдържанието от работната област и да го копираме в индексната.





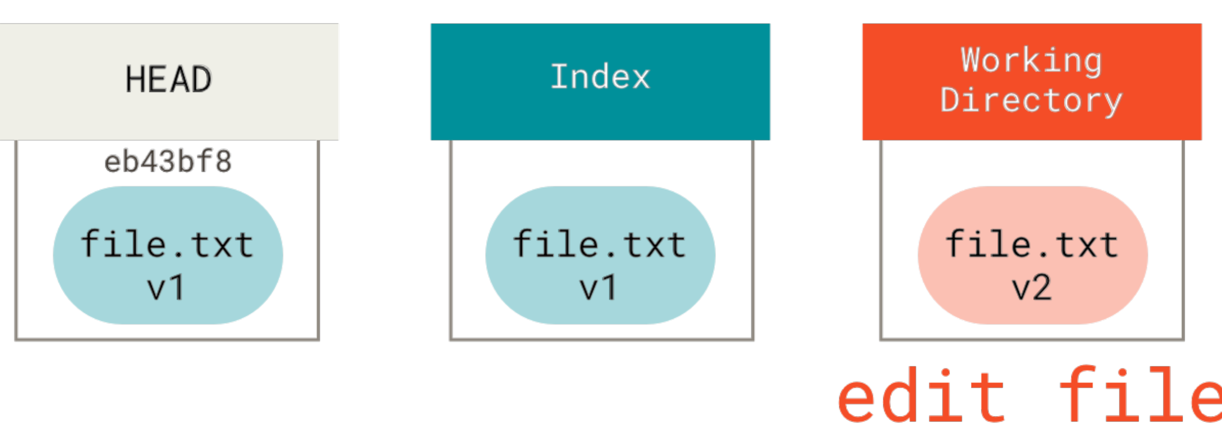
След това, изпълняваме `git commit`, което ще вземе съдържанието на индекса и ще го запише като перманентен snapshot, ще създаде къмит обект, който сочи към този snapshot и ще настрои нашия `master` клон да сочи към този къмит.



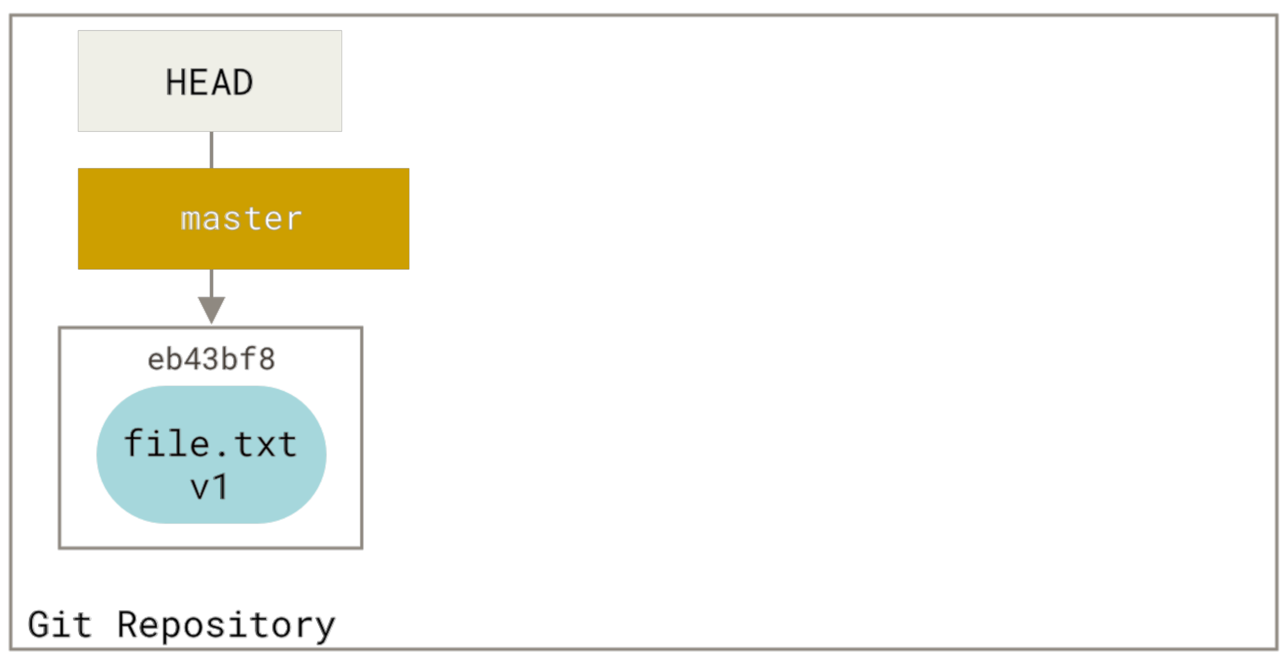


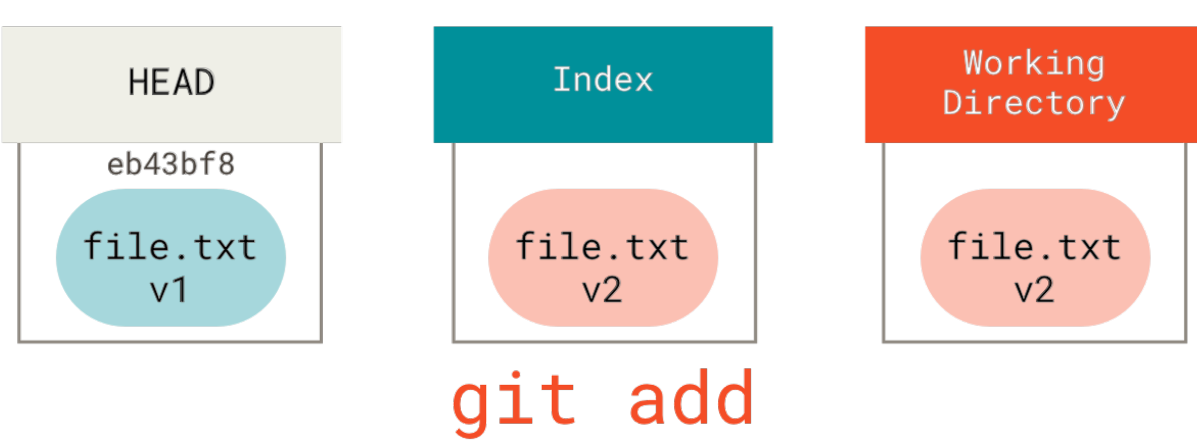
Ако сега изпълним `git status` няма да видим промени, защото трите ни дървета са идентични.

Сега правим промяна по файла и го къмитваме. Ще минем през същия процес, първо променяме файла в работната директория. Нека наречем това **v2** на файла и да го маркираме в червено.

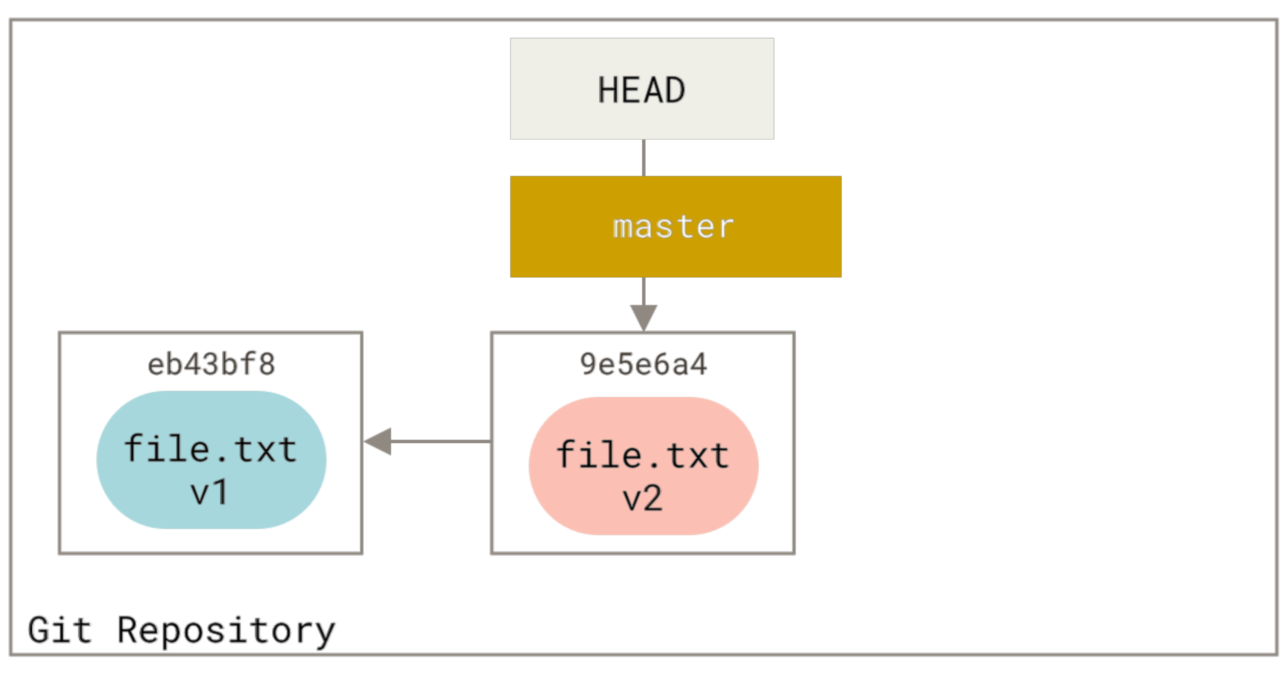


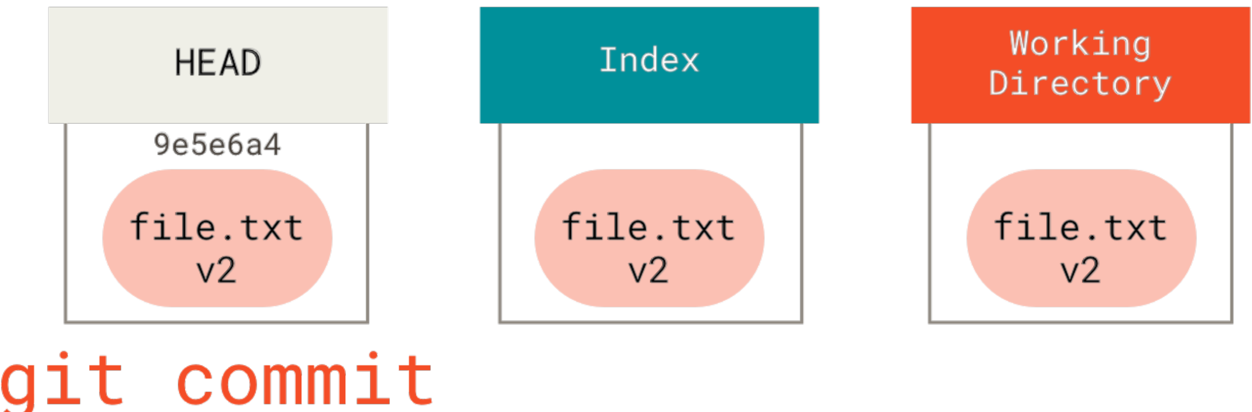
Когато изпълним `git status` в този момент, ще видим този файл в червено в секцията "Changes not staged for commit", защото той сега се различава от копието си в индекса. След това изпълняваме `git add` и го индексираме.





В момента, `git status`, ще ни покаже файла в зелен цвят в секцията “Changes to be committed” защото индексът и HEAD се различават — тоест нашият очаквам следващ къмит е различен от последно съхранения. Последно, изпълняваме `git commit` за да финализираме новия къмит.





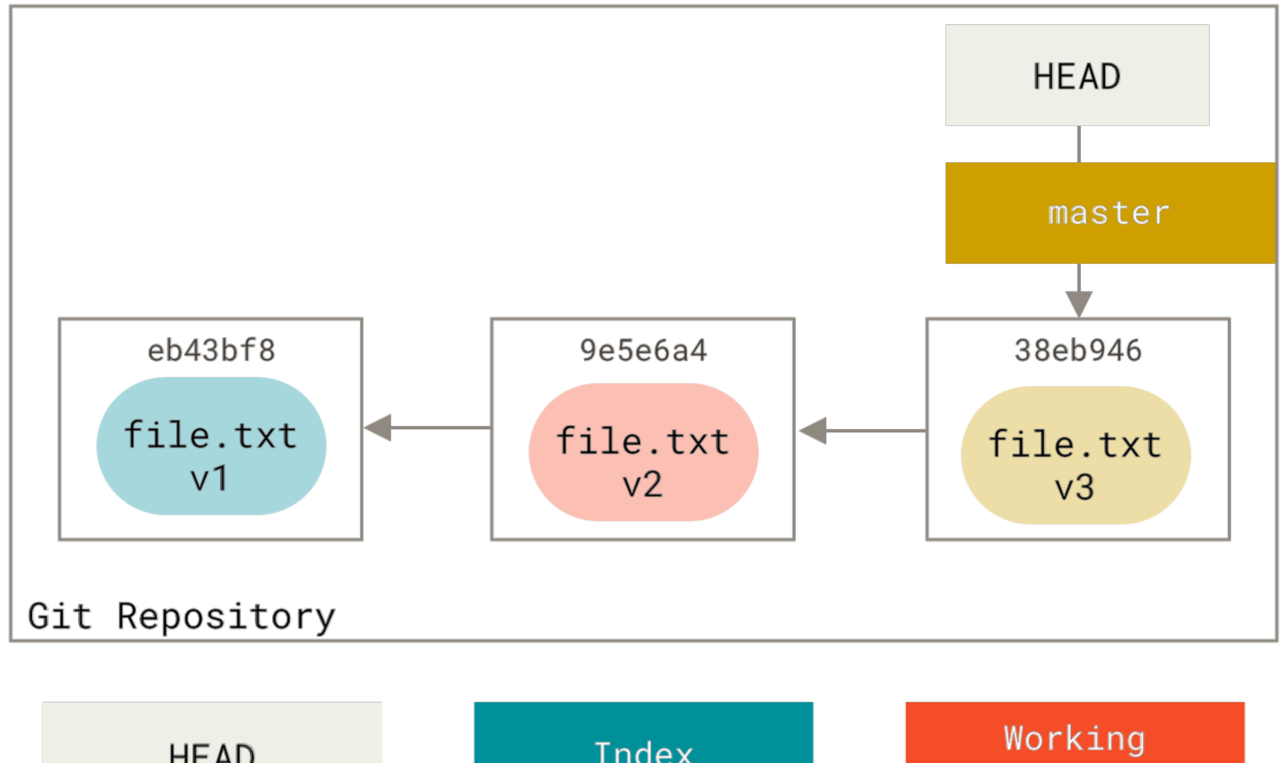
Сега `git status` няма да покаже разлики, защото трите дървета отново са еднакви.

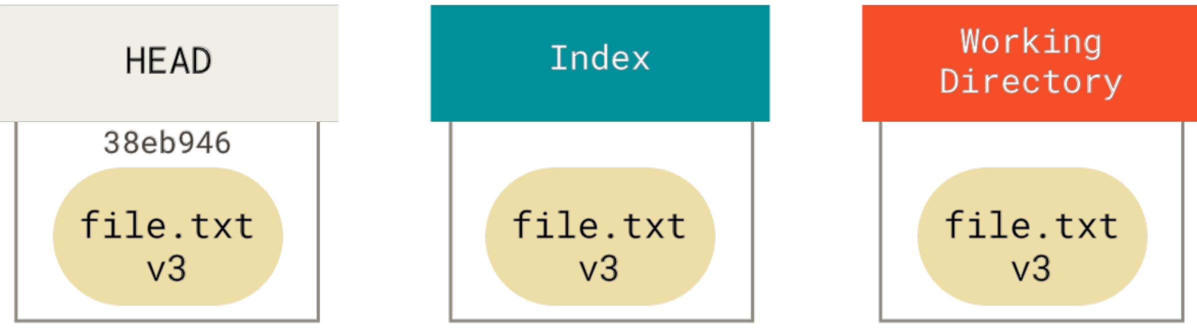
Клонирането и превключването на клонове минават през подобен процес. Когато превключим към друг клон, **HEAD** се променя и сочи към референцията на този клон, **индексът** се попълва със snapshot-а на този къмит и след това съдържанието на **индекса** се копира в **работната директория**.

Ролята на Reset

Командата`reset` придобива по-ясно значение, когато се разглежда в такъв контекст.

За целта на тези примери, нека кажем, че сме променили `file.txt` отново и сме го къмитнали за трети път. Така историята ни сега ще изглежда по този начин:

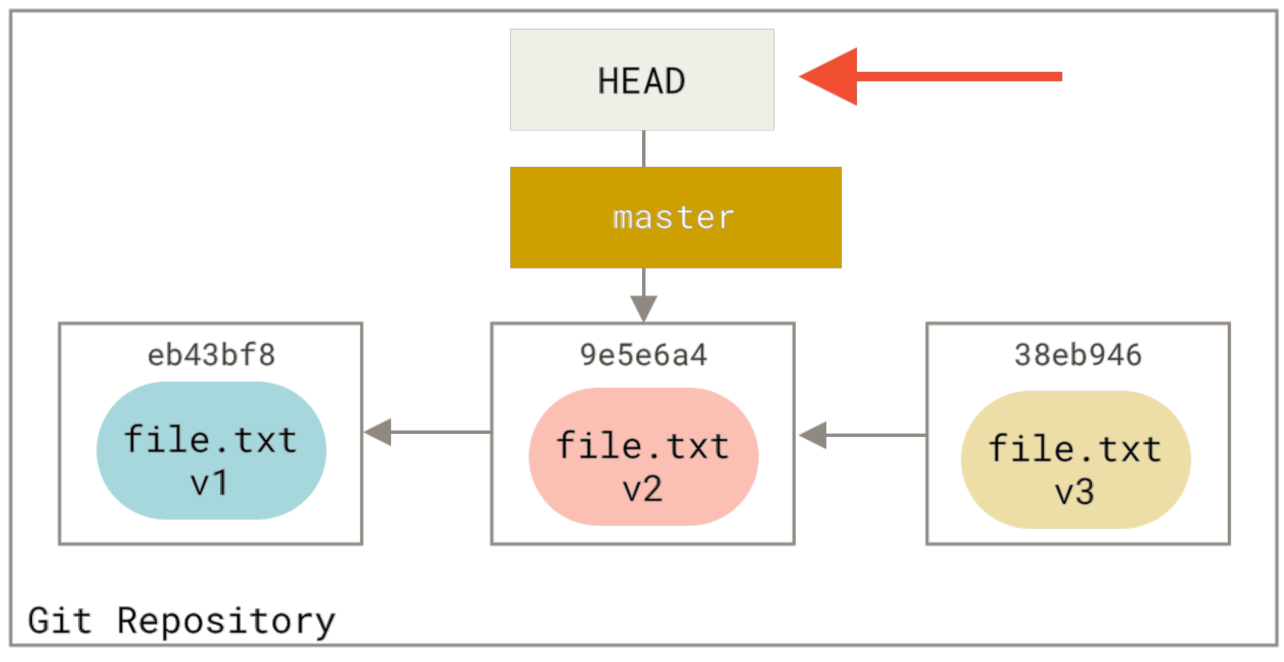


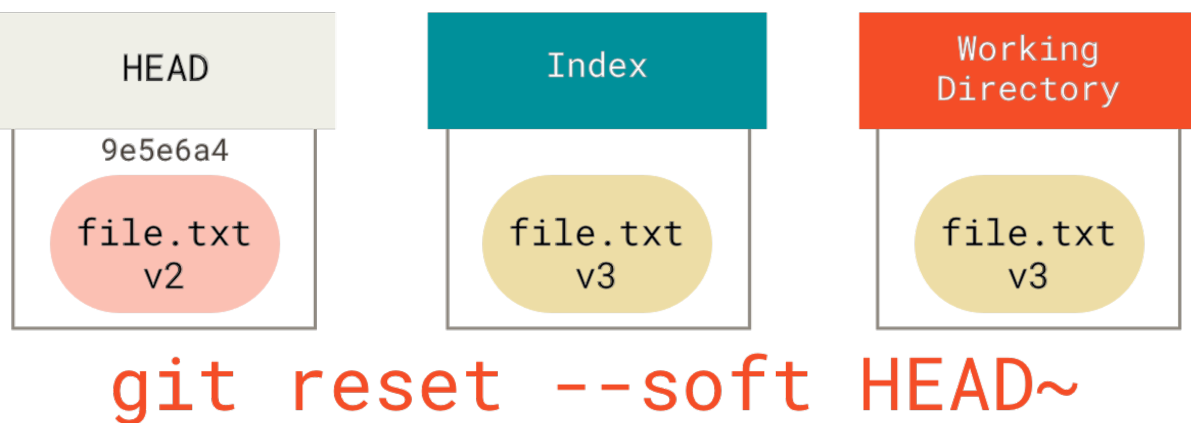


Нека минем през това какво точно прави `reset`, когато я изпълним. Тя директно манипулира тези три дървета по прост и предвидим начин. Извършват се три основни операции.

Стъпка 1: Преместване на HEAD

Първото нещо, което `reset` прави е да смени мястото, където HEAD сочи. Това не означава, че самия HEAD се променя (което става с `checkout`), `reset` премества клона, към който сочи HEAD. Което ще рече, че ако HEAD е на `master` клона (тоест в момента сте в този клон), изпълнението на `git reset 9e5e6a4` ще започне като направи `master` да сочи към `9e5e6a4`.





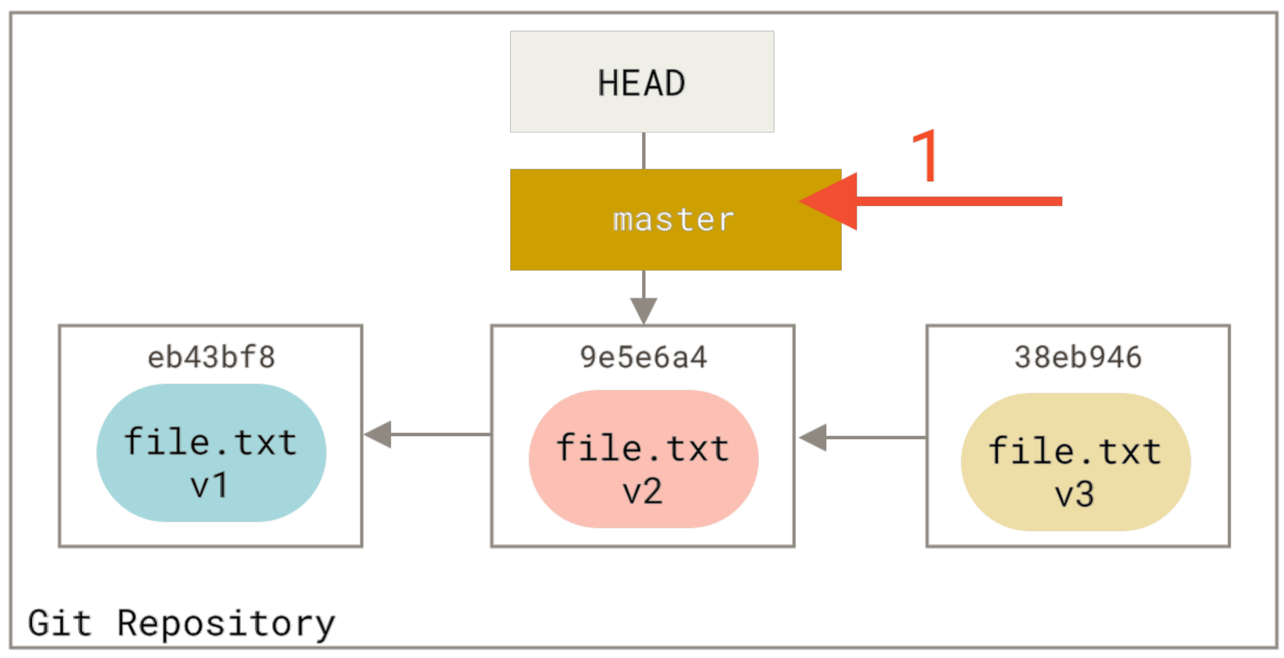
Без значение каква форма на `reset` с къмит сте изпълнили, това е първото нещо, което командата винаги ще опита да направи. С `reset --soft`, тя просто ще завърши тук.

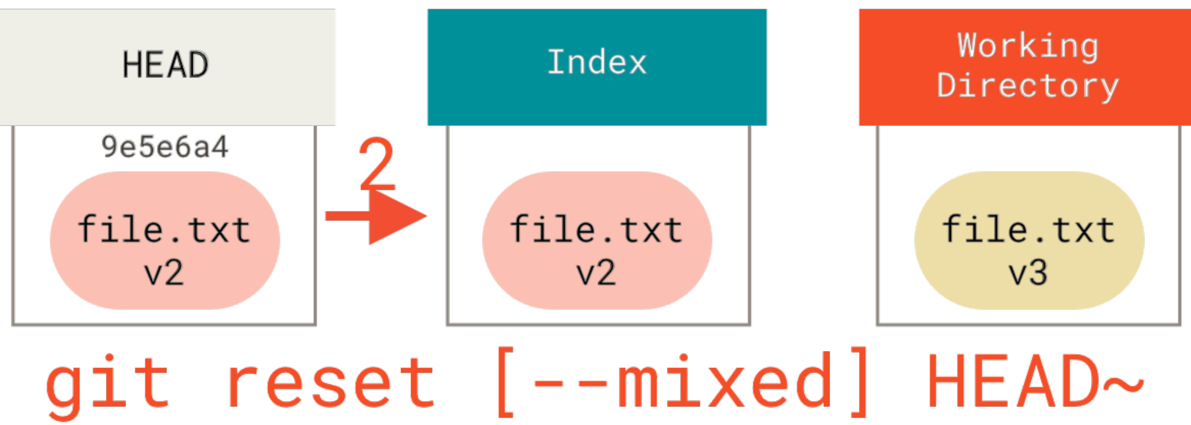
Сега погледнете пак последната диаграма и ще видите какво се е случило: командата практически е отменила последно изпълнената `git commit` команда. Когато изпълните `git commit`, Git създава нов къмит и премества клона, към който сочи HEAD към този къмит. Когато ресетнете обратно към `HEAD~` (тоест родителя на HEAD), вие премествате клона обратно където е бил без да променяте индекса или работната директория. Сега можете да обновите индекса и да изпълните `git commit` отново, така че да постигнете резултата, който бихте имали с `git commit --amend` (вижте Промяна на последния къмит).

Стъпка 2: Обновяване на индекса (--mixed)

Ако сега пуснете `git status`, ще видите в зелено разликата между индекса и новия HEAD.

Следващото нещо, което `reset` ще направи е да обнови индекса със съдържанието на snapshot-а, към който вече сочи HEAD.



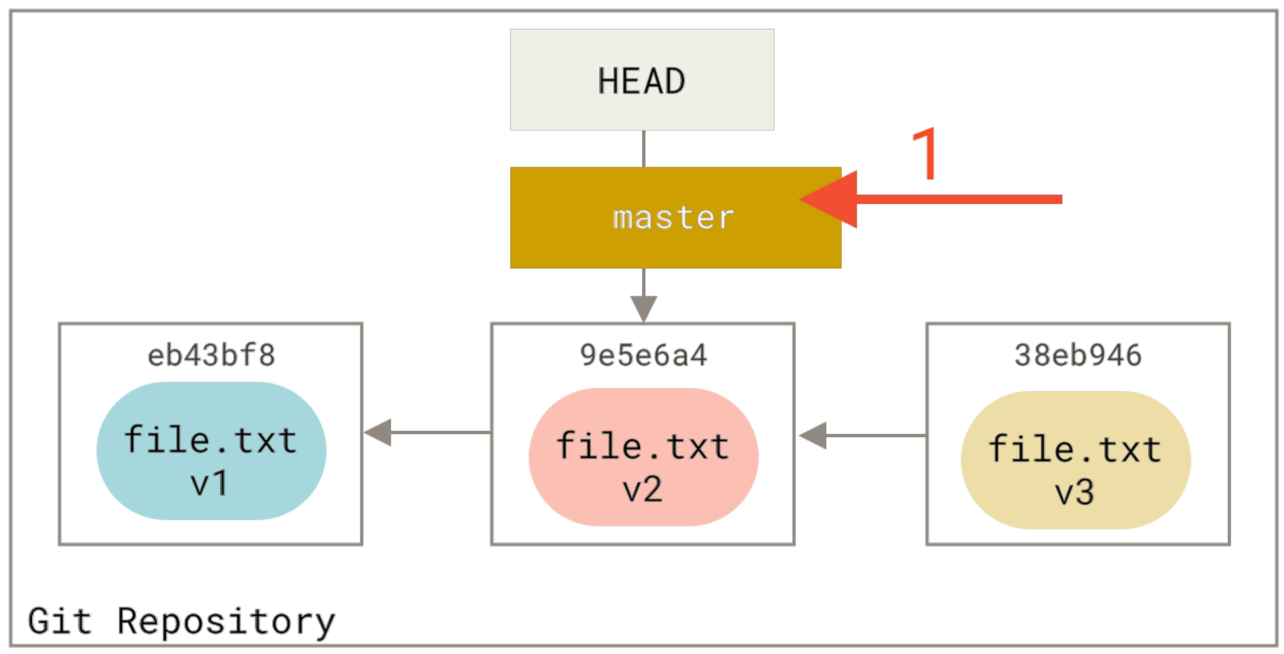


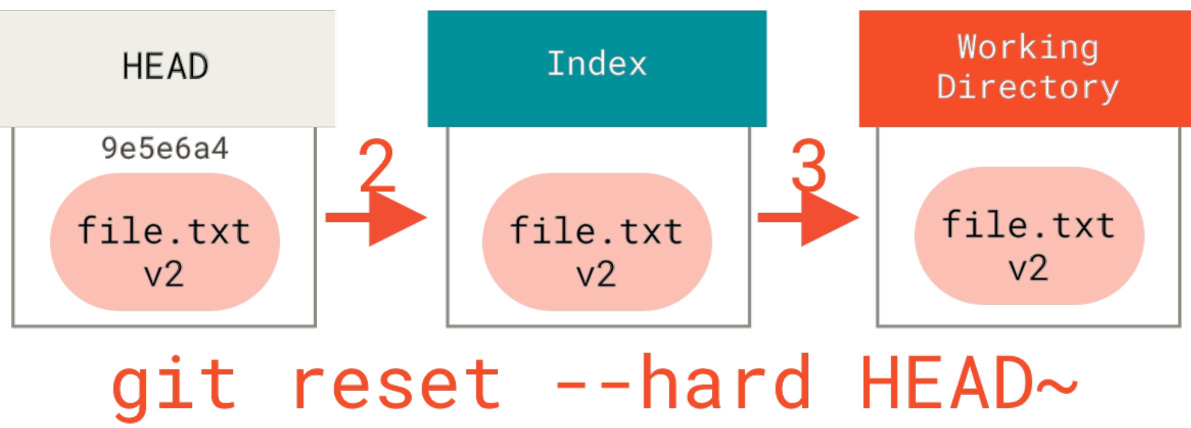
Ако подадете аргумента `--mixed`, `reset` ще спре процеса в тази точка. Този аргумент се подразбира, така че ако не подадете никакви аргументи, а просто изпълните `git reset HEAD~`, това е точката в която командата ще спре процеса.

Поглеждайки отново диаграмата, осъзнаваме че командата пак е отменила последната `commit`, но в допълнение е *деиндексирала* всичко. По същество сега се върнахте обратно до момента преди изпълнението на командите `git add` и `git commit`.

Стъпка 3: Обновяване на работната директория (--hard)

Третото нещо, което командата `reset` може да стори, е да обнови съдържанието на работната директория така, че да я направи като индексната. Ако подадете параметъра `--hard` тя ще стигне чак до там.





Нека да помислим какво се случи току що. Вие отменихте последния къмит (командите `git add` и `git commit`) **и също така** цялата работа, която сте свършили в работната си област.

Важно е да подчертаем, че параметърът `--hard` е единственият, който може да направи командата `reset` наистина опасна и е едно от нещата, което могат да причинят загуба на данни в Git. Всяко друго `reset` изпълнение може лесно да се отмени, но опцията `--hard` не може, тя безвъзвратно презаписва файловете в работната директория. В този примерен случай, ние все още имаме **v3** версията на файла в къмит в нашата база данни на Git и бихме могли да го извлечем поглеждайки в `reflog`-а ни, но ако не бяхме го къмитнали преди, Git щеше да го презапише без връщане назад.

Обобщение

Командата `reset` презаписва съдържанието на трите дървета в специфичен ред, спирайки там, където сме ѝ указали:

1. Премества клона, към който сочи HEAD *(спира дотук с параметъра `--soft`)*.
2. Модифицира индекса да изглежда като HEAD *(спира дотук, ако не е подаден параметър `--hard`)*.

3. Модифицира работната директория да изглежда като индексната област.

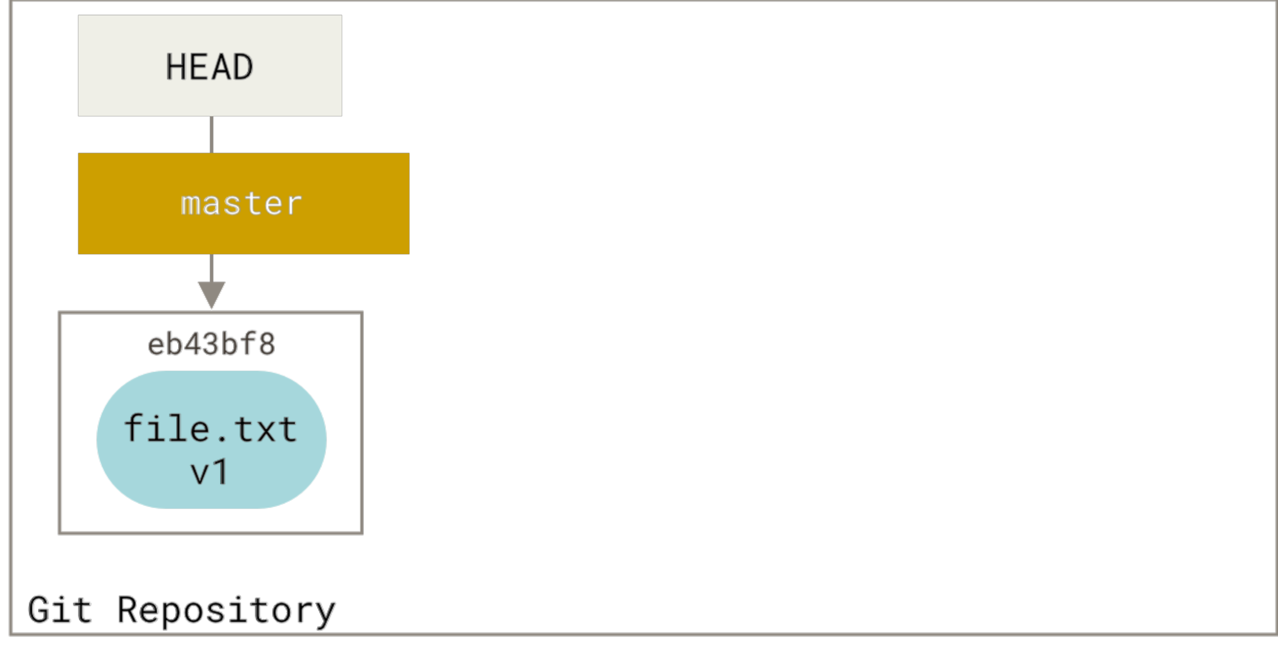
Reset с път

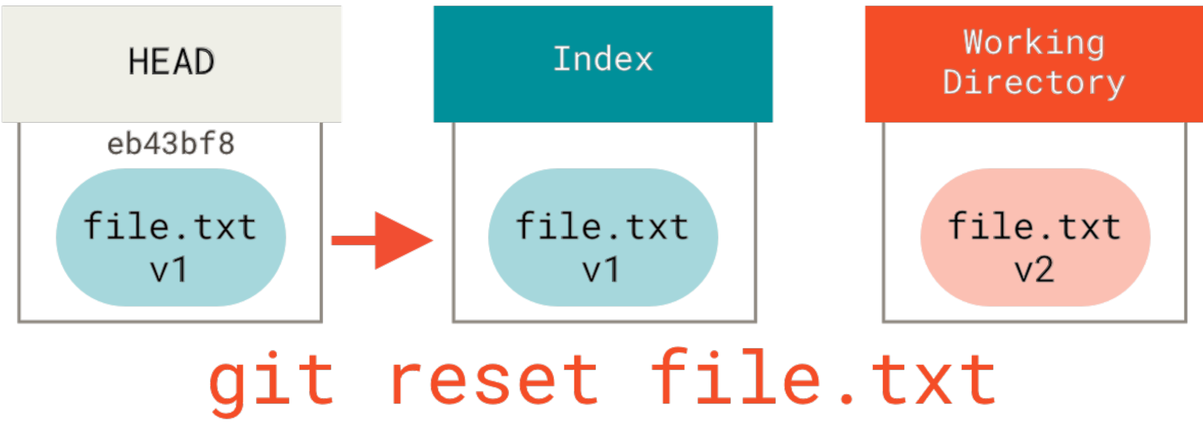
Дотук разгледахме `reset` в основната ѝ форма, но можете също така да ѝ посочите път, по който да работи. Ако укажете път, `reset` ще пропусне стъпка 1 и ще ограничи действието си до специфичен файл/файлове. В това всъщност има смисъл — HEAD е просто указател и вие не можете да сочите част от един къмит и част от друг. Но индексът и работната директория *могат* да бъдат частично обновени, така че reset преминава към стъпки 2 и 3.

Да допуснем, че сте изпълнили `git reset file.txt`. Тази форма (понеже не сте указали SHA-1 на къмит или клон, както и параметрите `--soft` или `--hard`) е съкратена версия на командата `git reset --mixed HEAD file.txt`, която:

1. Ще премести клона, към който сочи HEAD *(пропуска се)*.
2. Ще направи индекса да изглежда като HEAD *(спира тук)*.

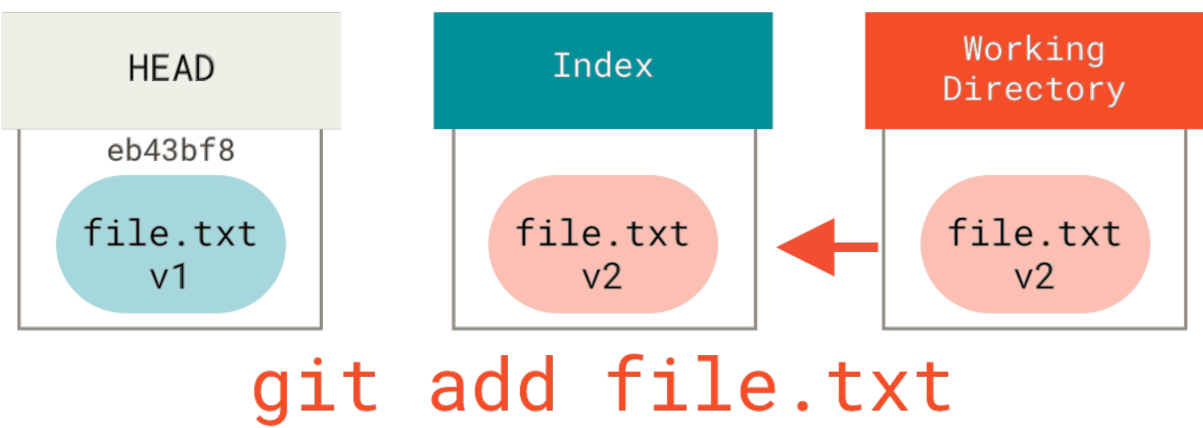
Така практически командата просто копира `file.txt` от HEAD в индекса.





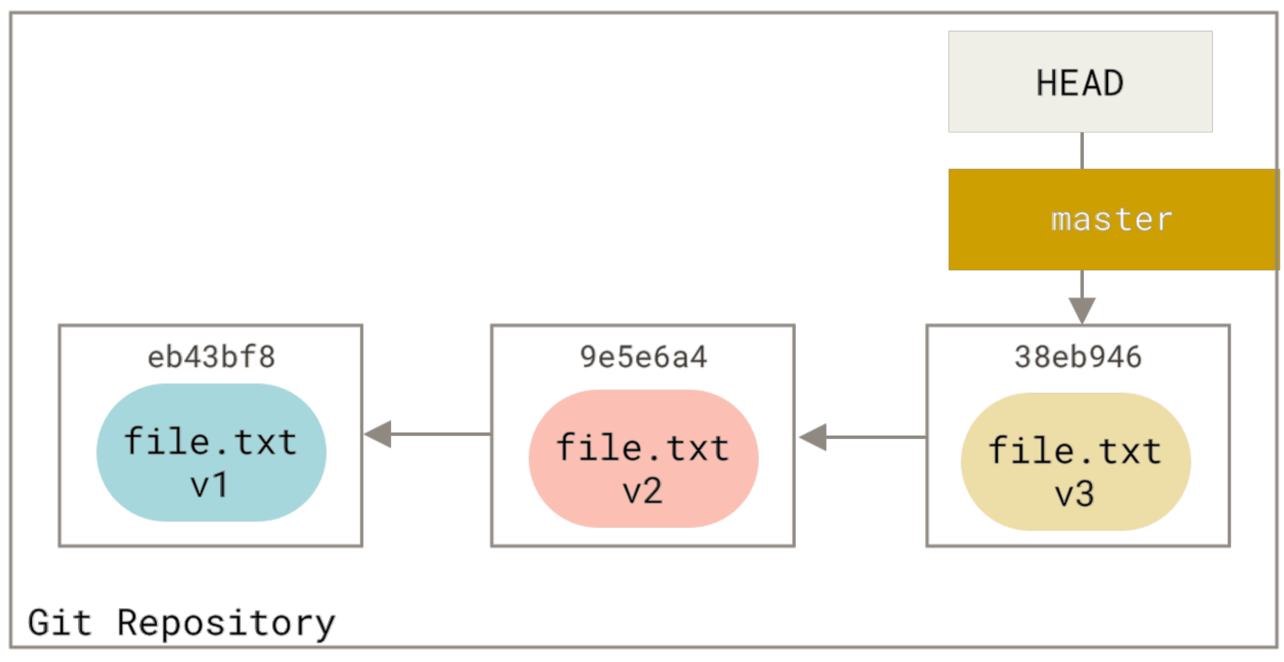
Ефективно това *деиндексира* файла. Ако погледнем диаграмата за тази команда и помислим

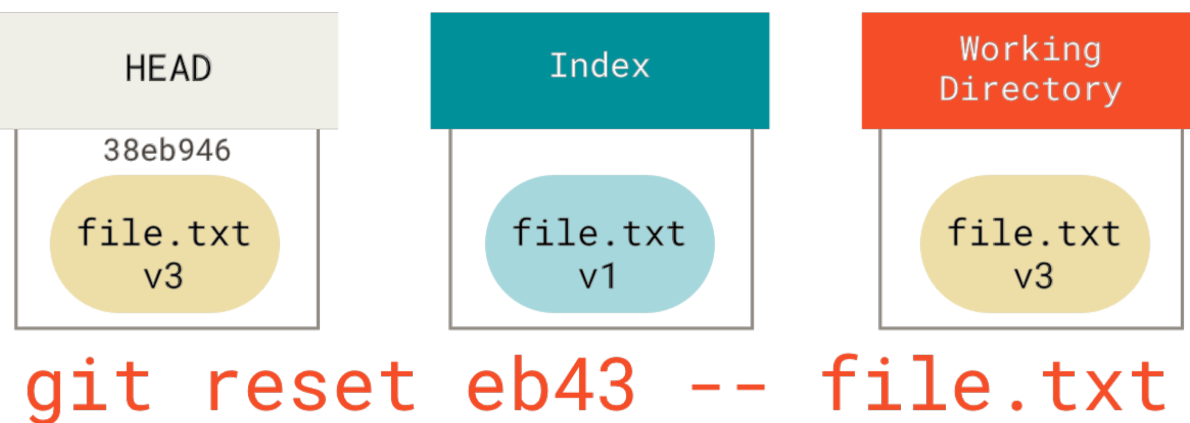
какво прави git add, ще установим че те работят точно по обратен начин.



Това е причината, поради която изходът на git status ви съветва да направите това за да деиндексирате файл (вижте Изваждане на файл от индекса за подробности).

Можем също толкова лесно да кажем на Git да "не изтегля данните от HEAD" указвайки специфичен къмит, от който да извлечем файла вместо това. В такива случаи изпълняваме нещо като git reset eb43bf file.txt.





Това ще направи същото нещо, както ако бяхме върнали назад съдържанието на файла до версията **v1** в работната директория, бяхме изпълнили `git add` върху него и след това го бяхме възстановили обратно отново във версия **v3** (без действително да минаваме през всички тези стъпки). Ако сега изпълним `git commit`, тя ще запише промяна, която връща файла до версия **v1**, въпреки че тази версия никога не сме я имали отново в работната директория.

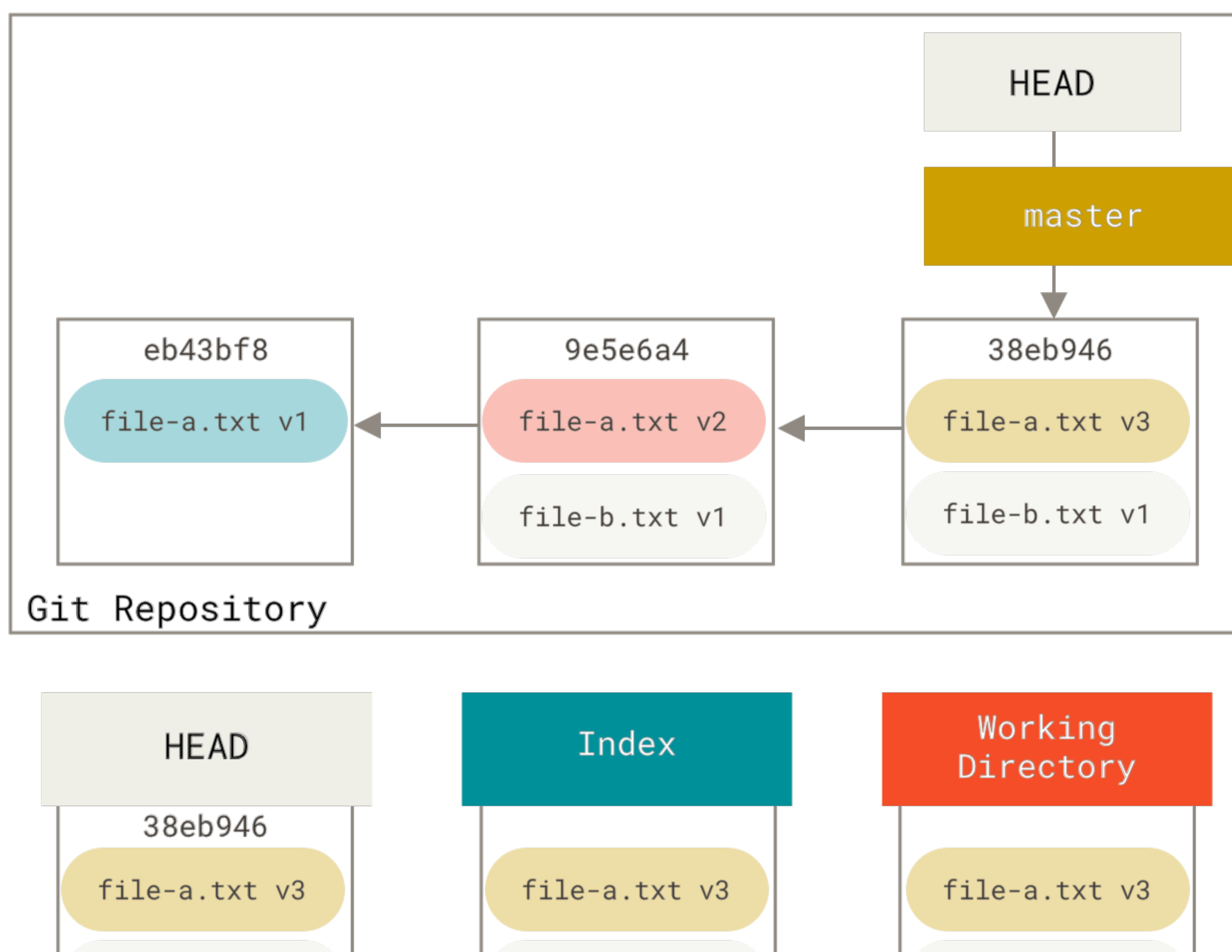
Интересно е да се отбележи и, че подобно на `git add`, `reset` също приема `--patch` аргумент за да деиндексира съдържание в hunk-by-hunk стил. Така можете селективно да деиндексирате или връщате съдържание.

Обединяване

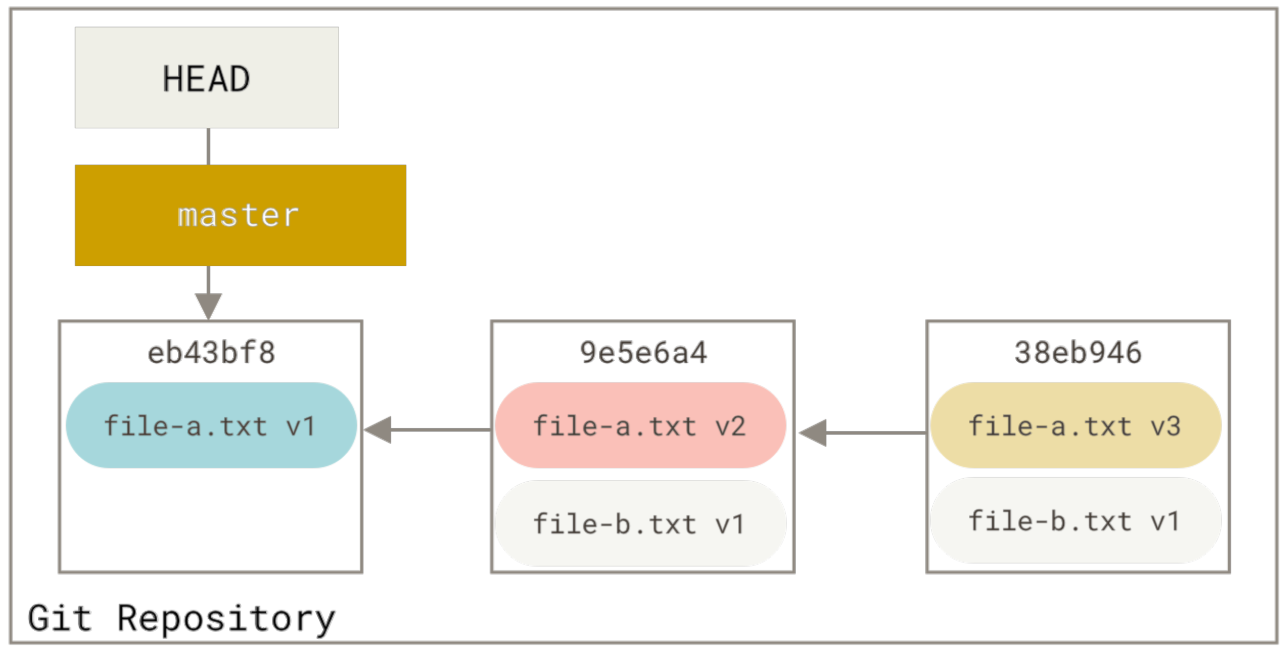
Нека видим как да направим нещо интересно с тази нова функционалност — да обединяваме къмити (squashing).

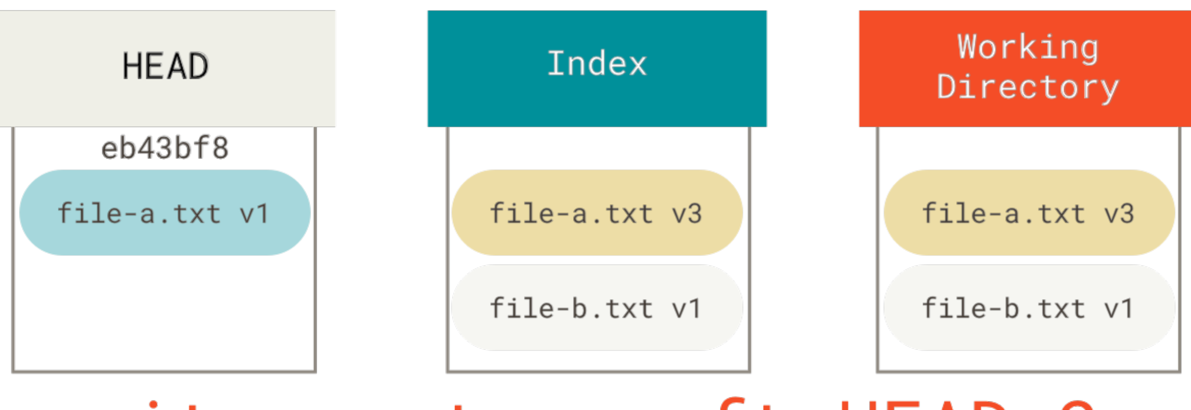
Да кажем, че имате серия къмити със съобщения като “oops.”, “WIP” и “forgot this file”. Можете да използвате `reset` за да ги обедините на бърза ръка в един общ къмит, което ще ви спечели уважение в очите на колегите. Обединяване на къмити показва друг начин да направите това, но в този пример е по-лесно да използваме `reset`.

Да приемем, че имате проект, в който първият къмит има един файл, вторият добавя нов файл и модифицира първия, а третият къмит модифицира първия файл още един път. Вторият къмит е бил work in progress и искате да го обедините с някой друг.

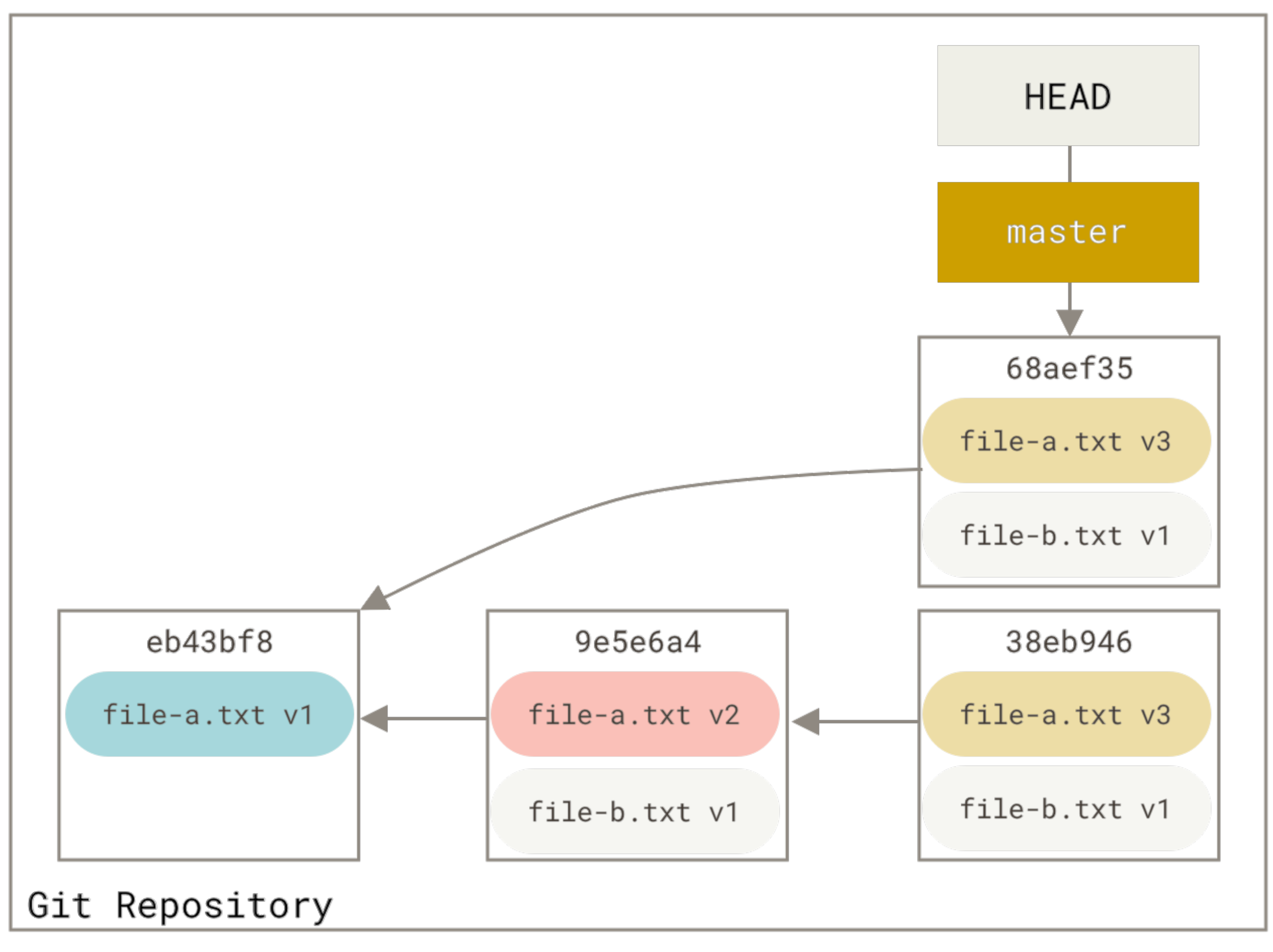


Може да изпълните `git reset --soft HEAD~2` за да преместите HEAD клона назад към по-стар къмит (най-скорошния, който искате да запазите):





След това просто изпълнете `git commit` отново:



Сега може да видите, че достъпната ви история, тази която ще публикувате, вече съдържа един къмит с `file-a.txt` v1 и след това втори, който е модифицирал `file-a.txt` до версия v3 и е добавил `file-b.txt`. Къмитът с версия v2 на файла вече е извън историята.

Check It Out

Накрая, може да се запитате каква е разликата между `checkout` и `reset`. Подобно на `reset`, `checkout` манипулира трите дървета и може да е различна в зависимост от това дали ѝ подавате път или не.

Без пътища

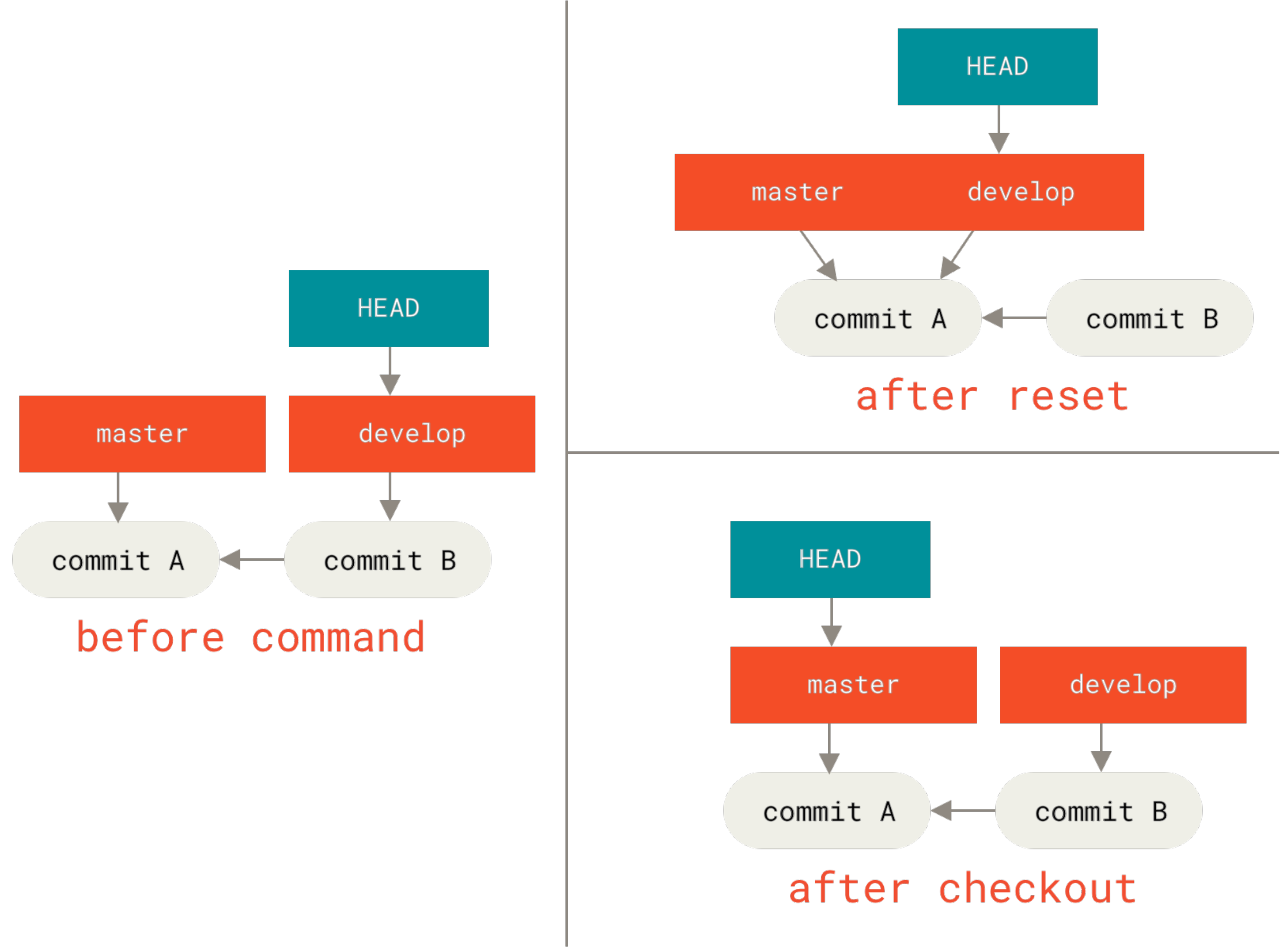
Изпълнението на `git checkout [branch]` е доста подобно по резултат от това на `git reset --hard [branch]` по отношение на това, че обновява всички три дървета, така че да изглеждат като `[branch]`, но с две основни разлики.

Първо, за разлика от `reset --hard`, `checkout` работи безопасно за работната ви директория, тя първо ще се увери, че в нея няма промени преди да превключи към другия клон. Всъщност, нещата са дори още по-интелигентни — командата опитва да направи тривиално сливане в работната директория, така че всички файлове, които *не сте променили* ще бъдат обновени. `reset --hard`, от друга страна, просто ще презапише всичко без никаква проверка.

Другата важна разлика е в това как `checkout` обновява HEAD. Докато `reset` ще премести клона, към който сочи HEAD, `checkout` ще премести самия HEAD да сочи към друг клон.

Нека имаме клонове `master` и `develop`, които сочат към различни къмити и се намираме в `develop` (HEAD сочи там). Ако изпълним `git reset master`, `develop` сега ще сочи към същия къмит, към който сочи `master`. Ако вместо това изпълним `git checkout master`, `develop` не се променя, мести се само HEAD. HEAD сега ще сочи към `master`.

Така и в двата случая променяме HEAD да сочи към commit A, но *начинът*, по който го правим е много различен. `reset` ще премести клона, към който сочи HEAD, `checkout` ще премести самия HEAD.



С пътища

Друг начин да изпълним `checkout` е с път към файл, което както и `reset`, не премества HEAD. То е точно като `git reset [branch] file` по смисъла на това, че обновява индекса с този файл в този къмит, но в допълнение на това презаписва и файла в работната област. Резултатът ще е подобен на `git reset --hard [branch] file` (ако `reset` ви позволи изпълнението) — не е безопасен за работната директория и не премества HEAD.

В допълнение на това, както `git reset` и `git add`, `checkout` също приема аргумента `--patch`, за да ви позволи селективно извличане на част от файл в hunk-by-hunk маниер.

Обобщение

Надяваме се, че сега се чувствате по-удобно с командата `reset`, но въпреки това знаем, че тя предизвиква конфуз, когато я сравнявате с `checkout` и е твърде възможно да забравите всички правила и различни начини на изпълнението ѝ.

Ето кратка таблица с това коя команда кое от дърветата променя. Колоната “HEAD” съдържа “REF”, ако командата отляво премества референцията (клона), към който сочи HEAD и съдържа “HEAD” ако командата премества самия HEAD. Обърнете специално внимание на *WD Safe?* (безопасна за работната директория) колоната — ако тя съдържа **NO**, помислете добре преди да я изпълните.

	HEAD	Index	Workdir	WD Safe?
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	**NO**
`checkout <commit>`	HEAD	YES	YES	YES
File Level				
`reset [commit] <paths>`	NO	YES	NO	YES
`checkout [commit] <paths>`	NO	YES	YES	**NO**

Сливане за напреднали

Обикновено сливането в Git е лесно. Понеже Git позволява сливането на друг клон много пъти, това означава, че можете да имате клон с много дълъг живот, да го поддържате обновен докато работите и да решавате своевременно и често малките конфликти, вместо да трябва да се оправяте с един голям конфликт в края на работата си.

Обаче, понякога възникват по-проблематични конфликти. За разлика от други системи за контрол на версиите, Git не се опитва да бъде прекалено умен що се касае до решаването на merge конфликти. Философията на системата е да е добра в установяването на това дали сливането може да се направи недвусмислено, но ако има конфликт - да не се опитва автоматично да го реши. По тази причина, ако чакате твърде дълго преди да слеете клонове, които се развиват бързо може да се сблъскате с проблеми.

В тази секция ще разгледаме какви биха могли да са някои от тези проблеми и какви инструменти имате на разположение за да се оправите с тях. Ще разгледаме и някои по-различни, нестандартни типове сливане, които можете да направите, както и ще видим как да отмените сливания.

Конфликти при сливане

Въпреки, че вече видяхме основните стъпки за разрешаване на конфликти в Конфликти при сливане, при по-заплетените такива Git осигурява инструменти, с които да установите какво точно се е случило и как по-добре да се справите с проблема.

Преди всичко, ако е възможно, уверете се, че работната ви директория е чиста, преди да опитате сливане, което може да доведе до конфликти. Ако имате текуща работа, опитайте да я къмитнете във временен клон или да я маскирате (stashing). Това ви гарантира, че ще можете да отмените **всичко**, което предстои да опитате. Ако имате незаписани промени в работната директория когато опитате сливане, някои от следващите действия могат да ви помогнат да съхраните тази работа.

Нека видим един прост пример. Имаме кратък Ruby файл, който отпечатва *hello world*.

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

В нашето хранилище, създаваме нов клон наречен `whitespace` и променяме Unix символите за край на ред в DOS такива ефективно модифицирайки всеки един ред но само с празни символи. След това сменяме реда “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 'Convert hello.rb to DOS'
[whitespace 3270f76] Convert 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 'Use Spanish instead of English'
[whitespace 6d338d2] Use Spanish instead of English
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Сега превключаме обратно към `master` клона и добавяме малко документация за функцията.

```
$ 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 'Add comment documenting the function'
[master bec6336] Add comment documenting the function
 1 file changed, 1 insertion(+)
```

Опитваме да слеем клона `whitespace` и изпадаме в конфликтна ситуация заради whitespace промените.

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

Прекъсване на сливане

Сега имаме няколко възможности. Първо, нека видим как да излезем от ситуацията, връщайки се в предишното състояние на хранилището. Ако не сте очаквали конфликти и не желаете да ги оправяте в момента, можете просто да откажете сливането с `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

Командата `git merge --abort` опитва да върне статуса, в който сте били преди да опитате сливането. Като казваме опитва, единствените случаи, в които не би успяла е ако имате

немаскирани (unstashed) или некъмитнати промени в работната директория. Във всички останали случаи тя ще работи коректно.

Ако по някаква причина просто искате да започнете отначало, можете също така да изпълните `git reset --hard HEAD` и работната ви директория ще бъде върната до последния къмитнат статус. Помнете, че всяка некъмитнатата работа ще бъде загубена, така че се уверете, че промените наистина не ви трябват.

Игнориране на празните символи

В този конкретен случай проблемите ни възникваха от празните символи във файла. Знаем това, защото случаят е прост, но също така ситуацията е лесно откриваема и в реални случаи, когато разглеждаме причината за конфликта, понеже се вижда, че всеки ред е премахнат от едната страна и повторно добавен в другата. По подразбиране, Git вижда всички тези редове като променени и отказва сливането.

Стратегията за сливане по подразбиране също приема аргументи и някои от тях подпомагат игнорирането на празните символи. Ако виждате, че имате много whitespace проблеми в сливането, можете просто да го откажете и да го опитате отново, този път с аргумента `-Xignore-all-space` или `-Xignore-space-change`. Първата опция игнорира празните символи **изцяло** при сравняването на редовете, докато втората третира последователностите от един или повече празни символи като еквивалентни.

```
$ 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(-)
```

Сега действителните промени по файла не водят до конфликт и сливането минава чисто.

Това може да ви измъкне в ситуации, когато някой член на екипа, любител на преформатирането, внезапно реши да смени интервалите с табулации или обратно.

Ръчно повторно сливане на файлове

Git се оправя добре с обработката на празни символи, но има други типове промени, с които вероятно няма да се оправи сам, но които вие знаете, че можете лесно да коригирате със скрипт. Например, нека кажем че Git хипотетично не може да обработи whitespace промените и трябва да направим това на ръка.

Това, което трябва действително да направим е да прекараме файла през програмата `dos2unix` преди да опитаме сливането. Как можем да направим това?

Първо, изпадаме в конфликтната ситуация. След това, искаме да извлечем собствената версия на файла, версията от клона, който сме опитали да слеем, както и общата версия (тази от която двата клона са стартирани). След това, искаме да поправим или нашата или другата версия и да опитаме отново сливане за само този единичен файл.

Извличането на трите версии в действителност е лесно. Git съхранява всички тях в индекса под формата на “етапи (stages)” като всеки от тях има съответен номер. Stage 1 е общия файл (common), от който произлизат другите два, stage 2 е вашата версия (ours) и stage 3 е версията от `MERGE_HEAD`, тоест от клона който опитвате да слеете (theirs).

Можете да извлечете копие от всяка от тези версии на конфликтния файл с командата `git show` и специален синтаксис.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Ако искате да видите повече подробности, можете да използвате plumbing командата `ls-files -u` за да получите действителните SHA-1 стойности за всеки от тези файлове

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1	hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2	hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3	hello.rb
```

Изразът `:1:hello.rb` е просто съкратен начин да потърсите blob обекта със съответния SHA-1 хеш.

След като вече имаме съдържанието на трите версии на файла в работната директория, можем ръчно да поправим whitespace проблема във файла от клона, който опитваме да слеем и след това да опитаме цялото сливане отново с малко позната команда `git merge-file`.

```
$ 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()
```

В този момент успешно сляхме файла. В действителност, това работи по-добре от `ignore-space-change` аргумента защото реално поправя промените с празните символи преди сливането, вместо просто да ги игнорира. При `ignore-space-change` сливането получихме няколко реда с DOS line ending символи, което смесва нещата и не изглежда красиво.

Ако преди да завършите къмита искате да получите представа за това какво действително е променено между едната страна или другата, можете да поискате от `git diff` да сравни намиращото се в работната ви директория (и което ще къмитнете) с всяка от гореописаните три версии. Нека видим всички сравнения.

За да сравните резултата с това, което сте имали във вашия клон преди сливането, с други думи да видите какво е въвело сливането, можете да изпълните `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()
```

Така тук лесно можем да видим, че това което се е случило с този файл в резултат на сливането е промяната на един единствен ред.

Ако искаме да видим разликите от сливането с версията от другия клон, изпълняваме `git diff --theirs`. В този и следващия пример, използваме флага `-b` за да изключим празните символи, защото сравняваме с това, което е в Git, а не с почистения `hello.theirs.rb` файл.

```
$ 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
```

Накрая, можем да проверим как файлът е бил променен и от двете страни с `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()
```

На този етап можем да използваме `git clean` за да изтрием допълнителните файлове, които създадохме за да осъществим ръчното сливане, те вече не ни трябват.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Извличане на конфликти

Стигайки дотук, по някаква причина може да не сме доволни от решението на конфликта или пък ръчната редакция на едната или другата версия все още не работи добре и се нуждаем от повече контекст.

Нека променим примера малко. В този случай, имаме два продължително развиващи се клона с по няколко къмита всеки, опита за сливане на които води до конфликт по съдържание.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) Update README
* 9af9d3b Create README
* 694971d Update phrase to 'hola world'
| * e3eb223 (mundo) Add more tests
| * 7cff591 Create initial testing script
| * c3ffff1 Change text to 'hello mundo'
|/
* b7dcc89 Initial hello world code
```

Сега имаме три уникални къмита само в `master` клона и три други в клона `mundo`. Ако опитаме да слеееем `mundo`, получаваме конфликт.

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

Искаме да видим какъв точно е конфликта. Ако отворим файла, ще видим нещо такова:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

И двете страни са добавили съдържание към този файл, но някои от къмитите са го модифицирали в едно и също място, което поражда конфликта.

Нека разгледаме няколко инструмента, които биха ни подсказали как е възникнал проблема. Твърде възможно е да изпаднем в ситуация, в която може би да не е очевидно как точно трябва да го разрешим. Трябват ни повече контекстни данни.

Едно полезно средство е командата `git checkout` с параметър `--conflict`. Това ще извлече файла отново и ще замени merge conflict маркерите. Това може да е полезно, ако искате да нулирате маркерите и да опитате да разрешите конфликта отново.

Можете да подадете на `--conflict` или `diff3` или `merge` (което е по подразбиране). Ако подадете `diff3`, Git ще използва малко по-различна версия на маркерите за конфликти и ще ви покаже не само "ours" и "theirs" версиите, но също и "base" версията вътре във файла, за да имате повече информация.

```
$ git checkout --conflict=diff3 hello.rb
```

Изпълнявайки това, файлът ни вече изглежда така:

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

Ако този формат ви харесва, можете да го направите подразбиращ се за бъдещи merge конфликти задавайки стойност `diff3` за настройката `merge.conflictstyle`.

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

Командата `git checkout` също може да приема `--ours` и `--theirs` параметри, което може да е наистина бърз начин за избор на едната или другата страна без въобще да сливаме.

Това може да е особено полезно за конфликти при двоични файлове, където можете просто да изберете едната страна или когато искате да слеете само определени файлове от друг клон — можете да направите сливането и след това да изберете дадени файлове от едната или другата страна преди къмитването.

Дневник на сливанията

Друг полезен инструмент при разрешаване на merge конфликти е `git log`. Командата може да ви помогне да получите представа за обстоятелствата, при които вие самите бихте могли да сте допринесли за конфликта. Разглеждането на малко история за да си припомните защо два паралелни работни процеса модифицират едно и също място в кода може да е много полезно понякога.

За да получим списък на всички уникални къмити интегрирани в клоновете, които участват в сливането, можем да използваме “triple dot” синтаксиса, за който научихме в Три точки.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 Update README
< 9af9d3b Create README
< 694971d Update phrase to 'hola world'
> e3eb223 Add more tests
> 7cff591 Create initial testing script
> c3ffff1 Change text to 'hello mundo'
```

Това е кратък списък на шестте къмита, които засягат сливането както и от коя линия на разработка идват.

Можем да опростим още изхода, така че да получим по-специфичен контекст. Ако добавим параметъра `--merge` към `git log`, ще получим само къмитите от двете страни на сливането, които променят текущо конфликтния файл.

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

Ако вместо това изпълните командата с параметър `-p`, получавате само разликите за файла, маркиран като конфликтен. Това може да е **наистина** полезно, ако бързо трябва да получите контекста, от който да разберете защо е възникнал даден конфликт и как по-интелигентно да го разрешите.

Комбиниран Diff формат

Понеже Git индексира всички успешни резултати от сливанията, когато изпълните `git diff` докато сте в режим на конфликт, ще получите само това, което е все още в статус на конфликт. Това може да ви помогне да видите какво все още ви остава да разрешите.

Изпълнявайки `git diff` в такова положение, командата ви дава информация в специфичен diff изходен формат.

```
$ 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()
```

Форматът е познат като “Combined Diff” и ви дава две колони с информация в началото на всеки ред. Първата колона показва дали редът е различен (добавен или изтрит) между “ours” клона и файла в работната ви директория, а втората колона показва същото но сравнявайки “theirs” клона и копието в работната директория.

Така в този пример можем да видим, че редовете `<<<<<<<` и `>>>>>>>` са в работното копие, но не са били в нито едната от страните участващи в сливането. Това има смисъл, защото сливащият механизъм ги е оставил там за нашия контекст, но ние сме очаквали да ги премахне.

Ако разрешим конфликта и изпълним `git diff` отново, ще видим същото нещо, но малко по-полезно.

```
$ 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()
```

Резултатът ни показва, че “hola world” е бил в нашия клон, но не е в работното копие, че “hello mundo” е бил в другия клон, но не е в работното копие, и накрая - че “hola mundo” редът не е бил в нито една от страните, но сега е в работното копие. Това би могло да е от помощ като финален преглед преди да къмитнете решението на конфликта.

Същата информация можете да извлечете и от `git log` за всяко едно сливане, за да видите постфактум как даден проблем е бил разрешен. Git ще отпечата на екрана даннните в този формат ако изпълните `git show` за merge къмит или ако добавите аргумента `--cc` към `git log -p` (която по подразбиране показва само пачове за non-merge къмити).

```
$ 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()
```

Отмяна на сливания

След като вече знаете как да създадете сливащ къмит, твърде вероятно е да направите такъв погрешка. Едно от най-добрите неща в Git е, че системата няма проблем с вашите грешки, понеже е възможно (и в много случаи лесно) да ги поправите.

Сливащите къмити не правят изключение. Да кажем, че сте започнали работа по topic клон, по невнимание сте го сляли в `master` клона и сега вашата история изглежда така:

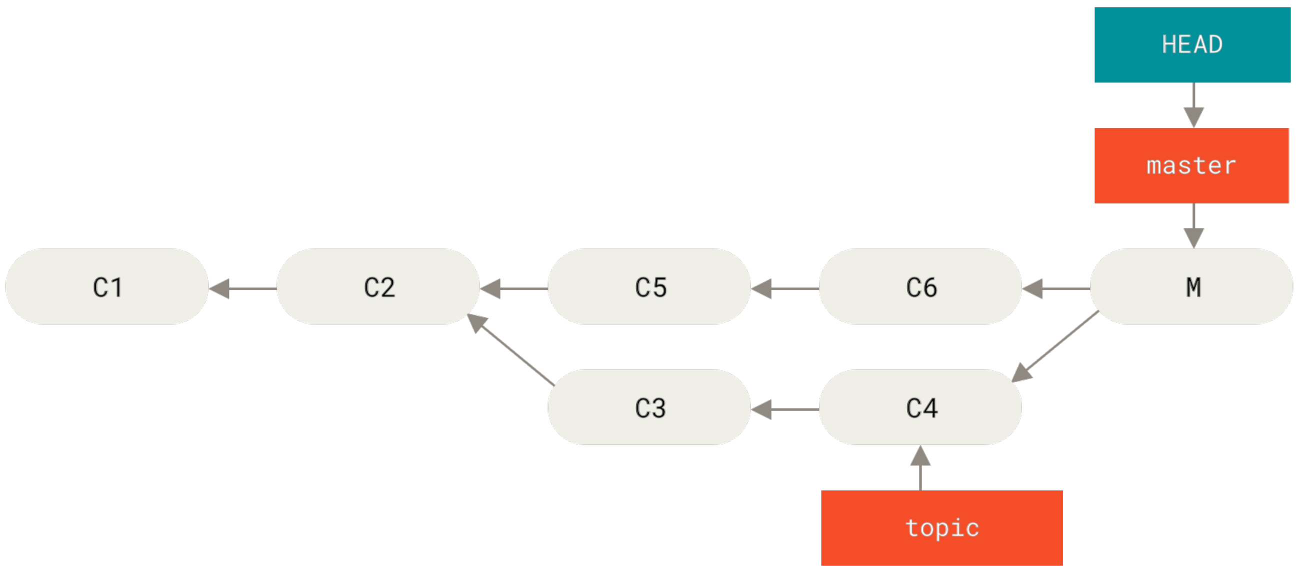


Figure 137. Инцидентен сливащ къмит

Има два подхода за справяне с проблема, в зависимост от желания резултат.

Корекция на референциите

Ако нежеланият къмит съществува само в локалното ви хранилище, най-лесното и добро решение е да преместите клоновете така, че да сочат където искате. В повечето случаи, ако след погрешната `git merge` изпълните `git reset --hard HEAD~`, това ще коригира указателите на клоновете, така че да изглеждат по следния начин:

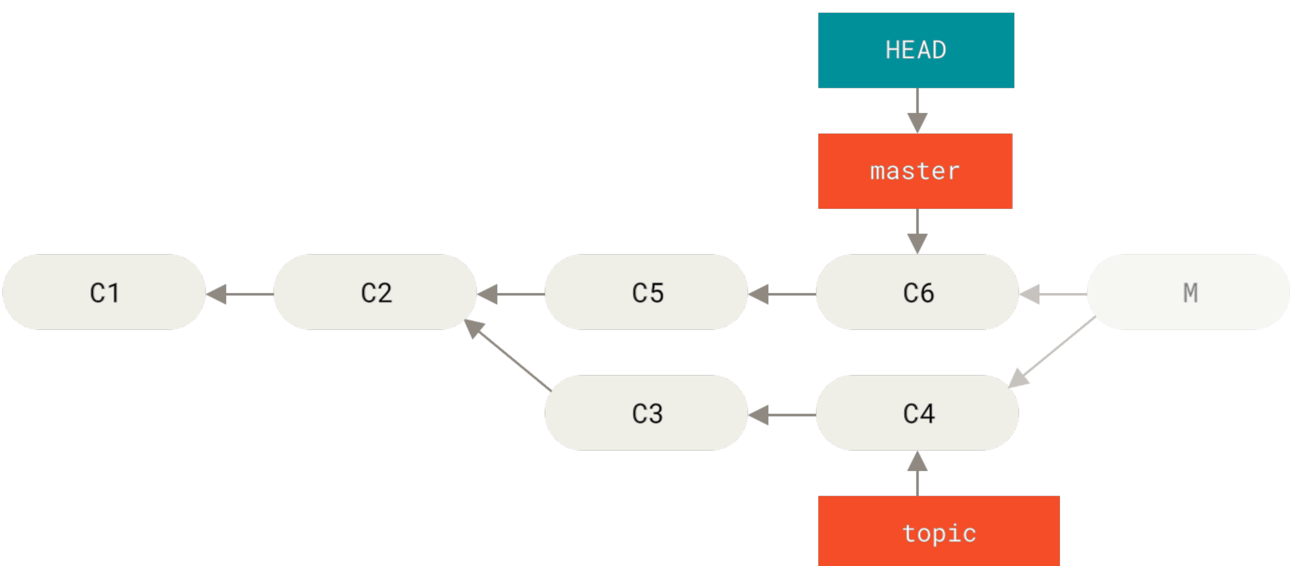


Figure 138. Историята след изпълнение на `git reset --hard HEAD~`

Разглеждахме вече `reset` в Мистерията на командата Reset, така че не би следвало да ви е трудно да разберете какво се случва тук. Като бързо припомняне: `reset --hard` обикновено работи на три стъпки:

1. Премества клона, към който сочи HEAD. В този случай искаме да местим `master` до позицията, в която е бил преди сливащия къмит (`C6`).

2. Променя индекса да изглежда като HEAD.
3. Променя работната директория да изглежда като индекса.

Недостатъкът на този подход е, че променя историята, което може да е проблематично при споделени хранилища. Погледнете Опасности при пребазиране за повече информация какво би могло да се случи. Накратко, би следвало да избягвате `reset` ако други хора вече имат къмитите, които ще преработите. Освен това, този подход няма да работи ако след сливането има направени други къмити. Преместването на указателите ще доведе до загуба на промените от тези къмити.

Връщане на къмит

Ако преместването на указателите на клоновете не работи за вас, Git ви дава възможността да направите нов къмит, който отменя промените на съществуващ такъв. Git нарича тази операция “revert” и в този специфичен сценарий се прави така:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

Флагът `-m 1` указва кой родител да се счита за “mainline” и да бъде запазен. Когато направите сливането в `HEAD` (`git merge topic`), новият къмит има два родителя: първият е `HEAD` (`C6`) и вторият е върха на клона, който бива сливан (`C4`). В този случай, ние искаме да отменим всички промени настъпили в резултат от сливането на родител #2 (`C4`) и същевременно да запазим съдържанието на родител # 1 (`C6`).

Историята с revert къмита сега изглежда така:

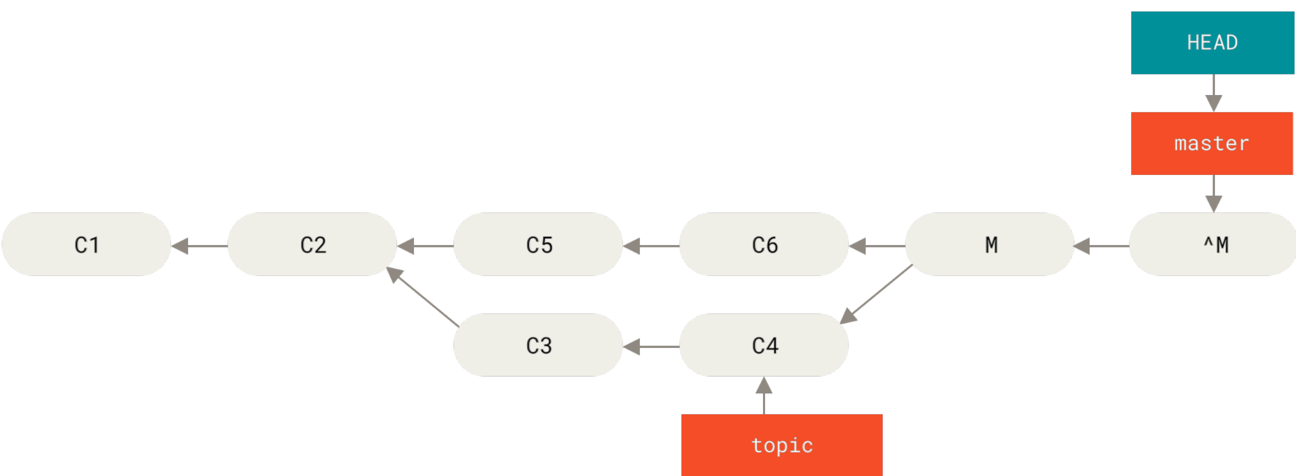


Figure 139. Историята след `git revert -m 1`

Новият къмит `^M` има същото съдържание като `C6`, така че оттук нататък нещата са такива сякаш сливането въобще не се е случвало с изключение на факта, че сега не-слетите къмити още са в историята на `HEAD`. Git ще изпадне в затруднение, ако сега отново се опитате да слеете `topic` в `master`:

```
$ git merge topic
Already up-to-date.
```

В `topic` сега няма нищо, което да не е вече достъпно през `master`. Което е по-лошо, ако сега направите промени в `topic` и слеете отново, Git ще вземе промените направени *след* reverted сливането.

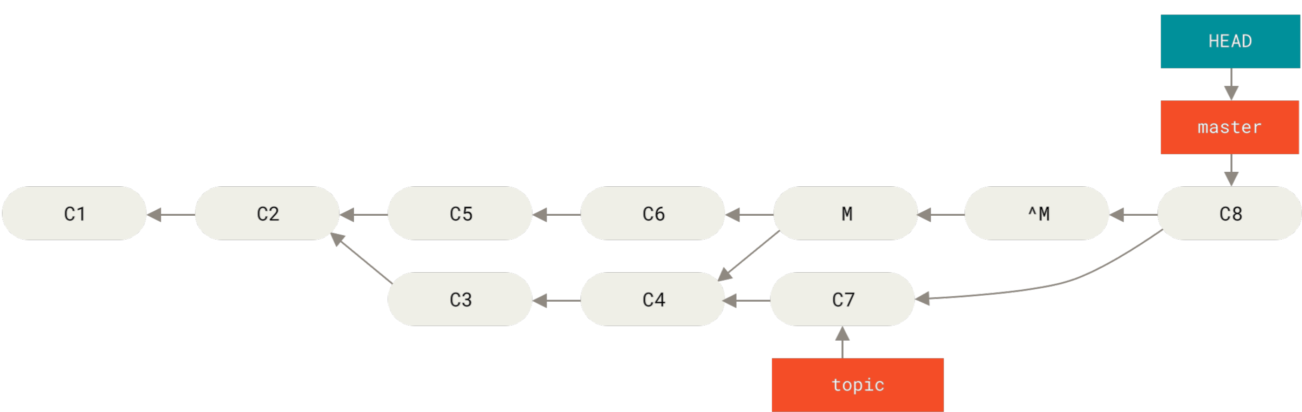


Figure 140. Историята след лошо сливане

Най-добрият начин да заобиколите това е да откажете revert-а на оригиналното сливане, понеже сега искате да върнете отменените промени и **след това** да създадете нов сливащ къмит:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

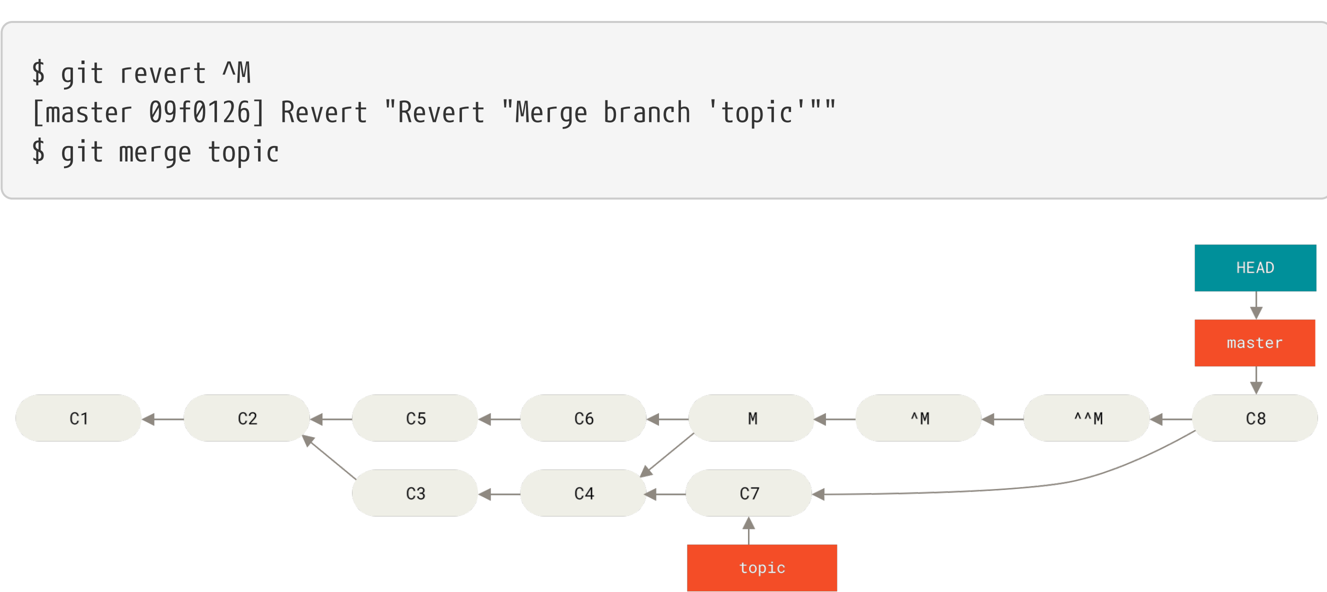


Figure 141. История след повторно сливане на reverted merge

В този пример, `M` и `^M` са отказани. `^^M` ефективно слива промените от `C3` и`C4`, а `C8` слива промените от `C7`, така че сега `topic` е напълно слят.

Други типове сливания

Досега прегледахме нормално сливане между два клона, което обикновено се осъществява с т. нар. “recursive” сливаща стратегия. Обаче, има и други начини за сливане на клонове. Нека видим някои от тях набързо.

Our или Theirs преференция

Преди всичко има още едно полезно нещо, което можем да правим с нормалния "recursive" режим на сливане. Вече видяхме `ignore-all-space` и `ignore-space-change` опциите подавани с `-X`, но можем също така да кажем на Git да дава предимство на едната или другата страна при сливането, когато установи конфликт.

По подразбиране, когато Git види конфликт между два сливащи се клона, ще добави merge conflict маркери в кода и ще маркира файла като конфликтен, очаквайки да го коригирате. Ако предпочитате Git просто да избере една от двете опции и да игнорира другата, вместо да остави на вас решаването на проблема, можете да подадете на `merge` параметрите `-Xours` или `-Xtheirs`.

В такъв случай маркери няма да бъдат добавяни. Всички промени, които могат да бъдат слети чисто, ще бъдат слети. При всички промени, които предизвикват конфликт, Git просто ще избере указаната от вас страна, включително за двоичните файлове.

Ако се върнем обратно до "hello world" примера, ще видим че сливането в нашия клон предизвиква конфликти.

```
$ 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.
```

Ако обаче изпълним командата с `-Xours` или `-Xtheirs`, конфликти няма.

```
$ 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
```

В този случай, вместо да слага маркери за конфликт с "hello mundo" от една страна и "hola world" от друга, Git просто ще избере "hola world". Но всички останали промени от този клон, които не предизвикват конфликт, ще бъдат успешно слети.

Тази опция може да се подаде и на командата `git merge-file`, която видяхме по-рано изпълнявайки нещо като `git merge-file --ours` за индивидуални сливания на файлове.

Ако искате да направите нещо такова, но да укажете на Git дори да не се опитва да слива промени от другата страна, има още по-рестриктивна опция, известна като "ours" merge *стратегия*. Това е различно от "ours" recursive merge *опцията*.

В действителност това ще направи лъжливо сливане. Процесът ще запише нов сливащ

къмит с двата клона като родители, но практически дори и няма да погледне в клона, който сливате. Резултатът ще е сливане, което просто записва съдържанието на текущия клон.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Може да видите, че няма разлика между клона, в който сме били и резултата от сливането.

Често подобен подход може да е полезен, за да прилъжете Git да приема, че клонът е вече слят, когато искате да правите сливане по-късно. Да кажем, че сте създали клон `release` и сте свършили някаква работа в него, която по-късно ще искате да слеете в `master`. Междувременно се оказва, че в `master` е направена някаква спешна корекция на грешка и тази промяна трябва да се интегрира в `release`.

Можете да слеете bugfix клона в `release` и също да направите `merge -s ours` за същия клон в `master` клона (въпреки, че поправката е вече вътре). Така, когато по-късно слеете `release` клона отново, няма да има конфликти от поправката на грешката.

Subtree сливане

Идеята на subtree сливането е че имате два проекта и единия от тях съществува в поддиректория на другия. Когато укажете subtree сливане, Git често е достатъчно добър да установи, че единия е поддърво на другия и слива съответно.

Ще видим пример за добавяне на отделен проект в съществуващ такъв и след това за сливане на код от втория в поддиректория на първия.

Първо, ще добавим приложението Rack към нашия проект. Ще добавим Rack проекта като отдалечена референция в нашия собствен проект и след това ще го извлечем в негов собствен клон:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
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"
```

Сега имаме корена на Rack проекта в нашия клон `rack_branch` и нашия собствен проект в `master` клона. Ако превключите единия и после другия, може да видите, че те имат различни корени:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Това е леко странна концепция. Не е задължително всички клонове във вашето хранилище да са клонове от един и същи проект. Не е често срещана ситуация, понеже рядко е полезна, но е сравнително лесно да имате клонове съдържащи изцяло различни истории.

В този случай, ние искаме да издърпаме Rack проекта в нашия `master` проект като поддиректория. Можем да направим това с `git read-tree`. Ще научим повече за тази команда и придружаващите я други в Git на ниско ниво, засега просто приемете, че тя прочита главното дърво на един клон в текущия индекс и работна директория. Току що превключихме към `master` клона и издърпваме `rack_branch` клона в поддиректорията `rack` на нашия `master` клон за основния ни проект:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Когато къмитнем, изглежда имаме всички Rack файлове в тази поддиректория — също както ако бяхме копирали директно вътре от архив. Интересното в случая е, че можем сравнително лесно да сливаме промени от единия клон в другия. Така, ако Rack проектът бъде обновен, можем да издърпаме upstream промените като превключим към този клон и

стартираме издърпването:

```
$ git checkout rack_branch
$ git pull
```

След това, можем да слеем новите промени обратно в нашия master клон. За да изтеглим промените и да попълним предварително къмит съобщението, използваме --squash опцията, както и -Xsubtree параметъра на recursive merge стратегията. Рекурсивната стратегия се подразбира тук, но я указваме за яснота.

```
$ 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
```

Всички промени на Rack проекта са слети и са готови да се къмитнат локално. Можете да направите и обратното — да направите модификации в rack поддиректорията на master клона и след това да го слеете в rack_branch клона по-късно за да ги изпратите на собствениците на проекта.

Това ни дава възможност да следваме работен процес донякъде подобен на submodule без да използваме подмодули (което ще разгледаме в Подмодули). В подобен маниер можем да пазим клонове с други свързани проекти в нашето хранилище и да ги интегрираме при необходимост в нашия собствен проект посредством subtree сливане. В някои аспекти това е хубаво, защото целият код се къмитва на едно място. От друга страна, един такъв подход си има и недостатъци, защото е малко по-сложен, по-лесно е да се правят грешки при повторна интеграция на промени и също така може по невнимание да се публикува погрешен клон в неподходящо хранилище.

Друго неудобство е, че за да получите diff между съдържанията на rack поддиректорията и кода в rack_branch клона (така че да разберете дали се налага да ги сливате) — не можете да използвате нормалната diff команда. Вместо това, трябва да използвате git diff-tree с клона, който искате да сравнявате:

```
$ git diff-tree -p rack_branch
```

Или, за да сравните съдържанието на rack поддиректорията със съдържанието на master клона на сървъра последния път когато сте го издърпали, може да изпълните:

```
$ git diff-tree -p rack_remote/master
```

Rerere

Функционалността на git rerere е един вид скрита опция. Името идва от фразата “reuse

recorded resolution" и както името подсказва, позволява ви се да укажете на Git да запомня как сте разрешили даден конфликт така че следващия път, когато той възникне отново — да бъде автоматично разрешен.

Има няколко сценария, когато това може да ви е от помощ. Един от примерите е упоменат в документацията и описва ситуация, в която искате да сте сигурни, че продължително съществуващ topic клон ще се слива чисто винаги, но не желаете да имате множество междинни сливащи къмити задръстващи историята ви. С разрешен `rerere`, можете да опитате случайно сливане, да разрешите конфликтите и след това да откажете сливането. Ако правите това продължително, тогава финалното сливане би трябвало да е лесно, защото `rerere` ще свърши корекциите вместо вас.

Същата тактика може да се използва, ако искате да пазите даден клон пребазиран и не желаете да се занимавате с едни и същи конфликти при пребазиране всеки път. Или, ако искате да вземете клон, който сте слели и в който сте разрешили много конфликти и след това пожелаете да го пребазирате — най-вероятно не искате да виждате всичките конфликти отново.

Друго приложение на `rerere` е когато случайно сливате множество развиващи се клонове в едно за тестване както Git проекта прави например. Ако тестовете не минават успешно, можете да превъртите назад сливанията и да ги повторите без участието на topic клона, който ги проваля без да трябва да решавате конфликтите отново.

За да активирате `rerere` функционалността, просто изпълнете:

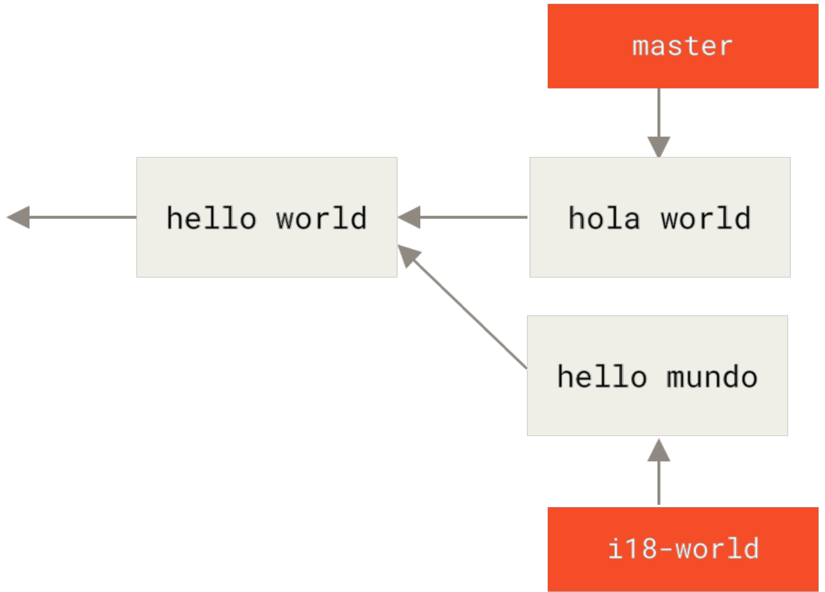
```
$ git config --global rerere.enabled true
```

Можете да я разрешите и за конкретно хранилище създавайки директорията `.git/rr-cache`, но конфигурационната опция е по-чист начин и позволява глобална настройка.

Нека видим просто пример подобен на предишните. Имаме файл `hello.rb` със съдържание:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

В един от клоновете ни сменяме думата "hello" на "hola", след това в друг клон променяме "world" на "mundo", точно както преди.



Когато сливаме двата клона в едно, получаваме конфликт по съдържание:

```
$ 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.
```

Забелязваме новия ред от изхода, `Recorded preimage for FILE`. Освен него, всичко си изглежда като при нормален конфликт. На този етап `rerere` може да ни каже няколко неща. Нормално, можете да пуснете `git status` за да видите какъв е конфликта:

```
$ 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
#
```

Обаче, командата `git rerere` в допълнение ще ви уведоми, че е запомнила статуса преди сливането:

```
$ git rerere status
hello.rb
```

А `git rerere diff` ще ви покаже текущия статус на корекцията на конфликта — с какво сте започнали корекцията и как сте я завършили.

```
$ 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
```

Също така (и това няма връзка с `rerere`), можете да използвате `git ls-files -u` за да видите конфликтните файлове и техните версии — оригинална, лява и дясна:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1   hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2   hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3   hello.rb
```

Сега можете да разрешите конфликта, така че редът да е `puts 'hola mundo'` и да пуснете `git rerere diff` отново, за да видите какво ще бъде запомнено от rerere:

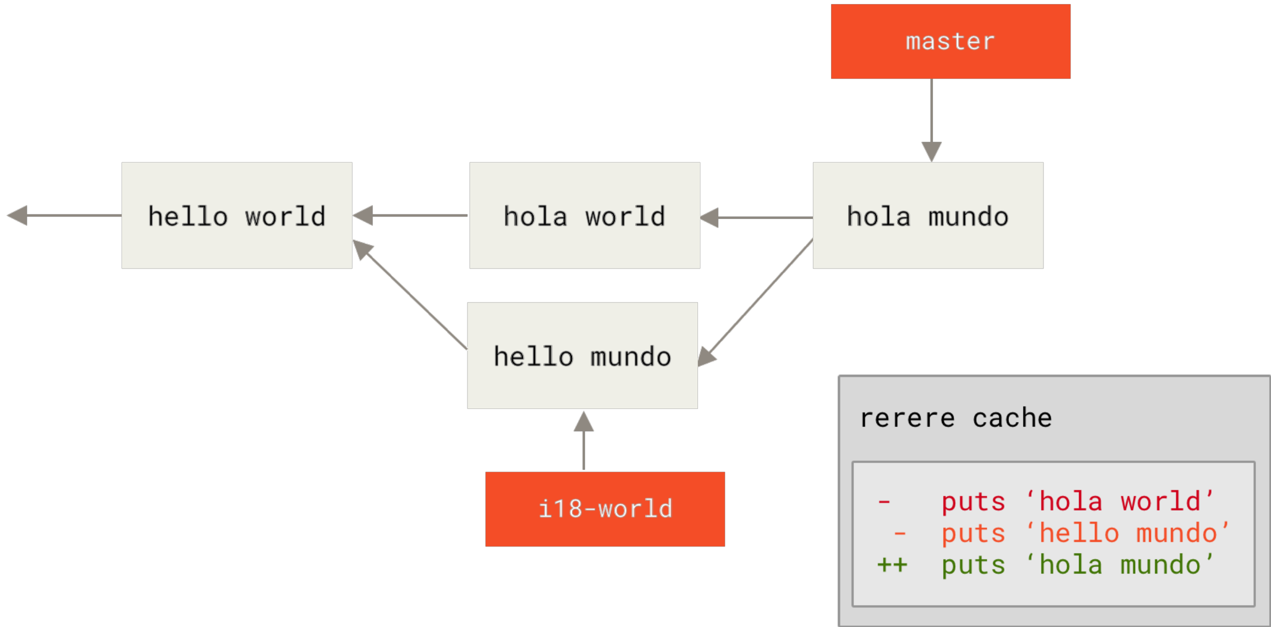
```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

Казано с други думи това означава, че когато Git намери конфликт във файла `hello.rb`, при който има “hello mundo” от едната страна и “hola world” на другата, той ще го коригира автоматично използвайки “hola mundo”.

Сега можем да маркираме конфликта като разрешен и да къмитнем:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Виждаме съобщението "Recorded resolution for FILE".



Сега, нека да отменим това сливане и да го пребазираме на върха на `master` клона. Можем да преместим клона назад с `git reset` както видяхме в Мистерията на командата Reset.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Сега сливането ни е отменено. Следва да пребазираме topic клона.

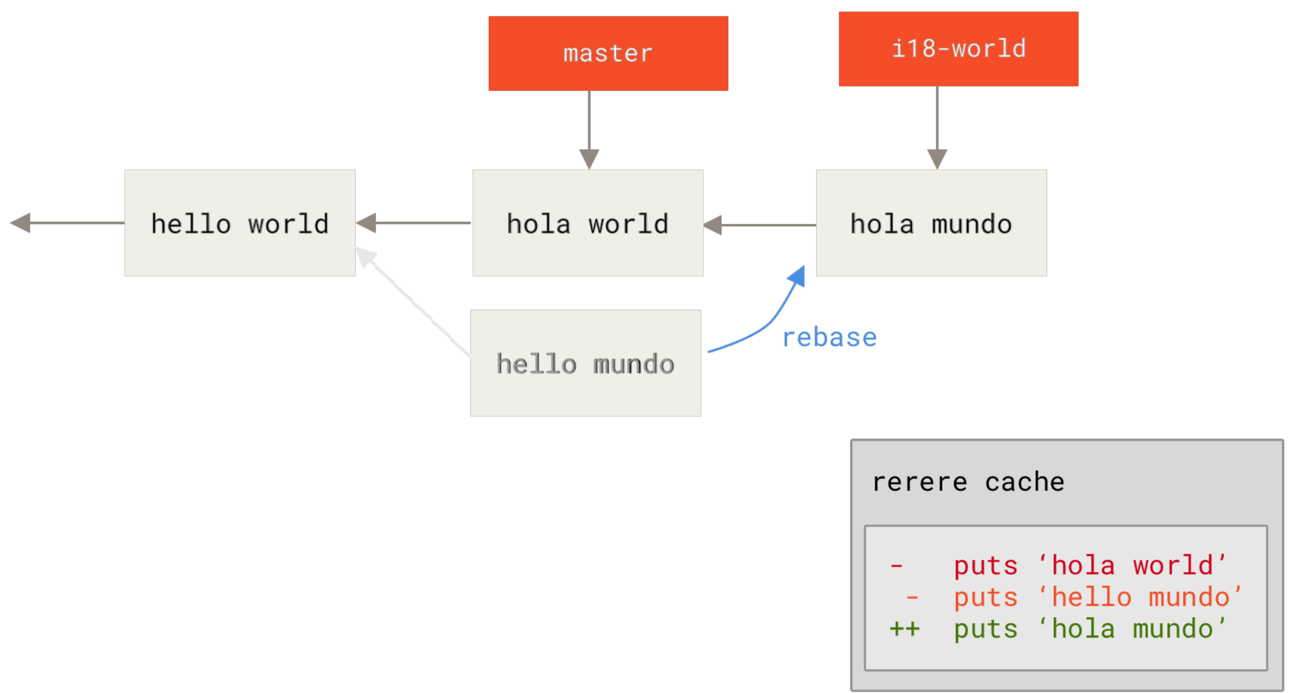
```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ 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
```

Сега получихме същия конфликт, който очакваме, но обърнете внимание на реда `Resolved FILE using previous resolution` в отпечатания изход. Ако погледнем файла ще видим, че той вече е коригиран и не съдържа маркери за конфликт.

```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Също, `git diff` ще ни покаже как е направена автоматичната корекция:

```
$ 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
```



Можете също да пресъздадете конфликтния статус на файла с `git 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
```

Видяхме пример за това в Сливане за напреднали. Засега обаче, нека да го коригираме отново като просто изпълним `git rerere` повторно:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Сега сме повторили корекцията автоматично използвайки кешираната от `rerere` информация за нея. Сега можете да добавите файла и да продължите пребазирането, за да го завършите.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

И така, ако правите много повтарящи се сливания или пък искате да държите topic клон в синхрон с промените на `master` клона без много излишни сливания, или пък ако често пребазирате — можете да включите `rerere`, за да си улесните работата.

Дебъгване с Git

В допълнение на основната си задача за контрол на версиите, Git също така осигурява някои команди, които могат да ви помогнат при търсене на грешки в сорс кода. Понеже Git е проектиран да обслужва всякакъв вид съдържание, тези инструменти са доста общи, не навлизат в дълбочина, но въпреки това често могат да бъдат полезни.

Анотации на файлове

Ако проследявате дадена грешка в кода си и искате да разберете кога и как е възникнала, файловите анотации често са най-добрия начин да направите това. Те показват кой е последния къмит модифицирал всеки един ред от произволен файл. Така че, ако срещнете метод с бъгав код в него, можете да анотирате файла с командата `git blame` за да установите кой къмит е въвел в кода специфичния ред или редове от код.

Следващият пример използва `git blame` за да установи кои къмити и разработчици са отговорни за редовете код от един `Makefile` файл в проекта на Linux ядрото и освен това използва флага `L` за да ограничи изхода до редовете между 69 и 82:

```
$ git blame -L 69,82 Makefile
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 69) ifeq ("$(origin V)",
"command line")
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 70)   KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 72) ifndef KBUILD_VERBOSE
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 73)   KBUILD_VERBOSE = 0
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 74) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75)
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 76) ifeq ($(KBUILD_VERBOSE),1)
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 77)   quiet =
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 78)   Q =
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 79) else
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 80)   quiet=quiet_
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 81)   Q = @
066b7ed955808 (Michal Marek   2014-07-04 14:29:30 +0200 82) endif
```

Първото поле в ляво на таблицата е част от SHA-1 хеша на къмита, който последно е модифицирал съответния ред. Следващите две колони се извличат от самия къмит — името на автора му и датата на къмита. В следващите колони идват номера и съдържанието на

реда от файла. Отбележете също и редовете започващи с `^1da177e4c3f4`, при тях префикса `^` обозначава, че те са били създадени с първия къмит на хранилището и след това не са променяни нито веднъж. Този вид обозначение причинява малко смут, защото досега видяхме поне три различни начина, при които Git използва този символ за да модифицира SHA-1 хеш стойностите, но в този конкретен случай това е неговото значение.

Друга хубава черта на Git е, че не проследява изрично преименуванията на файловете. Git записва snapshot-ите и след това се опитва да установи какво е било безусловно преименувано постфактум. Една от интересните особености на това е, че можете да правите запитвания и за премествания на код. С параметър `-C` към `git blame` Git анализира файла, който анотирате и се опитва да установи откъде са се появили дадени отрязъци от код, ако те са копирани от друго място. Например, да кажем че преработвате файл с име `GITServerHandler.m` в множество файлове, един от които се казва `GITPackUpload.m`. Изследвайки `GITPackUpload.m` с параметър `-C`, можете да видите къде първоначално са се намирали секции код от него:

```
$ 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)
```

Това е наистина полезно. Нормално бихте получили като оригинален къмит този, при който сте вмъкнали съдържанието в новия файл, защото това е първия път, когато сте докоснали тези редове в този файл. Git обаче може да ви намери и оригиналния къмит, в който сте написали тези редове, дори те да са били в друг файл.

Бинарно търсене

Анотирането на файлове помага, когато знаете къде започва проблема. Ако обаче не знаете какво се е счупило и има дузина или стотици къмити от последния път, когато знаете че кодът е работил, вероятно ще прибегнете до помощта на `git bisect`. Командата `bisect` се използва за бинарно търсене в историята на къмитите за да ви помогне да идентифицирате даден проблем възможно най-бързо.

Нека кажем, че сте публикували завършена версия на кода си и започвате да получавате новини за грешки, които не се наблюдават в работната версия и нямате идея защо кодът се държи така. Връщате се обратно в кода и се оказва, че всъщност можете да пресъздадете грешката, но все още не разбирате защо възниква. В такъв случай можете да *bisect-нете*

(разполовите) кода за да потърсите причината (понятието идва от начина, по който Git третира и разцепва историята в процеса на търсене описан в примерите, които следват). Първо, изпълнявате `git bisect start` за да стартирате процеса и след това `git bisect bad` за да кажете на системата, че текущия къмит е проблематичен. След това, трябва да кажете на bisect кога е бил последния работещ статус на нещата изпълнявайки `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 е установил около 12 къмита, които са се случили между този, който сте маркирали като последно работещ (v1.0) и текущата проблемна версия и е извлякъл за вас средния от тях. На този етап, можете да си пуснете тестовете, за да видите дали проблемът все още се появява в този извлечен къмит. Ако е така, то той е възникнал някъде преди средния къмит. Ако не е, значи възниква след него. Да кажем, че проблемът го няма на текущия къмит — вие уведомявате Git за това с `git bisec good` и продължавате да търсите:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Secure this thing
```

Сега сте на друг къмит, който е по средата между този който току що тествахте и проблематичния. Пускате тестовете отново и този път установявате, че грешката се появява и уведомявате Git с `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Drop exceptions table
```

Този къмит е безпроблемен и сега Git има информацията, която му е нужна за да разбере къде се е появил проблема. Вие получавате SHA-1 хеша на първия проблемен къмит, заедно с малко данни за него и списък на модифицираните файлове така че да успеете да установите източника на грешката:

```
$ 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
```

Когато приключите, трябва да изпълните `git bisect reset` за да пренасочите HEAD към мястото, където сте били преди да стартирате процеса:

```
$ git bisect reset
```

Това е мощен инструмент, който за минути може да провери стотици къмити в процеса на търсене на грешка. В допълнение, ако имате скрипт, който завършва със статус exit 0, ако проектът работи коректно или с различен статус, ако това не е така, можете напълно да автоматизирате `git bisect`. Първо, вие отново указвате обхвата на bisect подавайки познатите добър и лош къмит. Можете да направите това изписвайки ги с `bisect start` командата, първо подавате лошия (HEAD) и след това последно познатия добър (v1.0):

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Правейки това, Git автоматично ще изпълнява `test-error.sh` за всеки извлечен къмит докато намери първия проблематичен. Можете също да изпълнявате неща като `make` или `make tests` или каквото и да било друго средство, което използвате за автоматично тестване.

Подмодули

Често се случва докато работите по един проект да трябва да използвате друг такъв в него. Може да е библиотека разработена от други или такава, която сте написали вие, но я използвате в множество проекти. В такива случаи възниква тривиален проблем: искате да третирате двата проекта като отделни, но и да можете да използвате единия в другия.

Ето пример. Разработвате уебсайт и създавате Atom потоци. Вместо да пишете собствен Atom генериращ код, решавате да използвате библиотека. Може да вмъквате кода от споделени библиотеки като CPAN install или Ruby gem или да копирате сорса в дървото на собствения си проект. Проблемът с включването на библиотеката е, че е трудно да я настроите според вашите нужди и често е още по-трудно да я внедрите в production среда, защото трябва да сте сигурни, че всеки клиент я има при себе си. Проблемът с копирането във вашия проект пък е, че всички ваши промени по библиотеката ще станат трудни за сливане когато upstream източника се промени.

Git се занимава с тези проблеми посредством подмодули. Подмодулите ви позволяват да пазите Git хранилище като поддиректория в друго Git хранилище. Това от своя страна ви позволява да клонирате друго хранилище във вашия проект и да пазите къмитите им изолирани.

Първи стъпки с подмодулите

Ще преминем през разработката на прост проект, който ще разделим на основен проект и няколко подпроекта.

Нека започнем добавяйки налично Git хранилище като подмодул на това, в което работим. Необходимата ви команда е `git submodule add` с абсолютния или релативен URL на проекта, който искаме да започнем да следим. В нашия пример ще включим библиотека наречена "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.
```

По подразбиране, подмодулите се добавят в директории съответстващи на имената на хранилищата им, в този случай "DbConnector". Ако искате да са в различен път, добавяте го като параметър в края на командата.

Ако сега изпълним `git status` ще видим няколко неща.

```
$ 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
```

На първо място е новият файл `.gitmodules`. Това е конфигурационен файл, който съхранява съответствието между URL-а на проекта и локалната директория, в която сме го изтеглили:

```
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

При повече подмодули, ще имате и повече секции в този файл. Важно е да посочим, че този

файл попада под контрол на версиите както и другите ви файлове като `.gitignore`. Това значи, че ще се тегли и публикува с останалата част от проекта. Това се прави, за да може други хора, които клонират проекта да ви да знаят откъде да вземат проектите на подмодулите.

Понеже URL-ите в .gitmodules файла са мястото, от което другите хора първо ще опитат да клонират/теглят, уверете се доколкото е възможно, че това са адреси до които те имат достъп. Например, ако използвате различен URL за публикуване от този който другите ще ползват за теглене, постарайте се да ползвате този, до който другите имат достъп. Можете да коригирате тази стойност локално с `git config submodule.DbConnector.url PRIVATE_URL` за ваше собствено ползване. Когато е възможно, релативните адреси могат да са полезни.

Друга интересна информация от изхода на `git status` е елемента с директорията на проекта. Ако изпълните `git diff` ще видите нещо любопитно:

```
$ 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
```

Въпреки че `DbConnector` е поддиректория в работната ви директория, Git я вижда като подмодул и не следи съдържанието ѝ когато сте извън нея. Вместо това, Git я третира като отделен къмит.

Ако искате малко по-приятен diff изход, може да подадете `--submodule` параметъра към `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)
```

Когато къмитнете, виждате нещо такова:

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Забележете режима `160000` за елемента `DbConnector`. Това е специфичен режим в Git, който в общи линии показва, че записвате къмит като директория, вместо като поддиректория или файл.

Последно, публикуваме промените:

```
$ git push origin master
```

Клониране на проект с подмодули

Тук ще клонираме проект с подмодули в него. Когато клонирате подобно хранилище, по подразбиране получавате директориите на подмодулите им, но без съдържание в тях:

```
$ 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
$
```

Директорията `DbConnector` е тук, но е празна. Трябва да изпълните две команди: `git submodule init` за да инициализирате вашия локален конфигурационен файл и след това `git submodule update` за да издърпате данните за подмодулите и да попълните директориите им със съдържанието на последните им къмити:

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ 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 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Сега `DbConnector` поддиректорията е в същия статус, в който я къмитнахте по-рано.

Има и по-лесен начин да направите това. Ако подадете параметър `--recurse-submodules` на `git clone`, целият този процес ще бъде свършен от автоматично - Git ще инициализира и обнови всеки подмодул в хранилището, включително вложените такива, ако съществуват в хранилището.

```
$ git clone --recurse-submodules 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 path
'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.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Ако вече сте клонирали проект, но сте забравили `--recurse-submodules`, може да комбинирате стъпките `git submodule init` и `git submodule update` изпълнявайки `git submodule update --init`. За да инициализирате, изтеглите и извлечете в работната директория всички вложени подмодули, може да използвате и `git submodule update --init --recursive`.

Работа по проект с подмодули

Сега имаме копие от проекта с подмодули в него и ще работим заедно с колегите си както по основния проект, така и по подмодула.

Издърпване на Upstream промени от Submodule Remote

Най-простият модел на работа с подмодули е просто да ги ползвате в проекта си без да ги модифицирате и от време на време да проверявате за промени по тях. Да видим малък пример.

Ако искате да проверите за нова работа в даден подмодул, можете да влезете в директорията му и да изпълните `git fetch` и `git merge` към upstream клона.

```
$ 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(+)
```

Ако сега се върнете в основния проект и изпълните `git diff --submodule`, може да видите че подмодула е обновен и да получите списък на къмитите добавени в него. Ако не искате да пишете `--submodule` при всяко изпълнение на `git diff`, може да зададете опцията като подразбираща се с конфигурационната настройка `diff.submodule`, на която трябва да дадете стойност "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Ако къмитнете на този етап, ще заключите подмодула да получава нов код, когато други хора ъпдейтват.

Има и по-лесен начин да направите това, ако не ви се иска ръчно да дърпате и сливате в поддиректориите. Ако изпълните `git submodule update --remote`, Git ще премине през подмодулите, ще ги изтегли и ще ги обнови вместо 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 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Командата по подразбиране приема, че желаете да обновите съдържанието на `master` клона в хранилището на подмодула. Но можете да промените това поведение, ако желаете. Например, ако искате DbConnector подмодула да следи клона “stable” от хранилището си, можете да зададете това или в `.gitmodules` файла (така че и всички останали да следят този клон) или само в локалния ви файл `.git/config`. Нека използваме `.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 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Ако пропуснете частта `-f .gitmodules` командата ще направи промяната само за вас, но вероятно е по-смислено да следите тази информация с хранилището, така че всички останали да правят същото.

Когато пуснем `git status` сега, Git ще ни покаже, че имаме “нови къмити” в подмодула.

```
$ 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")
```

Ако активирате конфигурационната настройка `status.submodulesummary`, Git също така ще ви показва кратък списък на промените в подмодулите ви:

```
$ 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
```

Изпълнението на `git diff` сега ще ни покаже, че сме модифицирали `.gitmodules` файла ни и че имаме няколко къмита, които сме изтеглили и са готови да бъдат къмитнати в submodule проекта.

```
$ 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
```

Това е добре, защото можем в действителност да видим информацията от къмитите, която ще къмитнем в нашия подмодул. Веднъж къмитната, тази информация е видима и по-късно, когато изпълним `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
```

По подразбиране Git ще се опитва да обновява **всички** подмодули при изпълнение на `git submodule update --remote`. Ако те са повечко, можете да подадете само името на този, който желаете да обновите.

Издърпване на Upstream промени от Project Remote

Нека сега се поставим в ролята на вашия сътрудник, който има собствено локално копие на MainProject хранилището. Само изпълнението на `git pull` за издърпване на вашите промени не е достатъчно:

```
$ git pull
From https://github.com/chaconinc/MainProject
   fb9093c..0a24cfc  master     -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
   c3f01dc..c87d55d  stable     -> origin/stable
Updating fb9093c..0a24cfc
Fast-forward
 .gitmodules         | 2 +-
 DbConnector         | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)

$ 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:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
  < catch non-null terminated lines
  < more robust error handling
  < more efficient db routine
  < better connection routine

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

По подразбиране, `git pull` рекурсивно изтегля промените по подмодулите, както може да се види от изхода на първата команда отгоре. Обаче, тя не **обновява** подмодулите. Това е видно от изхода на `git status`, която показва, че подмодулът е "modified", и че има "new commits". В допълнение, скобите показващи новите къмити сочат наляво (<), което значи, че тези нови къмити са записани в MainProject, но не са налични в локалното DbConnector работно копие. За да финализирате обновлението, ще трябва да изпълните `git submodule update`:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'

$ git status
 On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Отбележете, че за да сте напълно сигурни, би трябвало да изпълните `git submodule update` с флага `--init` (в случай че MainProject къмитите, които току що сте издърпали, добавят нови подмодули), и с `--recursive` флага, ако някой от подмодулите съдържат вложени подмодули.

Ако искате да автоматизирате този процес, може да добавите флага `--recurse-submodules` към `git pull` (от версия на Git 2.14). Това ще накара Git д аизпълни `git submodule update` веднага след изтеглянето, което да постави подмодулите в коректен режим. Освен това, ако желаете Git винаги да тегли с `--recurse-submodules`, може да укажете това с конфигурационната опция `submodule.recurse` (това ще важи от версия 2.15 и нагоре). Така Git ще използва флага `--recurse-submodules` за всички команди, които го поддържат (с изключение на `clone`).

Съществува специална ситуация, която може да възникне при издърпване на промени от подпроект: може upstream хранилището да е променило адреса на подмодула в `.gitmodules` файла в някой от къмитите, които теглите. Това може да стане, ако например проектът на подмодула е със сменена хостинг платформа. В такъв случай е възможно командите `git pull --recurse-submodules`, или `git submodule update` да не успеят, ако главният проект сочи към къмит на подмодул, който не се намира в локално конфигурирания в хранилището submodule remote. За да се избегне тази ситуация е необходима командата `git submodule sync`:

```
# copy the new URL to your local config
$ git submodule sync --recursive
# update the submodule from the new URL
$ git submodule update --init --recursive
```

Работа по подмодул

Ако ползвате подмодули, твърде вероятно е да искате да работите и по техния код едновременно с основния проект. В противен случай може да предпочетете да ползвате по-проста dependency management система като Maven или Rubygems.

Да разгледаме пример с промяна на кода на подмодул и на основния проект, къмитване и публикуване на промените.

Досега, когато изпълнявахме `git submodule update` командата за да изтеглим промените от хранилищата на подмодулите, Git ще вземa новия код и ще обновява файловете в поддиректориите, но ще ги остави в режим известен като “detached HEAD”. Това означава, че нямаме работещ локален клон (като `master` например) следящ промените. Липсата на такъв клон означава, че дори да къмитнете промени по подмодула, тези промени е твърде вероятно да се загубят при следващото изпълнение на `git submodule update`. Необходими са малко допълнителни стъпки, ако искате промените в подмодул да бъдат следени.

Две неща трябва да се направят. Трябва да отидете в директорията на всеки подмодул и да стартирате клон, по който да работите. След това трябва да инструктирате Git как да действа, ако сте направили промени и след това `git submodule update --remote` внесе нови данни от upstream хранилището. Опциите са да можете да ги слеете с локалната ви работа или да се опитате да пребазирате локалната работа върху новите промени.

Първо, да влезем в директория на подмодул и да създадем клон.

```
$ cd DbConnector/
$ git checkout stable
Switched to branch 'stable'
```

Нека опитаме с опцията “merge”. За да я подадем ръчно, добавяме `--merge` към `update` повикването. Тук ще видим, че на сървъра е имало промяна в кода на този подмодул и тя се слива.

```
$ cd ..
$ 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'
```

Ако влезем в DbConnector директорията, ще имаме новите данни слети в локалния `stable` клон. Сега нека видим какво се случва, ако направим наша собствена промяна в библиотеката и някой друг публикува друга upstream промяна по същото време.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
 1 file changed, 1 insertion(+)
```

Пускаме обновяването на подмодула при едновременно налични локална и отдалечена промяна, която трябва да внедрим.

```
$ cd ..
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: Unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Ако забравите `--rebase` или `--merge`, Git просто ще обнови подмодула до съдържанието на сървъра и ще върне проекта в режим detached HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Ако това се случи, не е голям проблем, можете просто да се върнете в директорията и да извлечете клона отново (той все още ще съдържа вашата работа), след което да слеете или пребазирате `origin/stable` (или който и да е отдалечен клон) ръчно.

Ако не сте къмитнали промените си в подмодула и пуснете обновяване, което би могло да предизвика проблеми, Git ще издърпа промените, но няма да презапише несъхранената работа в поддиректорията.

```
$ 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'
```

Също, ако сте направили промени, които водят до конфликт с нещо променено в upstream-а, Git ще ви уведоми за това.

```
$ 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'
```

Можете да влезете в поддиректорията и да оправите конфликта по нормалния начин.

Публикуване на промени по подмодул

Сега имаме промени в директорията на подмодула. Някои от тях идват от upstream-а след обновяване а други са направени ръчно локално и не са достъпни за никой друг, защото все още не сме ги публикували.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > Update setup script
  > Unicode support
  > Remove unnecessary method
  > Add new option for conn pooling
```

Ако къмитваме в главния проект и го публикуваме без да включим промените в подмодула, то другите хора, които ползват кода ни ще бъдат в затруднение, защото няма да имат начин да се се сдобият с тези промени, а коректната работа на основния проект може да зависи от тях. Тези промени ще са само в локалното ви копие.

За да сте сигурни, че това няма да се случва, можете да укажете на Git да проверява дали всички подмодули са публикувани успешно преди да публикува основния проект. Командата `git push` приема аргумента `--recurse-submodules`, който може да се зададе със стойности "check" или "on-demand". Стойността "check" ще направи така, че публикуването да бъде отказано, ако произволна къмитната промяна в подмодул не е била публикувана успешно.

```
$ 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.
```

Както се вижда, получавате и напътствия как бихте могли да продължите. Най-простата възможност е да преминете през директориите на всички подмодули и ръчно да публикувате съдържанието им, след което да се върнете и да пробвате да публикувате главния проект. Ако искате това поведение с проверка да е валидно за всички публикувания, можете да изпълните `git config push.recurseSubmodules check`

Другата опция е да използвате стойността "on-demand", което ще опита да свърши работата 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
```

Както се вижда тук, Git е отишъл в DbConnector модула и е публикувал него преди да опита Push за главния проект. Ако по някаква причина този подмодул не успее да се публикува, главния проект също няма да успее. За да направите това поведение подразбиращо се, изпълнете `git config push.recurseSubmodules on-demand`.

Сливане на промени в подмодул

Ако промените референция на подмодул по същото време като някой друг, може да възникнат проблеми. Това е ситуация, при която историите на подмодул са се разделили и са къмитнати в разделящи се клонове в суперпроект.

Ако единият от къмитите е директен предшественик на другия (fast-forward merge), тогава Git просто ще избере последния за сливането, така че това работи добре.

Обаче, Git няма да опита да направи за вас дори и тривиално сливане. Ако къмитите на подмодул се разделят и трябва да се слеят, ще получите нещо такова:

```
$ 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.
```

Какво се случва тук? Git е установил, че двата клона пазят точки от историята на

подмодула, които са разделени и трябва да се слеят. Това установяваме от съобщението "merge following commits not found", което само по себе си е объркващо, но ще обясним какво иска да каже след малко.

За да решите проблема, трябва да установите в какъв статус би трябвало да е модула. Странно, но Git тук в действителност не ви помага с информация, не посочва дори SHA-1 хеш стойностите на къмитите от двете страни на историята. За щастие, това е лесно за извличане. Ако пуснете `git diff` можете да получите SHA-1 стойностите на къмитите в двата клона, които сте се опитали да слеете.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

В този случай, `eb41d76` е къмитът в нашия подмодул, който **ние** сме имали и `c771610` е upstream къмита. Ако отидем в поддиректорията на модула, той вече трябва да е на `eb41d76`, защото сливането не го е променило. Ако по някаква причина не е, може просто да създадете нов клон сочещ към него.

Важната част е SHA-1 стойността на къмита от другата страна. Това е, което трябва да слеете и коригирате. Можете или да опитате сливане с SHA-1 стойността директно или може да създадете клон за нея и да опитате сливането в него. Бихме препоръчали второто, дори само за да създадете по-добро merge commit съобщение.

И така, ще отидем в поддиректорията, ще създадем клон наречен "try-merge", базиран на тази втора SHA-1 стойност, изведена от `git diff` и ще слеем ръчно.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

$ 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.
```

Тук получихме конфликт, така че ако го разрешим и къмитнем, тогава просто можем да обновим главния проект с резултата.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
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 ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

① Първо разрешаваме конфликта.

② Връщаме се в директорията на основния проект.

③ Можем да проверим SHA-1 стойностите отново.

④ Разрешаваме конфликтния submodule обект.

⑤ Къмитваме сливането ни.

Може да е леко смущаващо, но не е толкова трудно.

Съществува и още един интересен случай, който Git обработва. Ако в директорията на подмодула съществува merge къмит, който съдържа **и двата** къмита в историята си, Git ще ви го предложи като възможно решение. Той вижда, че на даден етап от този submodule проект, някой вече е слял клоновете съдържащи тези два къмита, така че може да искате този къмит.

Ето защо объркващото съобщение от по-горе гласеше “merge following commits not found”, защото системата не може да направи **това**. Объркващият текст идва, защото едвали някой би очаквал Git дори да **опита** да направи това.

Ако вместо това Git намери приемлив сливащ къмит, ще видите нещо от рода:

```
$ 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.
```

Това, което ви се предлага е да обновите индекса както ако бяхте изпълнили `git add` (което изчиства конфликта) и след това да къмитнете. Но вероятно не трябва да правите това. Можете също толкова лесно да влезете в поддиректорията, да видите каква е разликата, да направите fast-forward до този къмит, да го тествате и след това да къмитнете.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forward to a common submodule child'
```

Това ви дава същия резултат, но най-малкото по този начин може да проверите, че нещата работят и разполагате с кода в поддиректорията когато сте готови.

Съвети

Можете да улесните работата си с подмодули по няколко начина.

Foreach команда

Съществува `foreach` команда, която ви позволява да изпълнявате други команди за всеки подмодул, като в цикъл. Това е полезно, ако имате много подмодули в проекта.

Да кажем, че стартираме работа по нова функционалност или оправяме грешка и същевременно имаме текущо свършена работа в няколко подмодула. Можем лесно да маскираме (stash) работата по всички подмодули наведнъж.

```
$ 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
```

След това, можем да създадем и превключим към нов клон във всички подмодули едновременно.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

В общи линии получавате представа. Едно наистина полезно нещо е, че можете да изведете добре изглеждащ унифициран diff на това какво се е променило в главния проект и също във всички подмодули.

```
$ 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;
```

Тук се вижда, че дефинираме функция в подмодул и я викаме в главния проект. Очевидно примерът е прост, но да се надяваме че получавате представа как foreach може да бъде полезна опция.

Псевдоними

Някои от тези команди може да са доста дълги и неудобни за писане, а и не за всички може да направите подразбиращи се настройки в конфигурацията. Така че, вероятно бихте искали да създадете псевдоними за тях. Ние погледнахме как се правят псевдоними в Git в Псевдоними в Git, но ето пример какво бихте могли да зададете, ако планирате да работите интензивно с подмодули.

```
$ 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'
```

По този начин можете просто да изпълните `git supdate`, когато искате да обновите подмодулите си или `git spush` за да публикувате с dependancy checking за тях.

Възможни проблеми

Ползването на подмодули си има и неудобства.

Превключване на клонове

Така например, превключването на клонове с подмодули в тях може да е проблематично при версии на Git по-стари от 2.13. Ако създадете клон, добавите подмодул в него и след това превключите обратно към клон без този подмодул, ще получите поддиректорията му като непроследявана директория в проекта:

```
$ git --version
git version 2.12.2

$ 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 'Add crypto library'
[add-crypto 4445836] Add 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)
```

Изтриването на тази директория е лесно, но е малко смущаващо да трябва да правите това. Ако я премахнете и след това превключите обратно към клона с подмодула, ще трябва да изпълните `submodule update --init` за да възстановите данните в нея.

```
$ git clean -ffdx
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
```

Не е твърде сложно, но е някак неприсъщо.

Версиите на Git от 2.13 и нагоре опростяват всичко това добавяйки флага `--recurse-submodules` към командата `git checkout`, която се грижи за поставянето на подмодулите в правилния статус за клона, към който превключваме.

```
$ git --version
git version 2.13.3

$ 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 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master
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'.

nothing to commit, working tree clean
```

Използването на флага `--recurse-submodules` към `git checkout` може да е полезно и когато работите по множество клонове в подпроекта, във всеки от които подмодула ви сочи към различни къмити. Наистина, ако превключите между клонове, които пазят подмодула в

различни къмити, `git status` ще рапортува подмодула като "modified" и ще индикира "new commits". Това е така, защото по подразбиране статуса на подмодула не се обслужва по време на превключване между клоновете.

Това може да е наистина смущаващо, така че добра идея е винаги да използваме `git checkout --recurse-submodules`, когато проектът съдържа подмодули. За по-стари версии на Git, които не поддържат флага `--recurse-submodules`, след извличането може да изпълните `git submodule update --init --recursive` за да поставите подмодулите в правилния режим.

За щастие, може да инструктирате Git (>=2.14) винаги да използва флага `--recurse-submodules` през конфигурационната опция `submodule.recurse`: `git config submodule.recurse true`. Както отбелязахме по-горе, това също така ще накара Git да минава рекурсивно по подмодулите за всяка команда, която поддържа `--recurse-submodules` (с изключение на `git clone`).

Превключване от поддиректории към подмодули

Друг значим проблем, който много хора срещат, идва при превключването от поддиректории към подмодули. Ако проследявате файлове в проекта ви и искате да ги преместите в подмодул, трябва да внимавате или ще си навлечете гнева на Git. Допускаме, че имате файлове в директория от проекта и искате да я прехвърлите към подмодул. Ако изтриете поддиректорията и след това изпълните `submodule add`, Git ще откаже това:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Трябва първо да извадите от индекса директорията `CryptoLibrary`. След това можете да добавите подмодула:

```
$ 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.
```

Сега представете си, че сте направили това в отделен клон. Ако се опитате да превключите обратно към клон (примерно `master`), в който тези файлове все още са в действителното дърво, а не в подмодул — ще получите грешка:

```
$ 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
```

Можете да форсирате превключването с `checkout -f`, но внимавайте да нямате незаписани промени там, защото те могат да бъдат презаписани с тази команда.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

След това, когато превключите обратно, получавате празна `CryptoLibrary` директория по някаква причина и `git submodule update` може да не може да оправи това. Ще трябва да отидете в поддиректорията и да изпълните `git checkout .` за да си върнете файловете. Бихте могли да стартирате това в `submodule foreach` скрипт, за да го изпълните за повече подмодули.

Важно е да отбележим, че подмодулите в днешни дни пазят всичките си Git данни в `.git` директорията на главния проект, така че унищожаването на директория на подмодул няма да изтрие клоновете или къмитите, които сте имали. Това беше възможно само при много остарели версии на Git.

С тези инструменти, подмодулите в Git могат да са ефективен и прост начин за разработка в множество взаимно свързани, но все пак независими проекта едновременно.

Пакети в Git (Bundling)

Вече разгледахме стандартните начини за трансфер на Git данни по мрежата (HTTP, SSH и т.н.), но има и още един начин да правим това, който е не толкова често използван, а може да бъде полезен.

Git може да "пакетира" своите данни в единичен файл. Това може да е ценно в различни ситуации. Може би мрежата ви е повредена, а искате да изпратите промените си до колегите. Може да работите някъде далеч офлайн и да нямате достъп до локалната офис мрежа. Дори може мрежовата ви карта да е повредена. Може да нямате достъп до споделен сървър за момента, искате да изпратите по имейла промени на някого, но не желаете да изпращате много къмити през `format-patch`.

Тук може да помогне командата `git bundle`. Тази команда ще пакетира всичко, което нормално би изпратено по мрежата с `git push` в единичен бинарен файл, който може да се изпрати по имейл или чрез флашка и след това да се разпакетира в друго хранилище.

Нека видим прост пример. Да кажем, че имате хранилище с два къмита:

```
$ 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
```

Ако искате да го изпратите към някой друг, но нямате достъп до хранилище, в което да публикувате или пък просто не искате да правите такова, можете да пакетирате хранилището с `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)
```

Сега ще имате локален файл с име `repo.bundle`, който съдържа всичко необходимо за пресъздаване на `master` клона на хранилището ви някъде другаде. С командата `bundle` трябва да посочите всяка референция или специфичен набор от къмити, които искате да бъдат включени. Ако желаете нещата да бъдат клонирани на друго място, трябва да посочите HEAD като референция както направихме току що.

Можете да изпратите файла по имейл или да го копирате на флашка за когото е нужно.

Ако сте от другата страна и имате вече такъв файл, ето как да го използвате. Клонираме от бинарния файл в директория също както от URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 Second commit
b1ec324 First commit
```

Ако HEAD не е бил включен в референциите, ще трябва също да укажем `-b master` или съответното име на клон, така че Git да знае кой клон да извлече в работната директория.

Нека сега предположим, че сме направили промени в три къмита и искаме да ги изпратим

обратно в пакет на USB флашка или по имейла.

```
$ git log --oneline
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
9a466c5 Second commit
b1ec324 First commit
```

Първо, трябва да определим обхвата от къмити, които да включим в пакета. За разлика от случая с мрежовите протоколи, които определят това вместо нас, ще трябва да го подадем ръчно. Можете да направите същото нещо като в началото и да пакетирате цялото хранилище, това ще работи, но по-елегантно е да пакетирате само разликите, тоест само трите къмита направени локално.

За да направите това, ще трябва да калкулирате разликите. Както описахме в Обхвати от къмити, можете да указвате обхват от къмити по няколко начина. За да вземем само локалните три къмита, които липсват в оригинално клонирания клон, бихме могли да използваме нещо като `origin/master..master` или `master ^origin/master`. Можем да тестваме това с командата `log`.

```
$ git log --oneline master ^origin/master
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
```

След като сега имаме списъка къмити, нека ги обединим в пакет. Правим това със същата `git bundle create` команда, предоставяйки ѝ като параметри името на пакетния файл и обхвата от къмити.

```
$ 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)
```

Сега имаме файла `commits.bundle` в директорията ни. Ако го изпратим на наш колега, той/тя може да го импортира в оригиналното хранилище дори ако там междувременно е свършена и друга работа.

Другата страна има възможност да инспектира съдържанието на пакета преди импорта. Първата команда е `bundle verify` и тя ще провери, че файлът в действителност е валиден Git пакет и че налице са всички необходими родителски обекти за коректното му импортиране.

```
$ 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
```

Ако човекът направил пакета беше го сглобил само от двата последни къмита вместо от всичките три, оригиналното хранилище няма да може да ги внедри поради липса на пълна история. В такъв случай изходът от командата `verify` би изглеждал така:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit - second repo
```

В нашия случай пакетът си е валиден, така че можем да извлечем къмитите от него. Ако искате да видите какви клонове съдържа пакета, също можете да го направите:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Подкомандата `verify` може също да ви даде тази информация. Целта е да се види какво може да бъде слято, така че можем да използваме `fetch` или `pull` за да импортираме къмити от този пакет. Тук ще издърпаме `master` клона от пакета в клон наречен `other-master` в нашето хранилище:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

Сега можем да видим, че имаме импортираните къмити в клона `other-master` както и междувременно направените такива в нашия собствен `master` клон.

```
$ 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
```

И така, видяхме как `git bundle` може да е ценен помощник за споделяне на работа, когато не разполагаме с необходимата мрежова свързаност.

Заместване

Както вече подчертахме, обектите в базата данни на Git са непроменими, но Git предлага интересен начин да се *преструва*, че замества обекти в базата си данни.

Командата `replace` ви позволява да укажете един специфичен обект в Git и да кажете "всеки път, когато се обръщаме към *този* обект, третирай го като *различен* такъв". Това най-често е полезно за заместване на един къмит в историята с друг такъв без необходимост от преправяне на цялата история с `git filter branch` например.

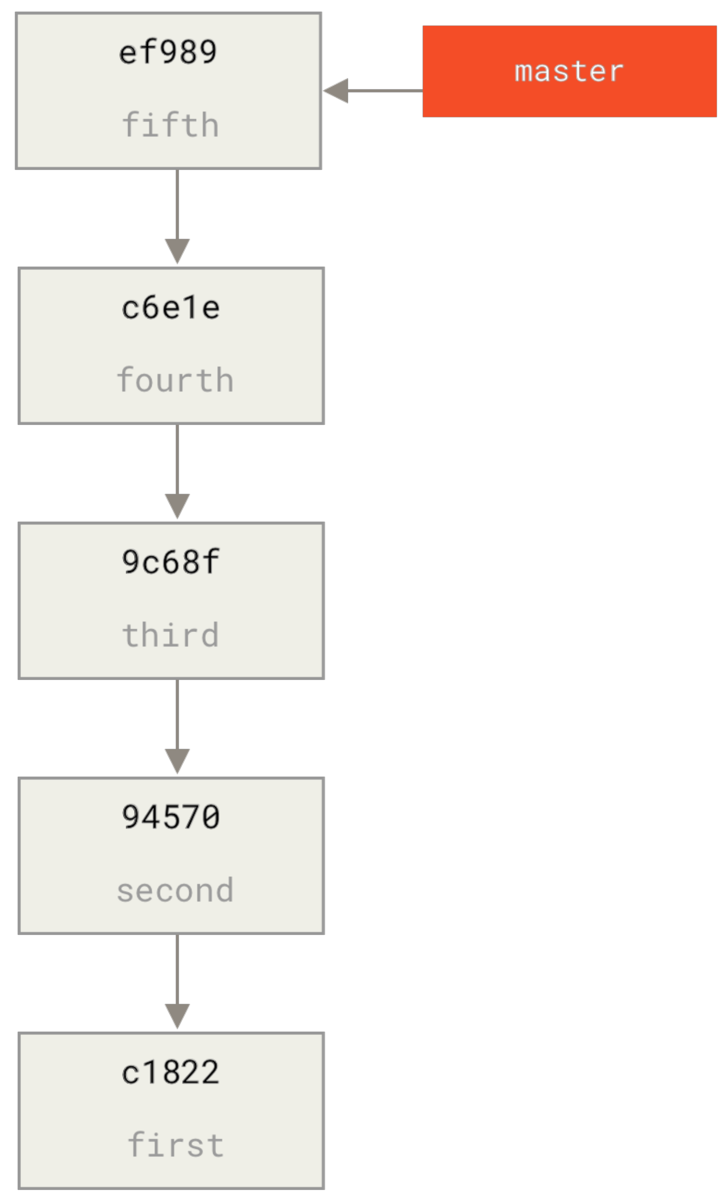
Нека кажем, че имате обширна история за даден проект и искате да я разделите на една по-кратка част за новите разработчици и една много по-дълга за хората, които искат да изследват кода в дълбочина. Можете да присадите едната история в другата "заместvайки" най-ранния къмит в новата линия с най-късния от по-старата. Това е добре, защото означава, че в действителност не трябва да пренаписвате всеки къмит в новата история, което нормално бихте направили за да ги обедините в едно (защото наследствеността засяга SHA-1 хешовете).

Нека да опитаме. Ще вземем налично хранилище, ще го разделим в две отделни, едно актуално и едно хронологическо и след това ще видим как с `replace` можем да ги комбинираме повторно без да модифицираме SHA-1 стойностите на новополучените.

Използваме хранилище с пет кратки къмита:

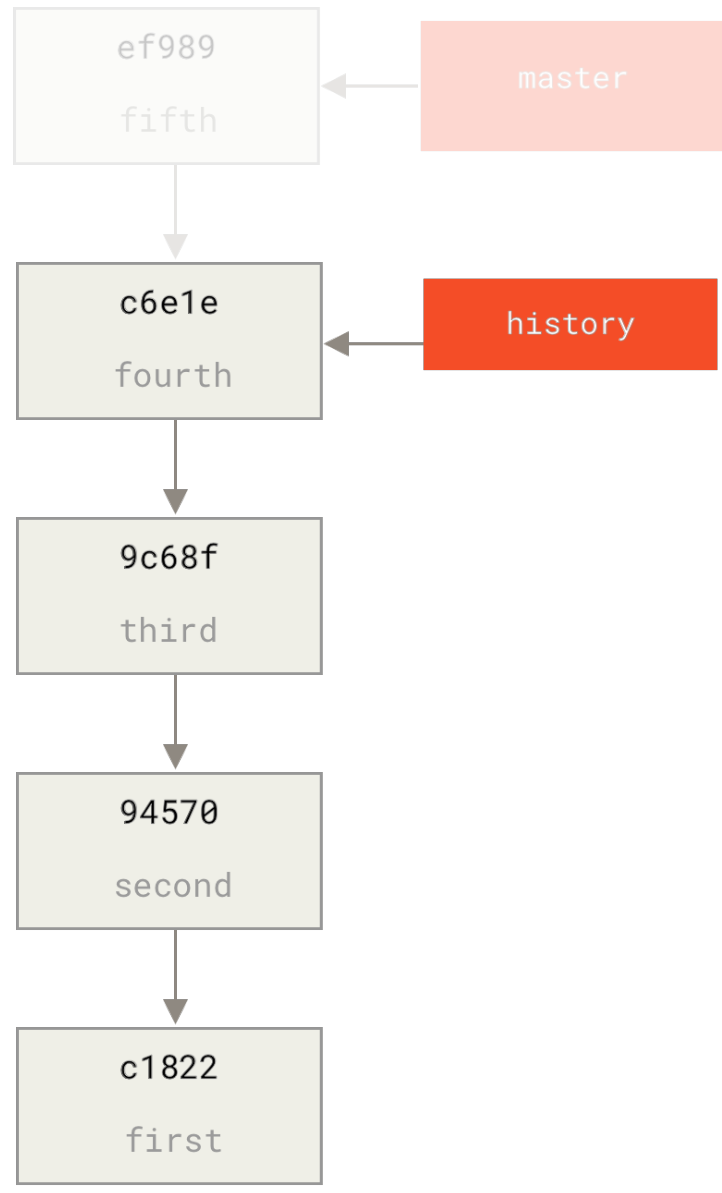
```
$ git log --oneline
ef989d8 Fifth commit
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Искаме да разделим това на две линии история. Едната линия минава от къмити 1 до 4 - това ще ни е хронологическата линия. Втората линия ще е само с къмити 4 и 5 - това ще е актуалната история.



Създаването на хронологическата линия е лесно, просто създаваме клон до точка в историята и след това го публикуваме в `master` клона на ново отдалечено хранилище.

```
$ 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
```



Сега можем да публикуваме новия `history` клон към `master` клона в новото ни хранилище:

```
$ 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
```

Така историята ни е публикувана. По-трудната част е да орежем актуалната си история,

така че да стане по-кратка. Трябва ни обща пресечна точка (общ къмит), такава в която да можем да заменим къмит от едната линия с къмит в другата. Ето защо ще отрежем историята до два къмита - четвърти и пети (така че къмит 4 да е общ за двете страни).

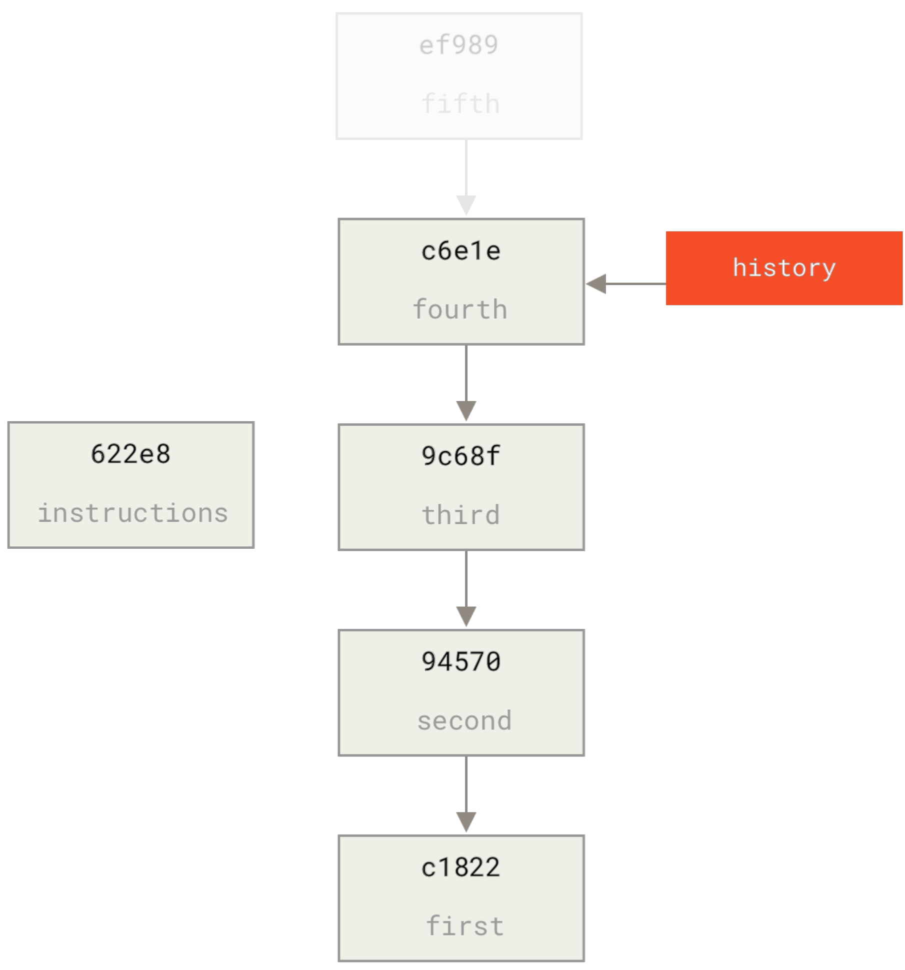
```
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

В този случай е полезно да създадем base къмит с инструкции за това как да се разшири историята, така че другите разработчици да знаят какво да правят ако срещнат първия къмит в съкратената история и се нуждаят от по-старите. Така че, това което ще направим, е да създадем начален къмит играещ ролята на изходна точка с инструкциите, след което да пребазираме останалите два къмита (четвърти и пети) върху него.

За да започнем, трябва да изберем точка за разделяне, която за нашия случай ще е в третия къмит, `9c68fdc`. Така base къмитът ни ще бъде базиран на това дърво. Можем да го създадем с командата `commit-tree`, която просто приема дърво и ще ни върне SHA-1 хеша на един нов къмит без родители.

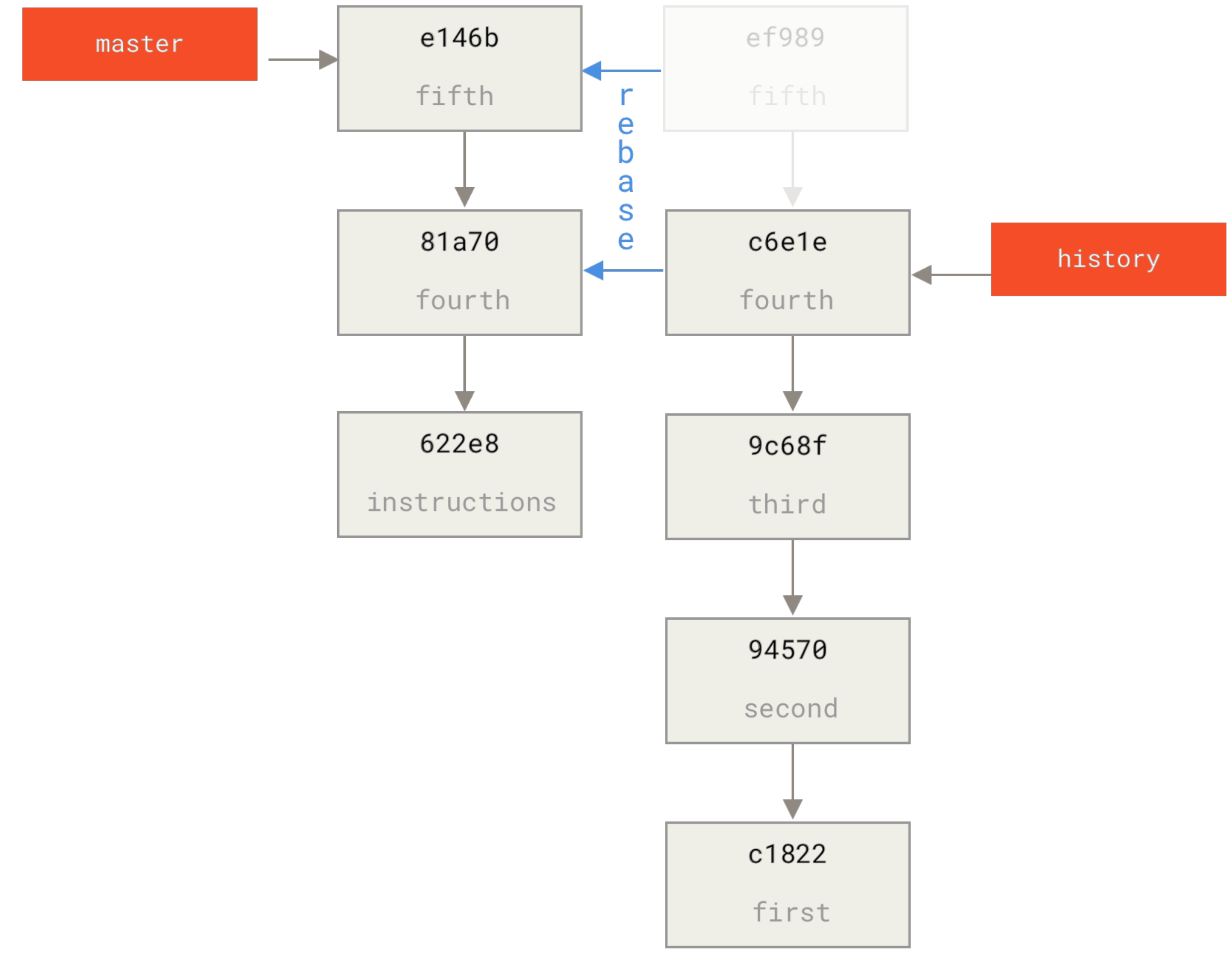
```
$ echo 'Get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

Командата `commit-tree` е една от командите в Git известни като *plumbing* команди. Това са команди предназначени за индиректно използване с **други** Git команди за извършване на по-малки дейности. В случаи като този, в който извършваме необичайни дейности, тези команди ни дават средства от по-ниско ниво, но като цяло не се използват в ежедневната работа. Повече за plumbing командите ще видим в Plumbing и Porcelain команди.



Сега имаме базов къмит и можем да пребазираме остатъка от историята ни върху него с `git rebase --onto`. Аргументите към `--onto` ще бъдат SHA-1 стойността, която получихме от `commit-tree`, както и точката на пребазиране, тоест третия къмит (родител на този, който искаме да пазим, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



Сега пренаписахме актуалната история базирайки я на начален къмит съдържащ в себе си инструкции за това как да реконструираме цялата история, ако поискаме това. Можем да публикуваме тази история в нов проект и когато хората клонират това ново хранилище те ще видят само последните ни два къмита и базов къмит с инструкции.

Нека сега се поставим на мястото на тези хора и да видим как можем да се сдобием с пълната история на проекта. За да вземем хронологическите данни след клонирането на това орязано хранилище, трябва да добавим втора отдалечена референция към хронологическото хранилище и да изтеглим:

```
$ 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
```

Сега актуалните къмити са в клона `master` а хронологичните в `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
```

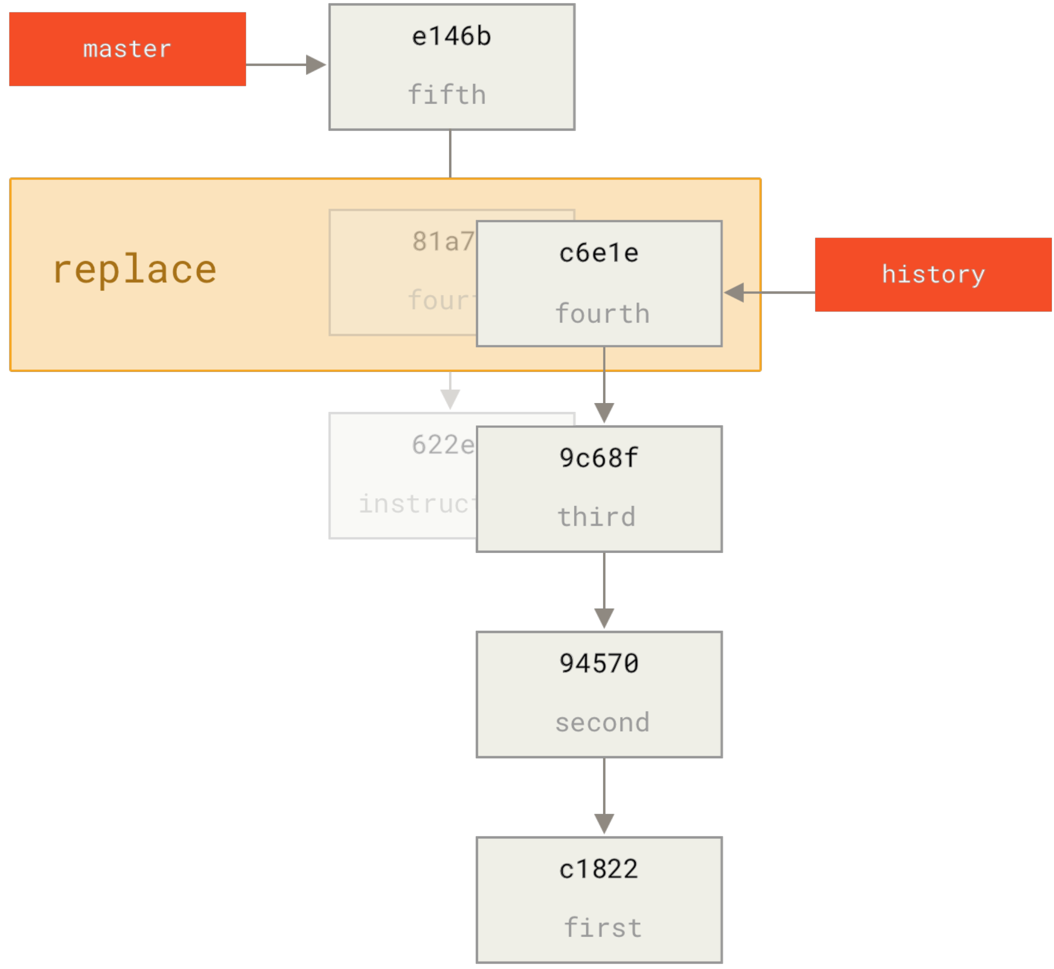
За да ги комбинираме, можем да изпълним `git replace` с къмита, който искаме да заместим и този, с който искаме да го заместваме. Ние искаме да заменим "fourth" къмита в `master` клона с "fourth" къмита от `project-history/master`:

```
$ git replace 81a708d c6e1e95
```

Сега, ако прегледаме историята на клона `master`, тя е подобна:

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Изглежда наистина много добре, защото без да се налага да променяме цялата SHA-1 верига успяхме да заместим един къмит от историята с изцяло друг къмит и всички нормални инструменти (като `bisect`, `blame`) ще работят както бихме очаквали от тях.



Обаче, историята все още показва `81a708d` като SHA-1 стойност, въпреки че реално се използват данните от `c6e1e95` с който заместихме. Дори ако изпълните команда като `cat-file`, тя ще ви покаже заменените данни:

```
$ 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
```

Спомнете си, че действителният родител на `81a708d` беше нашия placeholder къмит (`622e88e`), а не `9c68fdce` както се твърди.

Друг интересен момент е, че тази информация се пази в референциите ни:

```
$ 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/81a708dd0e167a3f691541c7a6463343bc457040
```

Това означава, че е лесно да споделим подмяната си с други хора, защото можем да я публикуваме в сървъра ни и те лесно могат да я свалят. Това не е така полезно в сценария, който следвахме тук (понеже така или иначе всеки ще изтегля и двете истории и е излишно да ги разделяме), но може да е полезно в други ситуации.

Credential Storage система

Ако използвате SSH като транспортен протокол, възможно е да имате ключ без passphrase, позволяващ ви защитѐн трансфер на данни без въвеждане на име и парола. Обаче това не е така при HTTP протоколите — всяка връзка изисква да ги въвеждате. Това става дори по-сложно при системи с двустъпкова автентикация, където стрингът изпълняващ ролята на парола се генерира случайно и е невъзможен за произнасяне.

За щастие Git разполага с credentials система на ваше разположение. Имате няколко фабрични опции:

- По подразбиране не се кешира нищо. Всяка конекция ще изисква име и парола.
- Режимът “cache” запазва данните в паметта за определен период от време. Никоя от паролите не се записва на диска и те се изчистват от кеша на всеки 15 минути.
- Режимът “store” съхранява данните за достъп в обикновен текстов файл на диска и те не се премахват от там. Това значи, че докато не смените паролата си на Git хоста, няма да е нужно да я въвеждате. Недостатъкът е, че паролите ви се съхраняват в чист текст в домашната директория.
- Ако използвате Mac, Git предлага “osxkeychain” режим, при който данните за достъп се съхраняват в защитена keychain прикрепена към системния ви акаунт. Този метод пази данните на диска, те остават валидни, но са криптирани със същата система, която съхранява HTTPS сертификати и auto-fill данните на браузъра Safari.
- Под Windows, можете да инсталирате helper наречен “Git Credential Manager for Windows.” Това е подобно на “osxkeychain”, но използва Windows Credential Store за контрол на поверителната информация. Може да се намери на https://github.com/Microsoft/Git-Credential-Manager-for-Windows.

Избирате един от тези методи чрез конфигурацията на Git:

```
$ git config --global credential.helper cache
```

Някои от helper-ите имат опции. Например “store” може да приема аргумент `--file <path>`,

който определя къде да се пази текстовия файл (по подразбиране `~/.git-credentials`). При “cache” е наличен аргумента `--timeout <seconds>`, с който можете да промените времето на работа на неговия daemon (по подразбиране в секунди е “900”, т.е. 15 минути). Ето как да променим името на файла при “store” helper-а:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git позволява конфигурирането и на няколко helper-а. Търсейки данните за достъп до определен хост, Git ще ги изпитва един по един и ще спре при първия получен отговор. При запис на данни за достъп, Git ще изпраща името и паролата към **всички** helper-и в списъка и те поемат грижата за това как да ги ползват. Ето как би изглеждал `.gitconfig`, ако пазите данните за достъп на външно устройство, но искате да използвате in-memory кеша, за да си спестите малко писане, когато стикчето не е включено:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Зад кулисите

Как работи всичко това? Основната команда на Git за credential-helper системата е `git credential`, която приема допълнителна команда като аргумент и данни от стандартния вход.

По-лесно за разбиране е с пример. Нека кажем, че имаме настроен credential helper и че той пази данните за достъп до `mygithost`. Ето една сесия, която използва “fill” командата, която се изпълнява, когато Git се опитва да намери данните за достъп до даден хост:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① Това е командата, която стартира поредицата.

② Git-credential след това чака за вход от stdin. Ние подаваме нещата, които знаем: протокола и името на хоста.

③ Празният ред показва, че входът е приключил и credential системата трябва да отговори с това, което тя знае.

④ Git-credential поема инициативата и изпраща в stdout намерените данни.

⑤ Ако данните за достъп не са намерени, Git пита за име и парола и ги връща обратно към stdout (тук те са в една и съща конзола).

Credential системата реално извиква програма, която не е част от Git — коя и как зависи от стойността на конфигурационния ключ `credential.helper`. Това може да изглежда по няколко начина:

Configuration Value	**Behavior**
`foo`	Runs `git-credential-foo`
`foo -a --opt=bcd`	Runs `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Runs `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Code after `!` evaluated in shell

Така helper-ите описани по-горе в действителност са наречени `git-credential-cache`, `git-credential-store` и т.н. и можем да ги настроим да приемат аргументи от командния ред. Шаблонът за това е “git-credential-foo [args] <action>.” Протоколът stdin/stdout е същият като git-credential, но те използват леко различни набори от действия:

- `get` е запитване за чифт от име и парола.
- `store` е заявка за запис на набор от данни за достъп в паметта на този helper.
- `erase` изтрива данните за достъп до дадени свойства от паметта на този helper.

За `store` и `erase` действията не се изисква отговор (и да има такъв Git го игнорира). Обаче за действието `get`, Git е силно заинтересован за това, което има да каже helper-а. Ако той не знае нищо важно, Git може просто да излезе без отговор, но ако знае трябва да сравни предоставената информация с тази, която е съхранена. Изходът се третира като серия от assignment оператори и всичко подадено ще замени това, което Git вече знае.

Ето примера отгоре, но с пропуснат git-credential и преминаване направо към git-credential-store:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① Тук казваме на `git-credential-store` да съхрани малко данни за достъп: потребителското име "bob"и паролата ``s3cre7" трябва да се ползват за достъп до адрес `https://mygithost`.

② Сега ще извлечем тези данни. Предоставяме частите от конекцията, които вече знаем (`https://mygithost`) и след това празен ред.

③ `git-credential-store` отговаря с името и паролата, които записахме по-рано.

Ето как изглежда файла `~/git.store`:

```
https://bob:s3cre7@mygithost
```

Той е просто серия от редове, всеки от които съдържа URL форматиран така, че да подава името и паролата. Helper-ите `osxkeychain` и `wincred` използват нативния формат на техните системи за съхранение, докато `cache` използва свой собствен in-memory формат (който е недостъпен за четене от други процеси).

Потребителски Credential Cache

Предвид това, че `git-credential-store` и подобните са отделни програми от Git, лесно е да се досетим, че *всяка* програма би могла да играе ролята на Git credential helper. Helper-ите осигурени от Git покриват много от най-честите сценарии за използване, но не всички. Да допуснем, че екипът ви пази данни за достъп споделени между всички разработчици в него, например за цели свързани с внедряване на продукти. Те се пазят в споделена директория, но не желаете да ги запазвате локално в собствения ви credential store, защото често се променят. В този случай, нито един от стандартните helper-и не отговаря на ситуацията и вариантът е да си напишем свой собствен такъв. Хипотетичната ни помощна програма трябва да има няколко ключови способности:

1. Единственото действие, с което трябва да се съобразява е `get`; `store` и `erase` са записващи действия, така че просто ще излизаме чисто, когато ги получим.
2. Файловият формат на споделения credential файл е същият като този използван от `git-credential-store`.
3. Локацията на файла е сравнително стандартна, но бихме искали да позволим на потребителя да подаде специфичен такъв просто за всеки случай.

Отново, ще напишем програмата си на Ruby, но всеки език би работил стига Git да може да изпълни финалната програма. Ето пълния изходен код на нашия нов credential helper:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
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' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| ④
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] and user ==
known['username'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① Първо парсваме аргументите от командния ред, позволявайки на потребителя да укаже входен файл. По подразбиране използваме `~/.git-credentials`.

② Програмата отговаря само на действието `get` и само ако въпросния файл съществува.

③ Цикълът чете вход от stdin докато срещне празен ред. Подадените стойности се пазят в `known` хеша за по-късно ползване.

④ Този цикъл чете съдържанието на storage файла, търсейки съвпадения. Ако протоколът и хостът от `known` съответстват на дадения ред, програмата печата резултатите в stdout и спира.

Ще запишем нашия helper като `git-credential-read-only`, ще го добавим към `PATH` променливата и ще го направим изпълним. Ето как изглежда интерактивната сесия:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Понеже името му започва с “git-”, можем да използваме опростения синтаксис за конфигурация:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

Както виждате, разширяването на системата е доста праволинейно и може да бъде полезно за вас и вашия екип.

Обобщение

Разгледахме множество разширени инструменти, позволяващи ви да манипулирате вашите къмити и индексната област по прецизен начин. Когато срещнете проблем, би следвало да можете лесно да откриете кой къмит го е причинил, кога и от кого идва. Ако искате да ползвате вложени проекти (подмодули) в основния такъв, научихте как да го направите. На този етап, вече би следвало комфортно да можете да правите повечето неща в Git, от които ще имате нужда в ежедневната си работа.

Настройване на Git

Досега прегледахме основите на работата с Git, представихме и множество инструменти, помагащи ни да работим по-лесно и ефективно. В тази глава ще видим как да накараме Git да работи в по-специфични режими посредством няколко важни конфигурационни настройки и hooks системата. С тези инструменти е лесно да накарате Git да работи точно както желаете вие, вашия екип или компанията ви.

Git конфигурации

Както погледнахме накратко в Начало, можете да използвате командата `git config` за манипулация на конфигурационни настройки. Едно от първите неща, които направихме беше да зададем име и имейл адрес:

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

Сега ще разгледаме повече интересни опции, които можем да променяме по този начин, така че да настроим по-фино поведението на Git системата.

Първо, кратък преглед: Git използва серия от конфигурационни файлове, за да определи нестандартното поведение, което бихте могли да искате. Първото място, което Git проверява, е system-wide файла `[path]/etc/gitconfig`, който съдържа множество настройки валидни за всички потребители и всички хранилища в машината. Ако подадете опцията `--system` на `git config`, командата чете и пише точно в този файл.

След това Git търси файла `~/.gitconfig` (или `~/.config/git/config`) в потребителските директории, в който се съхраняват настройки специално за конкретния потребител на операционната система. Този е засегнатия файл, когато подавате на командата аргумента `--global`.

Най-накрая Git проверява за конфигурационни настройки във файл в конкретното текущо хранилище (`.git/config`). Тези стойности са специфични само за конкретното хранилище и се подават с аргумента `--local` към `git config`. Ако не укажете аргумент за обхват на командата, именно това локално ниво се използва по подразбиране.

Всяко от тези "нива" (system, global, local) презаписва стойностите от предишното, така че стойностите в `.git/config` са с приоритет пред тези в `[path]/etc/gitconfig` например.

Конфигурационните файлове на Git са чист текст и можете да редактирате файловете директно, спазвайки съответния синтаксис. Все пак, вероятно е по-лесно да използвате `git config`.

Основни конфигурации на клиента

Конфигурационните опции, които Git използва попадат в две главни категории: клиентска и сървърна. Болшинството от опции са клиентски — настройват персоналните ви

предпочитания за работа. Поддържат се *огромен* брой опции, но голяма част от тях се използват само в специфични случаи, ето защо ще разгледаме най-популярните и полезни. Ако искате списък на всички, можете да изпълните:

```
$ man git-config
```

Тази команда изброява и обяснява всички налични опции в подробности. Алтернативно място да получите тази информация е https://git-scm.com/docs/git-config.html.

`core.editor`

По подразбиране, Git използва настро ения за вашия персонален акаунт текстов редактор през environment променливите `VISUAL` или `EDITOR` и ако такъв няма, използва vi за манипулация на къмит съобщенията и таговете. Ако искате да промените редактора, използвайте настройката `core.editor:

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

След това, без оглед на подразбиращия се шел редактор, Git ще стартира Emacs за редакция на съобщенията.

`commit.template`

Ако настроите тази опция да сочи към файл в компютъра, Git ще използва съдържанието на този файл като подразбиращо се начално съобщение когато къмитвате. Предимството в това да имате шаблон е, че можете да го използвате като припомяне за вас (а и за останалите) как да се пишат и форматират добри къмит съобщения.

Например, имаме файла `~/.gitmessage.txt`:

```
Subject line (try to keep under 50 characters)

Multi-line description of commit,
feel free to be detailed.

[Ticket: X]
```

Забелязваме как този шаблон припомня на разработчика да поддържа subject реда кратък (за по-красив `git log --oneline` изход), да добавя подробности под него и да упоменава issue или номер на тикет в bug tracker система, ако има такава.

Задаването на шаблона се прави с командата:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

След което, текстовият редактор ще изглежда така, когато бъде стартиран:

```
Subject line (try to keep under 50 characters)

Multi-line description of commit,
feel free to be detailed.

[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
```

Ако екипът ви спазва commit-message политика, тогава използването на един такъв шаблон би било много полезно и увеличава шансовете за стриктно спазване на политиката.

`core.pager`

Тази настройка определя кой пейджър да се използва, когато Git странира изхода от команди като `log` и `diff`. Можете да я настроите на `more` или на нещо друго (по подразбиране е `less`), а също и можете да изключите странирането задавайки ѝ празен стринг:

```
$ git config --global core.pager ''
```

Така Git ще показва наведнъж целия изход от всички команди без значение от дължината му.

`user.signingkey`

Ако правите signed annotated тагове (както видяхме в Подписване на вашата работа), задаването на вашия GPG подписващ ключ като конфигурационна настройка, ще ви спести писане. Задава се така:

```
$ git config --global user.signingkey <gpg-key-id>
```

Сега можете да подписвате тагове без да трябва да указвате ключа си всеки път, когато пуснете `git tag`:

```
$ git tag -s <tag-name>
```

`core.excludesfile`

В Игнориране на файлове видяхме как да създаваме маски в `.gitignore` файл, така че Git да не вижда и да не се опитва да индексира определени файлове в проекта.

Понякога обаче е по-удобно да игнорирате дадени файлове във всички хранилища, с които работите. Ако използвате macOS, вероятно сте запознати с `.DS_Store` файловете. Ако предпочитаният ви редактор е Emacs или Vim, знаете за файловите имена, които завършват на `~` или `.swp`.

Тази настройка дава възможност за нещо като глобален `.gitignore` файл. Ако създадете файл `~/.gitignore_global` с това съдържание:

```
*~
.*.swp
.DS_Store
```

…и изпълните `git config --global core.excludesfile ~/.gitignore_global`, Git въобще няма да обръща внимание на подобни файлове във всички хранилища.

`help.autocorrect`

Ако сбъркате команда, Git показва нещо от рода:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

The most similar command is
    checkout
```

Услужливо ви предлага да отгатне какво имате предвид, но все пак не изпълнява предполагаемата команда. Ако обаче зададете `help.autocorrect` със стойност 1, Git ще изпълни командата:

```
$ 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...
```

Забележете съобщението “0.1 seconds”. `help.autocorrect` стойността в действителност е цяло число, което представлява десета от секундата. Така че, ако го промените на 50, Git ще ви даде 5 секунди да размислите преди да стартира командата, която предполага че искате.

Цветове в Git

Git има пълна поддръжка за цветен изход в терминала, което помага много за лесното разчитане на информацията от потребителя. Имате множество опции за настройка на

цветните предпочитания.

`color.ui`

Git автоматично оцветява повечето от изхода на командите си, но разполагате с главен ключ, ако не искате това. За да изключите всякакво оцветяване, изпълнете:

```
$ git config --global color.ui false
```

Стойността по подразбиране е `auto`, което оцветява изхода в терминала, но пропуска color-control кодовете, ако изходът е пренасочен към pipe или към файл.

Ако искате оцветяване навсякъде, настройката също приема и стойността `always`. Вероятно малко хора ще искат да правят това, в повечето случаи ако искате цветове в пренасочения изход, можете да подадете флага `--color` към конкретната единична команда. В почти всички случаи настройките по подразбиране ще са това, което очаквате.

`color.*`

Можете да бъдете и още по-прецизни в избора на това изходът от кои команди да се оцветява и как точно. Всяка от тези опции може да е `true`, `false`, или `always`:

```
color.branch
color.diff
color.interactive
color.status
```

В допълнение, всяка от тях има и поднастройки, които можете да използвате за да задавате специфични цветове на част от изхода им и да коригирате всеки един от зададените цветове. Например, ако искате метаданните във вашия diff изход да са със сини символи на черен фон и удебелен шрифт, може да направите така:

```
$ git config --global color.diff.meta "blue black bold"
```

Цветовете приемат стойности: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, или `white`. Ако искате специфичен атрибут за шрифта както беше bold преди малко, налични са вариантите `bold`, `dim`, `ul` (underline), `blink`, и `reverse` (размяна на цветовете на символите и фона).

Външни Merge и Diff инструменти

Въпреки, че Git има собствена вътрешна diff имплементация (която виждаме в действие в настоящата книга), можете да използвате и външен diff инструмент. Можете да си настроите графичен merge-conflict-resolution инструмент, вместо да трябва да коригирате конфликтите ръчно. Ще покажем как се настройва безплатния и удобен инструмент Perforce Visual Merge Tool (P4Merge) за да правите вашите diffs и merge resolutions.

Ако искате да го пробвате, P4Merge работи на всички основни платформи. В примерите ще използваме пътища, които се ползват в macOS и Linux, за Windows ще трябва да смените `/usr/local/bin` към съответния път в конкретната инсталация.

За начало, изтеглете P4Merge от Perforce. След това, ще създадем външни wrapper скриптове за изпълнение на командите ви. Ще използваме macOS пътя за изпълнимия файл, в други системи той ще е мястото, където се намира `p4merge` програмата. Създаваме merge wrapper скрипт наречен `extMerge`, който извиква програмата с всички необходими аргументи:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Скриптът за diff от своя страна проверява за подадени седем аргумента и изпраща два от тях към merge скрипта. По подразбиране, Git изпраща следните аргументи към diff програмата:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Понеже искаме само `old-file` и `new-file` аргументите, използваме wrapper скрипта, за да подаваме само тях.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Тези скриптове трябва да са изпълними:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Сега можем да използваме конфигурационния си файл да използва тези потребителски инструменти. Приемат се множество специфични настройки: `merge.tool` за да кажем на Git каква стратегия на сливане да ползва, `mergetool.<tool>.cmd` за начина на стартиране на командата, `mergetool.<tool>.trustExitCode` за да укажем на Git, че кода на изход на програмата индикира успешно/неуспешно сливане, и `diff.external` за командата използвана за diff. Така може да изпълните следните 4 конфигурационни команди:

```
$ 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
```

или пък директно да редактирате `~/.gitconfig` файла така:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Ако всичко това е направено и изпълните diff команда като тази:

```
$ git diff 32d1776b1^ 32d1776b1
```

Вместо да видите изхода на командния ред, Git ще стартира P4Merge, който би изглеждал подобно:

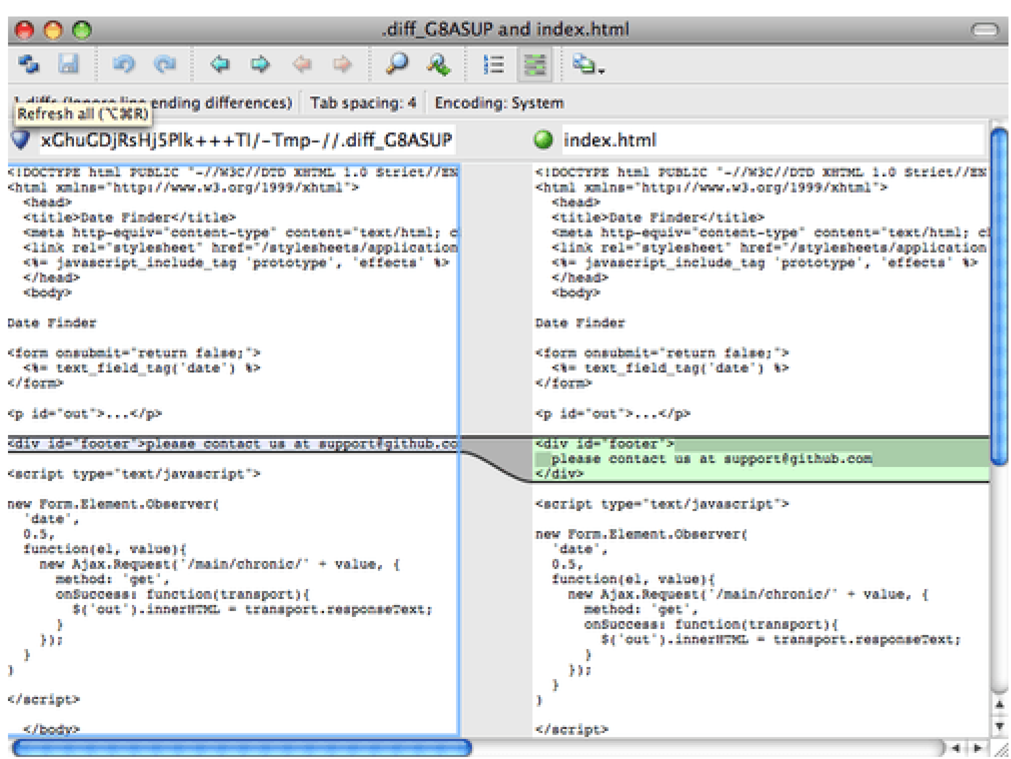


Figure 142. P4Merge

Ако опитате да слеете два клона и получите конфликти, може да изпълните `git mergetool` и тя от своя страна ще стартира P4Merge за да ви позволи да ги разрешите в графичен стил.

Удобното нещо на тези wrapper настройки е, че лесно можете да променяте diff и merge инструментите си. Например, ако желаете `extDiff` и `extMerge` скриптовете да пускат KDiff3 програмата, просто трябва да редактирате файла `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Сега вече Git ще използва KDiff3 за показване на diff информация и разрешаване на конфликти.

Git също така идва с известен брой предварително зададени външни merge-resolution инструменти за да не се налага да правите командната конфигурация. За да видите списък с тях, пробвайте това:

```
$ 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.
```

Ако не се интересувате от KDiff3 за diff, но искате да го използвате за корекция на конфликти и kdiff3 командата е в пътя ви, можете да изпълните:

```
$ git config --global merge.tool kdiff3
```

Ако направите това вместо да създавате `extMerge` и `extDiff` скриптовете, Git ще ползва KDiff3 за конфликти и вътрешния Git diff инструмент за diff визуализация.

Форматиране и празни символи

Проблемите с форматирането и празните символи са сред най-досадните неща, с които разработчиците се сблъскват по време на съвместна работа и особено ако работят на различни платформи. Много е лесно пачове или друга съвместна работа да внесат незабележими whitespace промени, защото редакторите ги вкарват задкулисно и ако файловете ви се озоват на Windows система, знаците им за край на ред може да се променят без предупреждение. Git разполага с няколко конфигурационни опции за да ви помогне в такива случаи.

`core.autocrlf`

Ако програмирате под Windows, а колегите ви не (или обратното), твърде вероятно е да възникнат проблемни ситуации в даден етап. Това е защото Windows използва за край на ред във файловете и двата символа (carriage-return и linefeed), докато под macOS/Linux се използва само linefeed. Това е незабележима за окото разлика, но постоянно предизвиква проблеми при многоплатформена работа, много редактори под Windows без да питат конвертират наличните LF знаци за край на ред в CRLF.

Git може да се справи с това конвертирайки CRLF символите в LF, когато добавяте файл в индекса и обратното, когато извличате файл в работната директория. Може да контролирате това поведение през настройката `core.autocrlf`. Ако сте на Windows машина, задайте `true` — това ще конвертира LF символите в CRLF, когато извличате файловете:

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

Ако сте на Linux или macOS, няма да искате автоматично конвертиране, обаче ако някой файл с CRLF внезапно се появи, бихте желали Git да го коригира. Можете да зададете CRLF към LF конвертирането да се прави при къмит със стойността input за `core.autocrlf`:

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

Това положение би трябвало да остави CRLF символите, когато извличате под Windows и LF символите под macOS и Linux, както и в хранилището.

Ако сте Windows програмист и работите по Windows проект, тогава може да изключите тази функционалност и да записвате CR символите в хранилището задавайки `false` за опцията:

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git също така разполага с възможност да разпознава и коригира някои проблеми с празните символи. Налични са инструменти за 6 главни ситуации — три са разрешени по подразбиране и могат да се изключат и три други по подразбиране са изключени, но могат да се активират.

Трите включени опции са `blank-at-eol`, която търси за интервали в края на ред; `blank-at-eof`, която усеща празни редове в края на файла, и `space-before-tab` търсеща за интервали преди табулации в началото на редовете.

Изключените фабрично опции са `indent-with-non-tab`, която търси редове започващи с интервали вместо с табулации (и се контролира с `tabwidth` опцията); `tab-in-indent`, която следи за наличието на табулации в indentation частта на редовете; и `cr-at-eol`, която казва на Git, че carriage return символите в края на редовете са OK.

Можете да кажете на Git кои от тези опции искате да са активни задавайки за `core.whitespace` стойностите, които искате да са включени/изключени, разделени със запетаи. Може да забраните опция добавяйки `-` преди името ѝ или да изисквате стойността по подразбиране като въобще не я включвате в стринга. Например, ако искате задаване на всички без `space-before-tab`, може да направите това (`trailing-space` е съкратено изписване за `blank-at-eol` и `blank-at-eof`):

```
$ git config --global core.whitespace \
    trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Или, може да укажете само специфичните промени:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Git ще намери проблемите при изпълнение на `git diff` и ще опита да ги оцвети така че да пробвате да ги поправите преди да къмитнете. Тези стойности се използват за ваше улеснение и когато прилагате пачове с `git apply`. Ако прилагате пачове, може да укажете на Git да ви предупреждава, ако те съдържат whitespace проблеми:

```
$ git apply --whitespace=warn <patch>
```

Или може да кажете на Git да се опита автоматично да ги реши преди да приложи пача:

```
$ git apply --whitespace=fix <patch>
```

Тези опции важат и за командата `git rebase`. Ако сте къмитнали whitespace нередности, но все още не сте публикували в upstream хранилището, можете да изпълните `git rebase --whitespace=fix` и така Git ще се опита да коригира нещата по същия начин, по който пренаписва пачовете.

Сървърни конфигурации

Конфигурационните опции за сървърната роля на Git не са толкова много на брой, но някои от тях заслужават да бъдат посочени.

`receive.fsckObjects`

Git може да проверява дали всеки обект получен по време на публикуване все още съответства на очакваната си SHA-1 стойност и дали сочи към валидни обекти. Но това по подразбиране не се прави, защото е ресурсоемка операция и може да причини забавяне при големи хранилища. Може да включите проверката задавайки true за `receive.fsckObjects`:

```
$ git config --system receive.fsckObjects true
```

Сега Git при всеки push ще проверява интегритета на хранилището преди да го приеме, така че некоректните (или умишлено злонамерени) клиенти да не могат да внесат повредени данни.

`receive.denyNonFastForwards`

Ако пребазирате къмити, които вече сте публикували и се опитате да ги публикувате пак, или пък се опитате да публикувате къмит към отдалечен клон и този къмит не съдържа къмита, към който този клон текущо сочи, ще ви бъде отказано. Това по принцип е добра политика, но в случая на rebase може да установите, че знаете какво точно правите и може да форсирате обновяването на отдалечения клон с `-f` флага на push командата.

Може да забраните форсираните публикувания задавайки `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Друг начин да направите това е чрез сървърни receive hooks, които ще разгледаме накратко. Този подход позволява да правим много по-сложни неща като например отказ за non-fast-forwards само за определени потребители.

`receive.denyDeletes`

По-напредналите потребители могат да заобиколят `denyNonFastForwards` политиките изтривайки клона и след това публикувайки го отново с новата референция. За да избегнете това, задайте `receive.denyDeletes` със стойност true:

```
$ git config --system receive.denyDeletes true
```

Това забранява изтриването на клонове и тагове — никой потребител няма да може да го прави. За да изтриете отдалечени клонове ще трябва да изтриете ref файловете от сървъра ръчно. Съществуват и по-интересни начини да правите това на per-user принцип през ACL, както ще видим в Примерна Git-Enforced политика.

Git атрибути

Някои от настройките на Git може да се прилагат към конкретен път, така че да важат за определена поддиректория или набор от файлове. Този вид настройки се наричат Git атрибути и се задават или във файл с име `.gitattributes` в една от директориите (основно

главната) или във файла `.git/info/attributes`.

Използвайки атрибути можете да правите неща като например отделни стратегии за сливане за индивидуални файлове или директории в проекта, да указвате на Git как да прави diff на файлове, които не са текстови, или да накарате Git да филтрира съдържание преди да ви го извлече или индексира. Тук ще разгледаме малко примери за използване на атрибути.

Двоични файлове

Един хубав трик, който можете да приложите, е да кажете на Git кои файлове са двоични (в случай, че по някаква причина не се разпознават като такива) и да подадете специални инструкции за тяхната обработка. Например, някои текстови файлове може да са машинно генерирани и да не може да им се направи diff, докато за някои други двоични файлове това е възможно. Ето как да кажете на Git кой какъв е.

Идентифициране на двоични файлове

Някои файлове изглеждат като текстови, но може да искате данните в тях се третират като двоични. Например, Xcode проектите в macOS съдържат файл с окончание `.pbxproj`, който всъщност е JSON (текстов формат за структуриране на данни) информация записана на диска от IDE средата и съхранява настройки за компилиране и други данни. Въпреки, че технически това е текстов файл (изцяло UTF-8), не искате да го третирате като такъв, понеже той играе ролята на олекотена база данни — не можете да слеете модификациите, ако двама души го променят и в общи линни diff информацията не е особено полезна. Файлът просто е предназначен да се обработва от машината. Реално, искате да го третирате като двоичен.

За да кажете на Git, че искате да третира всички `pbxproj` файлове като двоични, добавете това във вашия `.gitattributes` файл:

```
*.pbxproj binary
```

Сега Git няма да се опитва да конвертира или коригира CRLF проблеми, нито ще се опитва да изчислява или печата diff за промените в такива файлове при изпълнение на `git show` или `git diff`.

Diff за двоични файлове

Можете обаче да накарате Git да прави diff на двоични файлове. Това става, като указвате на системата да конвертира двоичната информация в текст, който да се съпоставя с нормална diff операция.

Ще използваме тази техника за решаване на един от най-досадните известни проблеми: контрол на версиите за Microsoft Word документи. За мнозина Word е един от най-ужасните редактори, но така или иначе продължава да се ползва масово. Ако искате да контролирате Word документи, можете да ги съберете в Git хранилище и да къмитвате от време на време, но каква полза получавате с това? Ако нормално изпълните `git diff`, виждате нещо такова:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Не можете директно да съпоставите две версии, докато не ги извлечете и разгледате на ръка. Оказва се, че можете да правите това добре с Git атрибути. Сложете следния ред в `.gitattributes` файла:

```
*.docx diff=word
```

Това указва на Git, че ако файлът отговаря на тази маска (`.docx`), то той трябва да използва филтъра "word", при опит за конструиране на diff. Какво е "word" филтър? Ще трябва да го настроите. В случая, ще използваме програмата `docx2txt` за да конвертираме Word документите в четими текстови файлове, за които могат да се правят четими diff-ове.

Първо инсталираме `docx2txt`; налична е на https://sourceforge.net/projects/docx2txt. Следваме инструкциите в `INSTALL` файла за да я сложим на място, в което шелът да я намира. След това, ще напишем кратък wrapper скрипт за конвертиране на изхода във формат, който Git очаква. Създайте файл достъпен в пътя ви с име `docx2txt`, съдържащ:

```
#!/bin/bash
docx2txt.pl "$1" -
```

След това изпълнете `chmod a+x` за него. Последно, конфигурираме Git да използва скрипта:

```
$ git config diff.word.textconv docx2txt
```

Сега Git знае, че ако се опитва да направи diff между два snapshot-а и някой от файловете е с окончание `.docx`, той трябва да го прекара през "word" филтъра, дефиниран като `docx2txt` програмата. Това на практика произвежда текстови версии на Word документите преди да опита да ги съпостави за diff.

Ето пример: Глава 1 от тази книга е конвертирана в Word формат и къмитната в Git хранилище. След това е добавен нов параграф. Ето изхода от `git diff`:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning
by explaining some background on version control tools, then move on to how to get Git
running on your system and finally how to get it setup to start working with. At the
end of this chapter you should understand why Git is around, why you should use it and
you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that
records changes to a file or set of files over time so that you can recall specific
versions later. For the examples in this book you will use software source code as the
files being version controlled, though in reality you can do this with nearly any type
of file on a computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or
layout (which you would most certainly want to), a Version Control System (VCS) is a
very wise thing to use. It allows you to revert files back to a previous state, revert
the entire project back to a previous state, compare changes over time, see who last
modified something that might be causing a problem, who introduced an issue and when,
and more. Using a VCS also generally means that if you screw things up or lose files,
you can easily recover. In addition, you get all this for very little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another
directory (perhaps a time-stamped directory, if they're clever). This approach is very
common because it is so simple, but it is also incredibly error prone. It is easy to
forget which directory you're in and accidentally write to the wrong file or copy over
files you don't mean to.
```

Git успешно забелязва промяната и сбито ни казва, че сме добавили стринга “Testing: 1, 2, 3.”. Разбира се, това не е перфектно — промените по форматирането няма да се покажат.

Друг интересен проблем е съпоставянето на файлове с изображения. Един начин да правите това е да прекарате изображение през филтър, който извлича EXIF информацията от него — това са метаданни съхранявани с повечето формати за изображения. Ако изтеглите и инсталирате програмата `exiftool`, може да я ползвате за да извличате метаданните, така че diff поне ще ви покаже в текстов вид направените промени. Сложете реда отдолу в `.gitattributes` файла:

```
*.png diff=exif
```

Настройте Git да използва инструмента:

```
$ git config diff.exif.textconv exiftool
```

Ако замените изображение в проекта и изпълните `git diff`, виждате нещо подобно:

```
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
```

Така поне можем да видим, че са променени размерите на изображението и големината на файла.

Попълване на ключови думи

SVN- или CVS-style попълването на ключови думи често се използва от разработчиците използващи тези системи. Основният проблем с това в Git е, че не можете да вмъкнете файл с информация за къмита след като сте къмитнали, защото Git първо изчислява чексума за файла. Обаче, можете да вмъквате текст във файл, когато файлът е извлечен и да го премахнете преди добавянето в къмит. Git атрибутите предлагат два начина за това.

Първо, можете да вмъкнете SHA-1 чексумата на обект в `$Id$` поле във файла автоматично. Ако зададете този атрибут за файл или множество файлове, тогава следващия път когато извлечете съдържанието на такъв клон Git ще замени това поле с SHA-1 стойността на обекта. Важно е да се помни, че това не е SHA-1 на къмита, а на самия blob обект. Сложете такъв ред в `.gitattributes`:

```
*.txt ident
```

Добавете `$Id$` референция в пробен файл:

```
$ echo '$Id$' > test.txt
```

Следващия път, когато извлечете този файл, Git вмъква SHA-1 на blob обекта:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Обаче, това е с ограничена полза. Ако сте използвали заместване на ключови думи в CVS или Subversion, можете да вмъквате datestamp — SHA-1 стойността не е толкова полезна, защото е случайна и само гледайки я не можете да кажете дали е по-стара или по-нова от друга такава.

Оказва се, че можете да пишете свои собствени филтри за правене на замествания във файлове по време на къмитване или извличане. Тези фитри са известни като “clean” и “smudge” филтри. Във файла `.gitattributes`, можете да настроите филтър за определени пътища и след това да направите скриптове, които да обработват файловете непосредствено преди да се извлекат (“smudge”, вижте Филтърът “smudge” се изпълнява по време на извличане) и непосредствено преди да се индексират (“clean”, вижте Филтърът “clean” се изпълнява по време на индексиране на файлове). С тези филтри могат да се правят най-различни забавни неща.

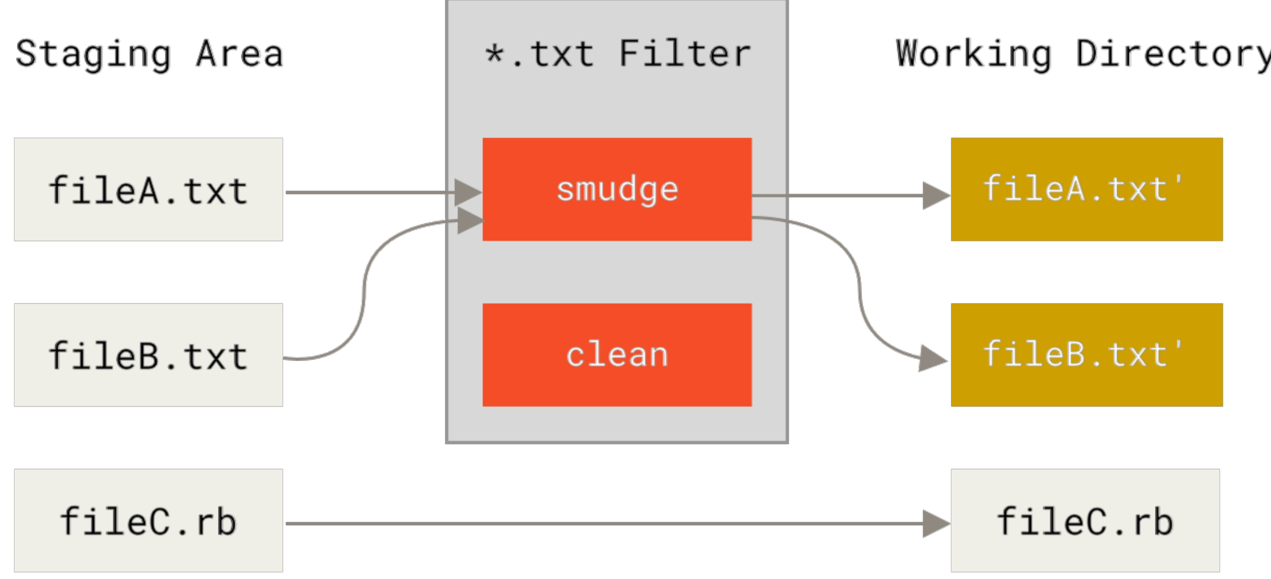


Figure 143. Филтърът “smudge” се изпълнява по време на извличане

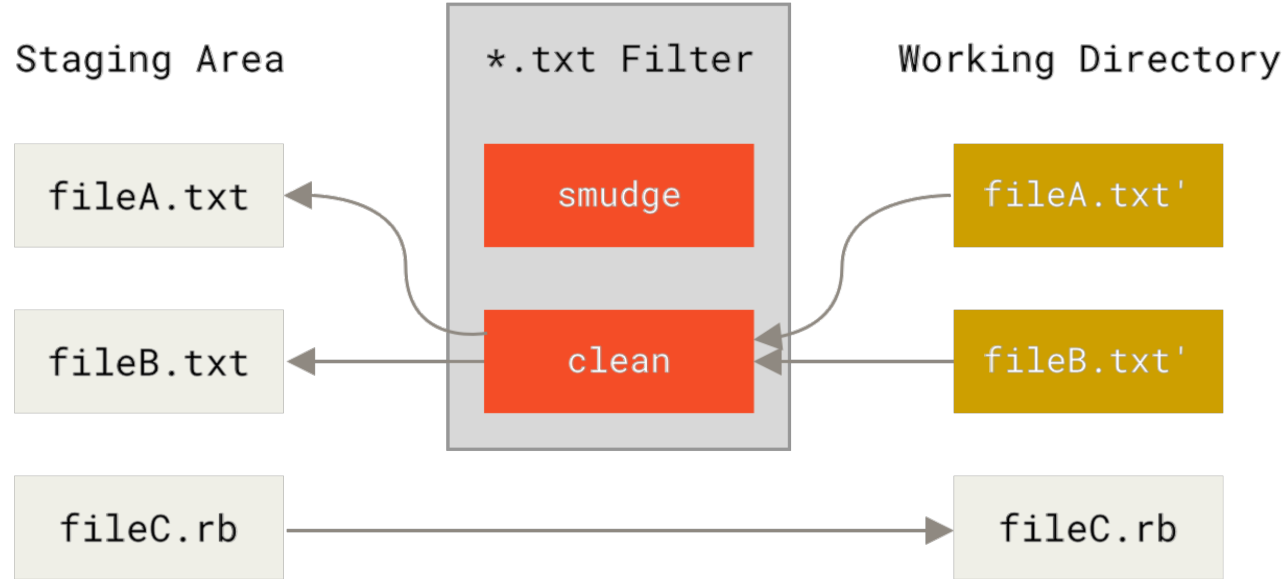


Figure 144. Филтърът “clean” се изпълнява по време на индексиране на файлове

Оригиналното къмит съобщение за тази функция дава прост пример за вмъкване на отстъпи в C сорс код преди къмитване. Можете да направите такъв филтър с атрибут в `.gitattributes` файла за да обработите всички `*.c` файлове с филтъра “indent”:

```
*.c filter=indent
```

След това, казваме на Git какво ще прави филтъра “indent” при операциите smudge и clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

В този случай, когато къмитнете файлове с окончание `*.c`, Git ще ги прекара през програмата indent преди да ги индексира и също така ще ги прекара през `cat` програмата, когато ги извлича обратно на диска. Програмата `cat` по същество не върши нищо освен да печата на екрана данните, които е получила. Ефективно тази комбинация филтрира всички C сорс файлове през `indent` преди къмитване.

Друг интересен пример е с `$Date$` keyword expansion израза, RCS стил. За да направите това правилно, нуждаете се от малък скрипт, който взема име на файл, намира последната дата на къмит за този проект и я вмъква във файла. Ето малък Ruby скрипт, който прави това:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Скриптът взема датата на последния къмит от командата `git log`, прикрепя я към всеки `$Date$` стринг, който вижда на stdin и печата резултатите — би трябвало да може лесно да се

направи с произволен език за програмиране. Можете да дадете на скрипта име `expand_date` и да го вмъкнете в пътя си. Следващата стъпка е да направите филтър в Git (наречен `dater`) и да му укажете да използва вашия `expand_date` скрипт по време на checkout на файлове. Ще използваме регулярен израз на Perl за clean операцията по време на къмитване:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Тази отрязък код премахва всичко, което намери в `$Date$` стринг за да получите обратно данните си. Сега можем да тестваме филтъра създавайки Git атрибут, който активира филтъра и файл с ключова дума `$Date$`:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Ако сега къмитнете тези промени и извлечете файла отново, ще видите ключовата дума заместена със съответната дата:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Test date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Можете да видите колко мощна може да е тази техника за специализирани приложения. Все пак трябва да се внимава, защото файлът `.gitattributes` се къмитва и разпраща заедно с проекта, но не и съответния филтър (в този случай `dater`), който на друга машина няма да работи. Проектирайки тези филтри, те трябва да могат да правят fail gracefully, така че проектът да работи правилно.

Експорт на хранилище

Атрибутите на Git позволяват да правите и други интересни неща, когато архивирате или експортирате проект.

`export-ignore`

Можете да кажете на Git да не експортира определени файлове или директории, когато генерира архив. Ако има поддиректория или файл, които не искате да влизат в архива, но да присъстват в самия проект, можете да ги укажете чрез атрибута `export-ignore`.

Например, имате тестови файлове в поддиректория `test/` и не желаете да влизат в архива на проекта. Можете да зададете това със следния ред във вашия Git attributes файл:

```
test/ export-ignore
```

Сега, когато изпълните `git archive`, за да създадете tarball архив на проекта, тази директория няма да влиза в архива.

`export-subst`

Когато експортирате файлове за внедряване, можете да прилагате правилата за форматиране и разширяване на ключови думи на `git log` към избрани части от файлове маркирани с атрибута `export-subst`.

Например, ако искате да вмъкнете файл с име `LAST_COMMIT` и съдържащ метаданните от последния къмит в проекта си автоматично, когато изпълните `git archive`, можете да го направите със следните `.gitattributes` и `LAST_COMMIT` файлове:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Когато пуснете `git archive`, съдържанието на архивния файл ще изглежда така:

```
$ 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
```

Заместванията могат например да включват къмит съобщението и наличните `git notes` и `git log` може да прави 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.
```

Полученият архив е подходящ за внедряване, но подобно на всеки експортиран архив, не е

подходящ за бъдеща програмистка работа.

Стратегии за сливане

Можете също така да изполвате Git атрибути за да укажете на Git да използва различни merge стратегии за специфични файлове в проекта. Една много полезна възможност е например да укажете, че не искате Git да опитва да слива определени файлове, когато при тях има конфликти, а вместо това директно да използва вашата версия на тези файлове с предимство.

Това е полезно, ако даден клон от проекта ви се е разклонил или е специализиран такъв, но искате да можете да сливате промените от него и да игнорирате определени файлове. Да кажем, че имате файл `database.xml` с конфигурация за достъп до база данни, който е различен за два клона и искате да слеете другия клон без да намесвате този файл в сливането. Можете да направите такъв атрибут:

```
database.xml merge=ours
```

След което създавате лъжлива merge стратегия `ours`:

```
$ git config --global merge.ours.driver true
```

Ако сега слеете другия клон, вместо да получите конфликт при сливането за `database.xml`, ще видите нещо такова:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

В този случай `database.xml` си остава в оригиналната версия.

Git Hooks

Както повечето VCS системи, Git разполага с механизъм да изпълнява потребителски скриптове при възникване на определни събития. Тези скриптове са известни като Hooks (обработчици на събития) и се разделят на две групи: client-side и server-side. Клиентските hooks се пускат при операции като къмитване или сливане, докато сървърните отразяват събития от мрежови операции като например получаване на публикувани къмити. Можете да използвате hooks за всякакви неща.

Инсталиране на Hook

Всички hooks се съхраняват в `hooks` поддиректория в Git директорията. В повечето проекти това е `.git/hooks`. Когато инициализирате ново хранилище с `git init`, Git попълва въпросната директория с множество примерни скриптове, които освен че са полезни сами

по себе си, също така и документират входните стойности за всеки скрипт. Всички примери са написани като шел скриптове с малко Perl, но всъщност кои да е коректно именувани и изпълними скриптове ще работят добре — можете да ги пишете на Ruby или Python или който език предпочитате. Ако искате да използвате фабрично доставените с Git скриптове, ще трябва да преименувате файловете, те завършват на `.sample`.

За да разрешите hook скрипт, поставете коректно именуван (без разширение) и изпълним файл в `hooks` поддиректорията на .git. От този момент натам, той би трябвало да се изпълнява. Ще разгледаме няколко от основните имена за hooks файлове.

Клиентски Hooks

Налични са много client-side hooks. Тази секция ги разделя в категориите committing-workflow hooks, email-workflow скриптове и всичко останало.

Важно е да споменем, че клиентските hooks **не се копират**, когато клонирате хранилище. Ако целта на скриптовете ви е да наложите дадена политика на работа, вероятно ще искате да правите това на сървъра, вижте примера в Примерна Git-Enforced политика.

Committing-Workflow Hooks

Първите четири hooks се занимават с процеса на къмитване.

Скриптът `pre-commit` се пуска първи, още преди да напишете къмит съобщението. Той се използва за проверка на snapshot-a, който ще бъде къмитнат и следи дали не сте забравили нещо, дали се изпълняват тестове и т.н. Ако изходният код от този hook не е нулев (успешно завършен), то къмитът ще бъде отказан, въпреки че това е преодолимо с `git commit --no-verify`. Можете да правите неща като проверка на стила на писане на код (чрез изпълнение на `lint` или нещо от рода), проверка за празни символи в края на редовете (подразбиращият се hook върши именно това), или проверка за подходяща документация на нови методи в кода.

Следва `prepare-commit-msg` hook-ът, който се стартира преди текстовия редактор, но след създаването на къмит съобщението по подразбиране. Това ви позволява да редактирате съобщението преди автора на къмита да го види на екрана. Този hook има няколко аргумента: пътя на файла, който съхранява къмит съобщението, типа на къмита и SHA-1 стойността, ако това е amended къмит. Това не е кой знае колко полезен Hook за нормални къмити, вместо това е приложим за такива, в които съобщението по подразбиране се генерира автоматично — като templated къмит съобщенията, сливащите къмити, а също и squashed и amended къмитите. Можете да го използвате в комбинация с commit template за да вмъквате информация програмно.

Hook-ът `commit-msg` приема един параметър, който отново е път към временен файл, съдържащ къмит съобщението от разработчика. Ако скриптът завърши неуспешно Git отказва къмит процеса, така че можете да го използвате за валидиране на статуса на проекта или на самото къмит съобщение преди да позволите на къмита да мине. В последната част от главата ще видим как да ползваме този hook за да проверим дали подадено къмит съобщение отговаря на определени правила.

След като целият къмит процес завърши се стартира `post-commit` hook-a. Той не приема параметри, но можете лесно да видите последния къмит с `git log -1 HEAD`. Общо взето, този hook се използва за нотификации или подобни процеси.

Email Workflow Hooks

Можете да настроите три client-side hooks за имейл базирани работни процеси. Всички те се стартират през командата `git am`, така че ако не я използвате, може да прескочите направо към следващата част в главата. Ако обаче получавате пачове през имейл, изготвени с `git format-patch`, тогава някой от тези скриптове може да ви е от полза.

Първият изпълнен е `applypatch-msg`. Той приема един аргумент: името на временния файл с къмит съобщението. Git отказва пача, ако скриптът завърши с код за неуспех. Бихте могли да го използвате за да се уверите, че къмит съобщението е правилно форматирано или да го нормализирате с допълнителна редакция.

Следващият hook, който се изпълнява при прилагане на пачове с `git am` е `pre-applypatch`. Противно на очакваното предвид името му, той се изпълнява *след* прилагането на пача, но преди да е направен къмит, така че би могъл да се използва за инспектиране на snapshot-а преди къмитване. С този скрипт можете да пускате тестове в работната директория. Ако нещо липсва или тестовете не минават, кодът за неуспех прекратява `git am` и пачът не се прилага.

Последният hook касаещ `git am` операцията е `post-applypatch`, който се пуска след къмитване. Можете да го използвате за да нотифицирате група хора или автора на пача, че сте го приложили. С този скрипт не можете да спрете patching процеса.

Други Client Hooks

`pre-rebase` hook-ът се изпълнява преди пребазирането на каквото и да било и може да прекрати процеса при код за неуспех. Може да се използва например за забрана на пребазиране на публикувани в отдалечени хранилища къмити. Примерният `pre-rebase` hook, който Git инсталира прави това, въпреки че използва някои презумпции, които може да не отговарят на конкретния ви работен процес.

Hook-ът `post-rewrite` се изпълнява от команди, които заместват къмити като `git commit --amend` и `git rebase` (макар и не от `git filter-branch`). Единичният параметър, който получава, е командата направила заместването ведно със списък от промените на `stdin`. Този hook може да се ползва за повечето неща, за които могат и `post-checkout` и `post-merge`.

След успешно изпълнение на команда `git checkout`, `post-checkout` hook-ът се стартира и бихте могли да го използвате за настройка на работната директория за специфичните изисквания на проекта например. Това може да означава преместване на големи двоични файлове, които не искате да проследявате, автоматично генериране на документация и т.н.

Hook-ът `post-merge` се пуска след успешна `merge` команда. Може да се използва за възстановяване на информация, която Git не проследява — като например права върху файлове. Също така би могъл да се използва за проверка за наличето на външни за Git файлове, които бихте желали да се копират в проекта, когато работната директория се промени.

Hook-ът `pre-push` работи по време на `git push`, след като отдалечените референции се обновят, но преди трансфера на каквито и да било обекти. Той получава в параметри името и адреса на отдалечените хранилища и списъка на референциите, които ще бъдат обновени през `stdin`. Може да се използва за валидация на набор от промени преди push операцията (код за неуспех на скрипта отменя публикуването).

Git редовно извършва garbage collection действия като част от нормалната си работа извиквайки `git gc --auto`. Съответно, hook-ът `pre-auto-gc` се изпълнява преди процеса по почистването и може да ви известява, че то предстои да се направи или пък да го отмени, ако моментът не е подходящ за това.

Сървърни Hooks

В допълнение към клиентските hooks, ако сте системен администратор, можете да използвате множество важни сървърни такива за да прилагате почти всеки вид политика за вашия проект. Тези скриптове се изпълняват преди и след публикувания на промени в сървъра. Тези, които работят преди приемане на публикуването могат да върнат код за неуспех по всяко време и да го откажат, както и да отпечатат съобщение за грешка към клиента. Така че можете да си създадете push политика колкото сложна, колкото ви е нужно.

`pre-receive`

Първият изпълняван скрипт по време на обработка на push от клиент е `pre-receive`. Той приема от stdin списък на референциите, които се публикуват и ако излезе с код за неуспех никоя от тях не се приема. Може да ползвате този hook за да проверявате, че никоя от променените референции не е non-fast-forward или да правите контрол на достъп за всички референции и файлове, които ще бъдат променени от публикуването.

`update`

Скриптът `update` е много подобен на `pre-receive`, разликата е, че се изпълнява по веднъж за всеки клон, който публикуващият се опитва да обнови. Ако се обновяват няколко клона, `pre-receive` стартира само веднъж, докато update върви за всеки клон. Вместо да чете от стандартния вход, този скрипт приема три параметъра: името на референцията (клона), SHA-1 стойността, към която референцията е сочела преди push операцията, и SHA-1 стойността, която потребителят опитва да изпрати. Ако update скриптът завърши с код за неуспех, то само текущата референция се отказва, другите все още могат да бъдат обновени.

`post-receive`

Накрая, `post-receive` hook-ът се изпълнява след края на целия процес и може да се използва за обновяване на други услуги или нотифициране на потребители. Той получава същите stdin данни както и `pre-receive`. Сред примерните негови приложения биха могли да са разпращане на имейли до потребители, нотифициране на continuous integration сървър или ъпдейт на ticket-tracking система - можете дори да парсвате къмит съобщения, за да проверявате дали дадени тикети трябва да се отворят, редактират или затворят. Скриптът не може да спре push процеса, но връзката с клиента не се прекъсва докато той не завърши, така че бъдете внимателни, ако се опитвате да правите с него неща отнемащи прекалено дълго време.

Ако пишете скрипт/hook, който останалите трябва да четат, стремете се да използвате дългите версии на флаговете за командния ред. Само половин година по-късно ще ни благодарите.

Примерна Git-Enforced политика

В тази секция ще видим как да изградим примерен Git работен процес, който проверява за специфичен формат на къмит съобщенията и позволява само на определени потребители да модифицират определени поддиректории в проекта. Ще напишем клиентски скриптове, които помагат на разработчика да разбере дали публикуванията му ще се отхвърлят и сървърни скриптове, които в действителност прилагат политиките.

Примерните скриптове са на Ruby, отчасти поради факта, че използвахме езика в примерите дотук, но и също така защото Ruby е лесен за четене дори да не можете да програмирате на него. Обаче, всеки език би следвало да е приложим — всички примерни hook скриптове фабрично доставени с Git са или на Perl или на Bash.

Сървърен Hook

Цялата работа от страна на сървъра ще се върши от файла `update` в директорията `hooks`. Този `update` hook се стартира еднократно за всеки клон, в който се публикува и получава три аргумента:

- Името на клона в който се прави push
- Старата версия, в която е бил клона
- Новата версия, която се публикува

Имате също така достъп до потребителя, който публикува, ако това се прави през SSH. Ако сте позволили на всеки да се свързва с едно и също потребителско име (като “git”) с public-key автентикация, може да трябва да дадете на въпросния потребителски акаунт shell wrapper, който определя реалния потребител чрез информацията от публичния ключ и съответно пази резултата в environment променлива. За нашия случай приемаме, че свързващият се реален потребител се пази в променливата `$USER`, така че update скриптът започва търсейки това:

```
#!/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]})"
```

Да, това са глобални променливи. Не е правилно, но е по-лесно за демонстрация.

Налагане на специфичен Commit-Message формат

Първата ни задача е да задължим потребителя да следва специфичен зададен от нас формат за всяко къмит съобщение. Приемаме че всяко съобщение трябва да включва стринг, който изглежда като "ref: 1234", защото искате всеки къмит да препраща към елемент от вашата ticketing система. Трябва да претърсвате всеки изпратен къмит, да проверявате дали стрингът присъства в съобщението и ако липсва дори в едно от всички, да подадете неуспешен код за изход от скрипта, така че push операцията да бъде отказана.

Можете да получите списък на всички SHA-1 стойности за къмитите, които се публикуват като подадете стойностите в `$newrev` и `$oldrev` към plumbing командата на Git известна като `git rev-list`. Това е по същество `git log` командата, но по подразбиране печата само SHA-1 стойности и никаква друга информация. И така, за да получите списък на всички чексуми на къмити появили се между два дадени къмита, можете да изпълните нещо такова:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Можете да ползвате този резултат, да преминете с цикъл през всяка от SHA-1 стойностите, да извлечете съобщението за всеки къмит и да го тествате с регулярен израз дали отговаря на дадения шаблон.

Сега трябва да откриете как да получите къмит съобщението за всеки от къмитите. За да получите суровите данни за къмит, може да използвате друга plumbing команда, `git cat-file`. Ще разгледаме в подробности plumbing командите в Git на ниско ниво, но засега ето какво ви дава тази конкретно:

```
$ 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

Change the version number
```

Прост начин да извлечете съобщението от данните за къмит, когато имате SHA-1 стойността, е да отидете на първия празен ред от данните и да вземете всичко след това. Това може да се направи с командата `sed` под Unix системи:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
Change the version number
```

Може да приложите това за всеки къмит, който се опитва да бъде публикуван и да излезете

със съответния код, ако нещо не е както се очаква да бъде. За да прекратите скрипта и отхвърлите push операцията, подайте non-zero код за изход от него. Целият метод изглежда така:

```
$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
```

Така изглеждащ, вашият `update` скрипт ще отхвърля всички публикувания, чиито къмит съобщения не съответстват на зададеното от вас правило.

Прилагане на User-Based ACL система

Представете си, че искате да добавите механизъм, който използва access control list (ACL) и определящ кои потребители до кои части от проектите ви могат да публикуват. Искате някои да имат пълен достъп, а други само до определени поддиректории или специфични файлове. За да реализираме това, ще попълним правилата във файл с име `acl`, който се съхранява в bare хранилището на Git сървъра. Нашият `update` hook ще следи за тези правила, ще определи кои файлове ще се променят във всички изпратени къмити и ще реши дали потребителят, който опитва обновяването ще има достъп до тях.

Първо, ще попълним нашия ACL. Тук използваме формат подобен на CVS ACL механизма: той използва серия от редове, където първото поле е `avail` или `unavail`, следващото поле е разделен със запетаи списък на потребителите, за които ще важат правилата и последното поле е пътя, към който се прилага правилото (празната стойност тук означава пълен достъп). Всички тези полета се разделят със символа pipe (`|`).

В този случай, имате няколко администратора, имате хора пишещи документация с достъп до директорията `doc` и един разработчик с достъп само до директориите `lib` и `tests`, така че ACL файлът изглежда така:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Започваме прочитайки тези данни в използваема структура. За да е опростен примерът, ще

използваме само `avail` директиви. Ето един метод да получите асоциативен масив, в който ключовете на елементите са имената на потребителите, а стойностите им — масив с пътищата до които потребителят има достъп:

```
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
```

Предвид ACL файла, който имаме, този `get_acl_access_data` метод ще ни върне структура от данни изглеждаща така:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Сега имаме правилата за достъп и следва да определим какви са пътищата, които публикуваните къмити опитват да променят и дали съответния потребител има достъп до всички тях.

Можете лесно да видите кои са файловете засегнати от един къмит с флага `--name-only` на командата `git log` (упоменато в Основи на Git):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Ако използвате ACL структурата върната от метода `get_acl_access_data` и я сравните със списъка от засегнати файлове за всеки от къмитите, може да определите дали потребителят ще има push достъп за всеки от къмитите си:

```
# 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
```

Списък на новите къмити изпратени към сървъра получавате с `git rev-list`. След това, за всеки от тези къмити определяте кои пътища ще се променят и дали съответния потребител има права за това.

Сега потребителите на сървъра не могат да публикуват къмити с лошо форматирани съобщения или засягащи файлове, до които нямат достъп.

Тестване на механизма

Ако изпълните `chmod u+x .git/hooks/update` (файла в който се съдържа всичкия ни Ruby код) и след това опитате да публикувате къмит с неотговарящо на правилата съобщение, получавате нещо подобно:

```
$ 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'
```

Тук има няколко интересни неща. Първо, виждате следното съобщение, когато hook-ът стартира:

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Спомнете си, че отпечатахте това в самото начало на update скрипта. Всичко, което скриптът изпраща в `stdout` ще бъде изпратено към клиента.

Следващото нещо е съобщението за грешка.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Първия ред печатате вие, следващите два идват от Git, който уведомява че update скриптът е завършил с код за неуспех и че публикуването е отказано. Последно, имаме това:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Ще видим remote rejected съобщение за всяка от референциите, които hook-ът е отхвърлил и то изрично упоменава, че отхвърлянето идва именно от неуспешно завършил hook.

Аналогично, ако някой се опита да промени файл, за който няма права за достъп, ще види нещо подобно. Например, ако автор на документация се опита да публикува къмит модифициращ съдържанието на `lib` директорията, то той ще види

```
[POLICY] You do not have access to push to lib/test.rb
```

Отсега нататък, стига `update` скриптът да е наличен и изпълним, вашето хранилище няма да приема къмити, чиито съобщения не съдържат вашата задължителна част в себе си и също така всеки потребител ще може да публикува само там, където трябва да може.

Клиентски Hooks

Недостатъкът на този подход са неизбежните оплаквания, които трябва да очаквате от колегите ви, чиито публикувания са били отхвърлени. Да се окажат в положение, при което внимателно свършената им работа бива отхвърлена в последния момент може да бъде изключително конфузно и изнервящо, защото те ще трябва да редактират историята си, за да отговорят на правилата — нещо на което никой няма да се зарадва.

Отговорът на този проблем е да осигурите клиентски hook скриптове, които потребителите могат да използват за да разберат дали нещо от работата им евентуално би могло да бъде отхвърлено на сървъра. По този начин те могат да коригират всички проблеми преди къмитване и преди да станат трудни за оправяне. Понеже hook скриптовете не се разпространяват когато клонирате проект, ще трябва да ги изпратите по някакъв друг начин и да се уверите, че потребителите са ги копирали в своите `.git/hooks` директории и че са ги направили изпълними. Можете да ги изпращате като част от проекта или като отделен проект, но Git няма да ги настрои автоматично.

Ако се поставите в ролята на вашите потребители, отначало трябва да проверите дали къмит съобщението ви отговаря на правилата още преди да направите къмита. За целта, може да използвате `commit-msg` hook-а. Ако го направите така, че да чете съобщението от файл, който му се подава като първи аргумент и сравните това съобщение с даден шаблон, може да накарате Git да откаже къмита, ако има несъответствие:

```
#!/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
```

Ако този скрипт е на правилното място (в `.git/hooks/commit-msg`), изпълним е, и опитате да къмитнете със съобщение, което е некоректно форматирано, ще получите този резултат:

```
$ git commit -am 'Test'
[POLICY] Your message is not formatted correctly
```

Къмитът е отказан. Ако съобщението е OK, тогава Git извършва операцията:

```
$ git commit -am 'Test [ref: 132]'
[master e05c914] Test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

След това искате да се уверите, че не променяте файлове, които са извън обхвата на пътищата определени за вас от ACL системата. Ако `.git` директорията на проекта ви съдържа копие от ACL файла, който използвахме преди малко, тогава следния `pre-commit` скрипт ще направи проверката локално за вас:

```
#!/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
```

Това е почти същия скрипт като този на сървъра, но с две важни разлики. Първо, ACL файлът е на различно място, защото този скрипт се изпълнява от работната ви директория, а не от `.git` директорията. Затова трябва да смените пътя до него от това:

```
access = get_acl_access_data('acl')
```

към това:

```
access = get_acl_access_data('.git/acl')
```

Другата важна разлика е начинът, по който получавате списъка на променените файлове. Сървърният метод търси за тях в лога на къмитите, докато локалния в този момент не може, защото къмитът не е записан. Ето защо трябва да извлечете списъка от индексната област. Вместо:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

трябва да използвате командата:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Изключая тези две разлики, скриптът работи по същия начин. Един от недостатъците му е, че очаква да бъде изпълнен локално от същия потребител като този, който публикува на сървъра. Ако това не е така, ще трябва също да коригирате и `$user` променливата ръчно.

Друго нещо, което можем да направим тук е да се уверим, че потребителят не се опитва да публикува non-fast-forward референции. За да получим подобна референция, можем да пребазираме последния публикуван къмит или да се опитаме да публикуваме различен локален клон към оригиналния отдалечен такъв.

Приемаме, че сървърът вече е конфигуриран с `receive.denyDeletes` и `receive.denyNonFastForwards` за да форсира тази политика, така че единственото нещо, което може да се опитате да засечете е пребазирането на вече публикувани къмити.

Ето един примерен pre-rebase скрипт, който следи за това. Той получава списък от всички къмити, които ще пренапишете и проверява дали съществуват в коя да е от отдалечените референции. Ако се установи един такъв, който е достъпен през някоя от отдалечените референции, пребазирането се отказва.

```
#!/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
```

Този скрипт използва синтаксис, който не разгледахме в Избор на къмити. Получавате списък на публикуваните къмити с това:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

Синтаксисът `SHA^@` отговаря на всички родители на този къмит. Търсите за всеки къмит, който е достъпен от последния отдалечен къмит и който не е достъпен от кой да е родител от всички SHA-1 стойности, които се опитвате да публикувате — това означава че е fast-forward.

Основният недостатък на този подход е, че той може да е много бавен и често ненужен — ако не се опитвате да форсирате push операцията с `-f`, то сървърът така или иначе ще ви предупреди и няма да я приеме. Все пак, това е интересно упражнение и на теория може да ви помогне да избегнете пребазирания, които по-късно да се налага да коригирате.

Обобщение

Разгледахме повечето основни начини, които са на ваше разположение за фина настройка на Git клиента/сървъра, така че да отговарят най-добре на вашите проекти и похвати за работа. Научихте много конфигурационни настройки, файлово-базирани атрибути, event hooks и дори създадохте примерен policy-enforcing сървър. Сега би следвало да можете да накарате Git да се вписва в почти всеки работен сценарий, който може да си представите.

Git и други системи

Светът не е перфектен. Обикновено не можете безпрепятствено да прехвърлите всеки проект, който имате към Git. Понякога имате проект, който ползва друга VCS, и бихте искали да е Git. Ще отделим първата част на тази глава за да научим начините за използване на Git като клиент, когато проектът, по който работите е на друга система.

На даден етап може да решите да конвертирате съществуващ проект към Git. Втората част на главата е посветена на това — разглежда начините да мигрирате проекти към Git от няколко други конкретни системи а също и метод, който ще работи ако не съществува импортиращ инструмент.

Git като клиент

Git е толкова полезен, че много разработчици са намерили начин да го използват на работните си станции макар останалите им колеги от екипа да ползват съвсем различни VCS. Съществуват много налични адаптери за това, известни като “bridges”. Тук ще посочим някои от тях, с които е вероятно да си имате работа.

Git и Subversion

Голяма част от проектите с отворен код, а също така и голям брой корпоративни проекти използват Subversion за управление на сорс кода. Subversion съществува повече от десетилетие и през повечето време от своя живот беше *де факто* първи избор за контрол на open source проекти. Тя много прилича на CVS, която пък преди това беше най-популярната система за контрол.

Една от най-съществените възможности на Git е двупосочния bridge към Subversion известен като `git svn`. Този инструмент позволява да използвате Git като валиден клиент на Subversion сървър, така че можете да ползвате всички локални възможности на Git и след това да публикувате работатата си на Subversion сървъра така сякаш ползвате Subversion и локално. Това означава, че можете да имате локални клонове и сливания, да използвате индексната област, да правите rebasing/cherry-picking и т.н. докато колегите ви продължават да си работят по стария начин. Това е добър начин за вмъкване на Git в корпоративни среди както и за привличане на повече разработчици (а защо не и на цели корпоративни звена) да преминат на Git. Subversion бриджът е ключ към DVCS света.

`git svn`

Основната команда в Git за всички Subversion функционалности е `git svn`. Тя приема множество подкоманди и ще покажем най-използваните от тях симулирайки няколко прости работни процеса.

Важно е да помним, че използвайки `git svn` ние комуникираме със Subversion, система функционираща по много различен начин от Git. Въпреки че **можете** да правите локални клонове и сливания, в общи линии е най-добре да поддържате историята си възможно най-праволинейна използвайки пребазирания и избягвайки да правите неща като едновременна комуникация с отдалечени Git хранилища.

Не опитвайте да пренаписвате историята и да правите повторно публикуване и не публикувайте в паралелно Git хранилище за да сътрудничите с други Git колеги по едно и също време. Subversion може да има само една, линейна история и е много лесно да бъде объркана. Ако работите с екип и някои от вас използват SVN, а други Git — уверете се, че всички разработчици използват SVN сървъра за споделяне на работа, това ще ви улесни живота значително.

Настройка

За целите на демонстрацията, се нуждаем от стандартно SVN хранилище с права за писане в него. Ако искате да копирате следващите примери, ще трябва да направите writeable копие на тестово Subversion хранилище. Най-лесно ще направите това с инструмента `svnsync`, който се разпространява със Subversion.

Първо, създаваме ново локално Subversion хранилище:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

След това, позволете всички да могат да променят revprops — лесният начин за това е да добавите скрипт `pre-revprop-change`, който винаги излиза с exit 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Сега можете да синхронизирате този проект в локалната машина изпълнявайки `svnsync init` с целево и изходно хранилища.

```
$ svnsync init file:///tmp/test-svn \
  http://your-svn-server.example.org/svn/
```

Това настройва параметрите за стартиране на синхронизацията. След това може да клонирате кода с:

```
$ 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.
[…]
```

Въпреки, че тази операция може да отнеме само няколко минути, ако се опитате да

копирате оригиналното хранилище към друго отдалечено такова (вместо към локално), процесът ще продължи близо час при все че има по-малко от 100 къмита. Subversion ще трябва да клонира една версия в един момент и след това да я публикува обратно в друго хранилище — това е тотално неефективно, но е единственият наличен начин.

Начало на работа

След като вече имате Subversion хранилище, до което имате права за писане, можете да изпълните един типичен работен процес. Ще започнем с командата `git svn clone`, която импортира цяло Subversion хранилище в локално Git такова. Да не забравяме, че импортираме от реално хостнато Subversion хранилище, така че трябва да заменим израза `file:///tmp/test-svn` с URL-а на вашето Subversion хранилище:

```
$ 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/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
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
```

Това всъщност изпълнява еквивалента на две команди, `git svn init` последвана от `git svn fetch` към URL-а посочен от вас. Процесът може да отнеме доста време. Ако например тестовият проект имаше само около 75 къмита и сорс кодът не е много голям, Git независимо от всичко ще трябва да извлече всяка една версия индивидуално и да я къмитне индивидуално всеки път. За проект със стотици или хиляди къмити, това буквално може да отнеме часове или дори дни.

Частта от командата `-T trunk -b branches -t tags` указва на Git, че това Subversion хранилище следва основните branching и tagging конвенции. Ако вие именувате вашия trunk, клонове или тагове по различен начин, можете да промените тези опции. Понеже това е толкова често ползвано, можете да замените целия израз просто с флага `-s`, което означава standart layout и прилага трите горни опции. Така че тази команда е еквивалентна:

```
$ git svn clone file:///tmp/test-svn -s
```

В този момент вече трябва да разполагате с валидно Git хранилище с импортирани тагове и клонове:

```
$ 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
```

Отбележете как инструментът управлява Subversion таговете като отдалечени референции. Нека погледнем в повече дълбочина с plumbing командата на Git `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 не прави това при клониране от Git сървър, ето как изглежда прясно клонирано хранилище с тагове:

```
$ 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 издърпва таговете директно в `refs/tags`, вместо да ги третира като отдалечени клонове.

Къмитване обратно в Subversion

След като имаме работна директория, можем да извършим някакви промени по нея и да публикуваме къмитите си обратно използвайки Git практически като SVN клиент. Ако променим един от файловете и къмитнем промяната, ще имаме къмит съществуващ локално, но не и на Subversion сървъра:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Следващата стъпка е да публикуваме тази промяна. Забележете как това променя начина ви на работа със Subversion — можете да направите повече от един локален къмит офлайн и след това да изпратите всички наведнъж към Subversion сървъра. За да направите това, използвайте командата `git svn dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Командата взема всички ваши новоизвършени къмити, създава Subversion къмит за всеки от тях и след това редактира локалния ви Git къмит за да включи уникален идентификатор. Това е важно, защото означава, че всичките ви SHA-1 чексуми на вашите къмити се променят. Отчасти поради тази причина, да работите с Git базирани отдалечени версии на проектите едновременно със Subversion такива не е добра идея. Ако погледнете последния къмит, може да видите новодобавения идентификатор `git-svn-id`:

```
$ 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
```

Забелязваме също така, че SHA-1 чексумата, която първоначално започваше с `4af61fd`, когато къмитнахме сега започва с `95e0222`. Ето защо, ако все пак настоявате да публикувате и към двата вида сървъри, трябва първо да направите това (`dcommit`) към Subversion сървъра, понеже това променя чексумата.

Издърпване на нови промени

При съвместната работа в един момент неизбежно се стига до опит за публикуване на промени предизвикващи конфликт. Промяната с конфликт ще бъде отхвърлена докато не слеете работата на другия колега преди това. В `git svn`, това изглежда така:

```
$ 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 rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M  README.txt
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.
```

За да се измъкнем от подобна ситуация, използваме `git svn rebase`, която издърпва от сървъра всички промени, които все още нямаме локално и след това пребазира текущата ни работа върху тях:

```
$ 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 rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M  README.txt
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.
```

Сега всичката ви работа е пребазирана върху последната изтеглена от Subversion сървъра, така че можете успешно да направите `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
```

Отбележете, че за разлика от Git, при който трябва да слеете upstream промените, които ви липсват локално преди да можете да публикувате, `git svn` ви кара да правите това само ако промените предизвикват конфликт (точно както работи Subversion). Казано с други думи, ако някой друг публикува промени по един файл и след това вие публикувате промени по друг файл, `dcommit` ще си работи без проблем:

```
$ 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,
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M  autogen.sh
First, rewinding head to replay your work on top of it...
```

Това е важен за запомняне момент, защото резултатът ще е статус на проекта, който не съществува и на двата локални компютъра. Ако промените са несъвместими (но не правят конфликт) може да се окажете с проблеми, които са трудни за установяване. При един Git сървър това не е така — можете изцяло да тествате статуса на проекта на локалната ви машина преди да го публикувате, докато при Subversion дори не можете да сте сигурни, че статусите преди и след къмитването ви са идентични.

Трябва да използвате тази команда за изтегляне на промени от Subversion сървъра дори все още да не сте готови да къмитвате вашите. Можете да изпълните и `git svn fetch` за сваляне на новите данни, но `git svn rebase` изтегля и обновява локалните ви къмити с една команда.

```
$ 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.
```

Изпълнявайте `git svn rebase` регулярно, за да сте сигурни, че локалния ви код е актуален. Обаче трябва да сте сигурни, че работната ви директория е в чист вид, когато пуснете командата. Ако имате некъмитнати промени, трябва или да ги маскирате (stash) или временно да ги къмитнете преди изпълнението на `git svn rebase`, иначе командата ще спре ако види, че пребазирането ще доведе до merge конфликт.

Проблеми с Git клонове

Когато започнете да се чувствате удобно с похватите на работа на Git, доста вероятно е да започнете да правите topic клонове, да работите в тях и да ги сливате. Ако публикувате към Subversion сървър с `git svn`, може да искате да пребазирате работата си върху единичен клон всеки път вместо да сливате клонове. Причината да предпочетете пребазирането е, че Subversion поддържа линейна история и не обработва сливанията така, както го прави Git. Така че `git svn` следва само първия родител при конвертирането на snapshot-ите в Subversion къмити.

Да допуснем, че историята ни изглежда така: създали сме `experiment` клон, направили сме два къмита и след това сме го слели обратно в `master`. При изпълнение на `dcommit`, ще видим нещо такова:

```
$ 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
```

Изпълнението на `dcommit` от клон със слята история работи добре с изключение на факта, че когато погледнете в историята на Git проекта, ще установите че двата къмита от клона `experiment` не са пренаписани, вместо това всички тези промени се появяват в SVN версията на единичния сливащ къмит.

Когато някой друг клонира тази работа, всичко което ще види е merge къмита ви с всичката работа обединена в него все едно сте изпълнили `git merge --squash`, но никакви подробности откъде и кога са промените направени от вас в `experiment`.

Subversion клонове

Клоновете в Subversion не са същите като в Git и ако можете да ги избягвате ще е най-добре. Но при все това, с `git svn` можете да създавате и да къмитвате в Subversion клонове.

Създаване на нов SVN клон

За да създадете нов клон в Subversion, използвайте `git svn branch [new-branch]`:

```
$ 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/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Това прави еквивалента на `svn copy trunk branches/opera` командата в Subversion и работи на Subversion сървъра. Важно е да посочим, че това не ви прехвърля автоматично в този клон и ако сега къмитнете, къмитът ще отиде в клона `trunk` на сървъра, вместо в `opera`.

Превключване на активни клонове

Git определя в кой клон отиват вашите dcommits гледайки върховете на всички ваши Subversion клонове в историята ви — трябва да имате само един и той трябва да е последния с `git-svn-id` в текущата ви branch история.

Ако искате да работите по повече от един клон едновременно, можете да настроите локалните клонове да правят `dcommit` към специфични Subversion клонове стартирайки ги от импортирания Subversion къмит за този клон. Ако искате клон `opera`, в който да работите отделно, може да изпълните:

```
$ git branch opera remotes/origin/opera
```

Сега, ако желаете да слеете вашия `opera` клон в `trunk` (вашия `master` клон), можете да го направите с нормална команда `git merge`. Но трябва да предоставите описателно къмит съобщение (чрез `-m`) или сливането ще гласи "Merge branch opera" вместо нещо полезно.

Помнете, че независимо че използвате `git merge` за тази операция и че сливането вероятно ще е много по-лесно отколкото би било в Subversion (защото Git автоматично ще установи подходящата merge база за вас), това не е стандартен Git merge къмит. Ще трябва да изпратите тези данни към Subversion сървър, който не може да обработва къмит следящ повече от един родител и когато го направите, те ще изглеждат като единичен къмит обединяващ цялата извършена работа от друг клон. След като слеете един клон в друг, не можете лесно да се върнете и да продължите да работите по този клон, както нормално бихте могли в Git. Командата `dcommit` изтрива всяка информация казваща кой клон е бил слят, така че евентуалните следващи merge-base изчисления ще са погрешни — `dcommit` прави така, че резултатът от `git merge` да изглежда като от `git merge --squash`. За жалост няма добър начин за избягване на такава ситуация — Subversion не може да пази тази информация и винаги ще бъдете ограничавани, когато го използвате като сървър. За да си спестите проблеми, добре е да изтривате локалния клон (в този случай `opera`) след като го

слеете в trunk.

Subversion команди

Инструментите на `git svn` осигуряват множество команди, които подпомагат по-лесното минаване към Git, предоставяйки функционалности подобни на тези в Subversion. Ето няколко команди, които ви дават това, което и Subversion.

История в SVN стил

Ако сте свикнали със Subversion и искате да видите историята си в SVN стил, може да използвате `git svn log`:

```
$ 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
```

Две неща са важни с `git svn log`. Първо, тя работи офлайн, за разлика от реалната `svn log` команда, която пита Subversion сървъра за данни. Второ, тя показва само къмитите, които са били публикувани на Subversion сървъра. Локалните Git къмити, които не сте публикували, не се показват — нито пък тези, които други разработчици евентуално са публикували на Subversion сървъра междувременно. Получавате нещо като last known статус на къмитите от сървъра.

SVN анотация

Точно както `git svn log` симулира командата `svn log` офлайн, можете да изпълните еквивалента на `svn annotate` с `git svn blame [FILE]`. Изходът изглежда така:

```
$ 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 Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Отново, вашите къмити и междувременно публикуваните от други хора промени в Subversion сървъра няма да се покажат.

Инфомация за SVN Server

Можете също да получите информацията, която `svn info` предоставя с `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)
```

Данните подобно на `blame` и `log` са офлайн и са актуални към момента, когато последно сте комуникирали със Subversion сървъра.

Игнориране на това, което Subversion игнорира

Ако клонирате Subversion хранилище, което има зададени `svn:ignore` настройки, вероятно бихте искали да създадете съответни `.gitignore` файлове, така че да не къмитнете по невнимание ненужни неща. `git svn` има две команди за случая. Първата е `git svn create-ignore`, която автоматично създава съответстващите `.gitignore` файлове, така че следващият ви къмит може да ги включи.

Втората команда е `git svn show-ignore`, която печата на stdout редовете, които трябва да вмъкнете в `.gitignore` файл така че бихте могли да пренасочите изхода ѝ в exclude файла си и да не правите индивидуални `.gitignore`:

```
$ git svn show-ignore > .git/info/exclude
```

По този начин не се налага да пълните проекта с `.gitignore` файлове. Това е добра опция, ако сте единствения Git потребител в Subversion екип и колегите ви не желаят да виждат `.gitignore` файлове в съвместния проект.

Git-Svn обобщение

Инструментите на `git svn` са полезни, ако си имате работа със Subversion сървър по една или друга причина. Би следвало да гледате на Subversion като на орязан Git или ще си имате проблеми в превода, предизвикващи объркване у вас и у колегите ви. За да си спестите главоболия, опитайте да спазвате тези правила:

- Поддържайте линейна Git история, която не съдържа сливащи къмити направени с `git merge`. Пребазирайте всяка работа, която сте извървили извън главния си клон обратно върху него, не я сливайте в него.
- Не работете паралелно по един и същи проект в Git сървър и Subversion сървър. По изключение може да имате един такъв за да ускорите клониранията за новите разработчици, но не публикувайте в него нищо, което не съдържа `git-svn-id` елемент. Може дори да искате да добавите `pre-receive` hook, който да проверява всяко къмит съобщение за наличие на `git-svn-id` поле и да отказва публикуванията с къмити, в които то липсва.

Спазвате ли тези съвети, работата ви със Subversion сървъри би могла да бъде по-поносима. Обаче, ако е налична възможност да преминете към реален Git сървър, това ще даде на екипа ви много повече позитиви.

Git и Mercurial

DVCS вселената не включва само Git. В действителност, налични са много други системи, всяка от която със собствени възгледи за това как трябва да се прави разпределен version control. Като изключим Git, най-популярната сред тях е Mercurial и двете са подобни в много аспекти.

Добрите новини са, че ако предпочитате да използвате Git локално, но се налага да работите по проект, чийто сорс код се контролира с Mercurial, съществува начин Git да функционира като клиент на Mercurial хранилища. Понеже Git използва remotes за комуникация със сървърни хранилища, не е изненадващо, че въпросният бридж е имплементиран като remote helper. Името на проекта е git-remote-hg, и може да се намери на https://github.com/felipec/git-remote-hg.

git-remote-hg

Първо, трябва да инсталираме git-remote-hg. Практически това се изчерпва с копирането на файла някъде в пътя ви:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

…допускаме че `~/bin` е включен в пътищата на променливата `$PATH`. Git-remote-hg има и още една друга зависимост: библиотеката `mercurial` за Python. Ако имате инсталиран Python, това е лесно:

```
$ pip install mercurial
```

Ако нямате Python, посетете https://www.python.org/ и го инсталирайте преди това.

Последното нещо, от което се нуждаем, е Mercurial клиента. Изтеглете го и го инсталирайте от https://www.mercurial-scm.org/.

Сега сте готови за работа. Имаме нужда от Mercurial хранилище, в което можем да публикуваме. За късмет, всяко Mercurial хранилище може да работи по такъв начин, така че ще използваме стандартното "hello world", което всички използват за да учат Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Начало

След като имаме подходящо “server-side” хранилище, можем да преминем през един стандартен работен процес. Както ще видите, тези две системи са доста подобни и не би трябвало да имате големи трудности.

Както винаги с Git, първо клонираме:

```
$ 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/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard 'hello, world' program
```

Веднага забелязваме, че при работата с Mercurial хранилища се използва стандартната `git clone` команда. Това е така, понеже git-remote-hg работи на сравнително ниско ниво, използвайки механизъм подобен на този, който Git използва за HTTP/S комуникация (remote helpers). И Git и Mercurial са проектирани така, че всеки клиент да има пълно копие на историята на хранилището и тази команда ви осигурява цялостно клониране, включващо цялата история на проекта — при това сравнително бързо.

Командата log показва два къмита, към последния от които сочи цял куп референции. Оказва се, че някои от тях реално не са там. Нека видим какво действително съдържа

директорията .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 се опитва да прави нещата в стила на Git, но зад кулисите тя управлява концептуалното свързване между две леко различаващи се системи. Директорията refs/hg съдържа действителните отдалечени референции. Например, refs/hg/origin/branches/default е Git ref файл, който съдържа SHA-1 започваща с “ac7955c”, което е къмита, към който сочи master. По този начин refs/hg директорията е нещо като фалшива refs/remotes/origin, но също така може да различава bookmarks и branches.

Файлът notes/hg е изходната точка за това как git-remote-hg съотнася хешовете на Git къмитите с changeset идентификаторите на Mercurial. Нека погледнем по-подробно:

```
$ 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
```

И така `refs/notes/hg` сочи към дърво, което в базата данни с обекти на Git е представено като списък от други обекти с имена. `git ls-tree` отпечатва режима, типа, хеш стойността и името на файла за елементите в дървото. Ако поразгледаме един от елементите на дървото, откриваме, че вътре има blob наречен “ac9117f” (SHA-1 хеша на къмита, към който сочи `master`), със съдържание “0a04b98” (което е ID-то на Mercurial changeset-а на върха на клона `default`).

Добрата новина е, че почти не се налага да се занимаваме с всичко това. Типичният работен процес няма да е много различен от този, при който работим с Git remote.

Трябва да обърнем внимание на още едно нещо преди да продължим, игнориранията. Mercurial и Git използват много подобен механизъм за това, но е вероятно да не искате да публикувате `.gitignore` файл в Mercurial хранилище. За щастие Git има начин да игнорира файлове, които са локални за хранилище на диска и форматът на Mercurial е съвместим с този на Git, така че просто трябва да копирате съответния файл:

```
$ cp .hgignore .git/info/exclude
```

Файлът `.git/info/exclude` работи точно като `.gitignore`, но не влиза в къмитите.

Работен процес

Да допуснем, че сме извършили някаква работа в проекта, имаме няколко къмита в клона `master` и сме готови да публикуваме промените. Ето как изглежда хранилището ни сега:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Нашият `master` клон е с два къмита напред спрямо `origin/master`, но те са само на локалната машина. Нека проверим дали някой друг не е публикувал някаква важна промяна междувременно:

```
$ 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/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Понеже използвахме флага `--all`, виждаме референциите "notes", които се използват вътрешно от git-remote-hg, можем да ги игнорираме. Останалото е каквото се очаква; `origin/master` е обновен с един къмит и историята ни сега е разклонена (diverged). За разлика от другите системи, които преглеждаме в тази глава, Mercurial може да обработва сливания, така че няма да се налага да правим нищо необичайно.

```
$ 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/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Перфектно. Пускаме си тестовете, те минават добре и сме готови да публикуваме промените си на сървъра:

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

Това е всичко! Ако погледнете Mercurial хранилището, ще видите че Push операцията е

направила каквото се очаква:

```
$ 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
```

Changeset-ът с номер *2* е създаден от Mercurial, а тези с номера *3* и *4* са направени от git-remote-hg, в резултат от публикуването на къмитите от Git.

Branches и Bookmarks

Git има само един вид клон: референция, която се премества с правенето на къмити. В Mercurial, този тип референция се нарича "bookmark," и тя се държи почти по същия начин като Git клон.

Концепцията на Mercurial за понятието "branch" е по-различна. Клонът, върху който е създаден changeset се записва *с changeset-а*, което значи че той винаги ще бъде в историята на хранилището. Ето пример за къмит направен в клона `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
```

Забележете реда, който започва с "branch". Git не може в действителност да пресъздаде това (а и не се нуждае, и двата типа клонове могат да се представят като Git референции), но git-remote-hg трябва да разбира разликата, защото Mercurial ѝ обръща внимание.

Създаването на Mercurial bookmarks е лесно колкото създаването на Git клонове. На Git страната:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

Това е всичко по въпроса. От страната на Mercurial това изглежда така:

```
$ 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
```

Забележете новия таг `[featureA]` на revision 5. Тези работят точно както Git клонове от страната на Git с едно изключение: не можете да изтриете bookmark от страната Git (това е ограничение на remote helpers).

Освен с Mercurial bookmarks, можете да работите и с Mercurial клонове: просто ги създавате в `branches` namespace:

```
$ 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
```

Ето как ще изглежда това от страната на 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'
[...]
```

Името на клона "permanent" беше записано с changeset-а маркиран като *7*.

От гледна точка на Git, работата с тези два вида клонове е една и съща: превключвате, къмитвате, изтегляте, сливате и публикувате както нормално се прави в Git. Нещо, което трябва да отбележим е, че Mercurial не поддържа презапис на историята, към нея може само да се добавя. Ето как ще изглежда нашето Mercurial хранилище след интерактивно пребазиране последвано от force-push:

```
$ 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
```

Changeset-ите *8*, *9*, и *10* са създадени и принадлежат към клона `permanent`, но старите changesets са все още там. Това може да е **много** смущаващо за колегите ви използващи Mercurial, така че се старайте да го избягвате.

Mercurial обобщение

Git и Mercurial са много подобни, така че съвместната им работа е сравнително безболезнена. Ако избягвате да преправяте историята, която публикувате (още веднъж, това е горещо препоръчително), може дори и да не разберете, че от другия край на връзката стои Mercurial система.

Git и Bazaar

Друга популярна DVCS система е Bazaar. Bazaar е безплатна система с отворен код, част от проекта GNU Project. Тя работи по много различен начин от Git. Понякога, за да направите

едно и също нещо като в Git, ще трябва да използвате различна ключова дума и също така някои често използвани ключови думи нямат значението, което може би бихте очаквали. По-специално, управлението на клонове е много различно и може да предизвика конфуз, особено сред Git потребителите. Но въпреки всичко, да се работи с Bazaar хранилище от Git такова, е възможно.

Съществуват много проекти, които позволяват да използвате Git като Bazaar клиент. Тук ще използваме проекта на Felipe Contreras, който можете да намерите на https://github.com/felipec/git-remote-bzr. За да го инсталирате, трябва просто да свалите git-remote-bzr в папка от пътя ви:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O
~/bin/git-remote-bzr
$ chmod +x ~/bin/git-remote-bzr
```

Ще ви трябва и инсталиран Bazaar. Това е всичко.

Създаване на Git хранилище от Bazaar хранилище

Инструментът е лесен за ползване. Достатъчно е да клонирате Bazaar хранилище като му дадете префикс `bzr::`. Понеже Git и Bazaar правят пълно клониране върху локалния компютър, възможно е да прикачите Git копие към локалното ви Bazaar копие на хранилище, въпреки че това не е препоръчително. Много по-лесно е да закачите вашето Git копие към същото място, към което е закачено Bazaar копието — към централното хранилище.

Приемаме, че работим с отдалечено хранилище на адрес `bzr+ssh://developer@mybazaarserver:myproject`. Клонираме го така:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git
$ cd myProject-Git
```

На този етап Git хранилището е създадено, но не е оптимизирано за ефективно използване на диска. Ето защо е хубаво да почиствате и compact-вате вашите Git хранилища, особено ако са големи:

```
$ git gc --aggressive
```

Bazaar клонове

Bazaar позволява само да клонирате клонове, но едно хранилище може да има много такива и `git-remote-bzr` може да ги клонира всички. Например, за да клонирате клон:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

За да клонирате цялото хранилище:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

Втората команда прави копия на всички клонове в emacs хранилището, но е възможно да укажете и само някои от тях:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Някои отдалечени хранилища не позволяват да показвате клоновете им и в такива случаи трябва да ги укажете ръчно. Въпреки, че можете да го направите в командата за клониране, това може да ви се стори по-лесно:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

Игнориране с .bzrignore

Понеже работите по Bazaar-контролиран проект, не трябва да създавате `.gitignore` файл, защото *можете* по невнимание да го включите в контрола и тогава другите хора работещи с Bazaar ще бъдат объркани. Решението е да създадете `.git/info/exclude` файл като символна връзка или като нормален такъв. По-късно ще видим как да се справим с това.

Bazaar използва същия модел за игнориране на файлове като Git, но в допълнение има две възможности без еквивалент в Git. Пълното описание може да се намери в документацията. Тези две възможности са:

1. "!!" позволява да игнорирате определени файлови маски дори ако те са указани с "!" правило.
2. "RE:" в началото на реда позволява да укажете като маска Python регулярен израз (Git позволява само шел globs).

В резултат на това, възникват две различни ситуации:

1. Ако `.bzrignore` файлът не съдържа някой от двата префикса, тогава просто можете да направите символна връзка към него в хранилището: `ln -s .bzrignore .git/info/exclude`.
2. В противен случай, трябва да създадете файл `.git/info/exclude` и да го адаптирате така, че да игнорира същите файлове, които игнорира и `.bzrignore`.

Във всички случаи трябва да бъдете внимателни за всяка бъдеща промяна по `.bzrignore` за да сте сигурни, че `.git/info/exclude` винаги отразява коректно съдържанието му. Ако `.bzrignore` се промени и се появят редове започващи с "!!" или "RE:", то Git няма да може да ги интерпретира и ще трябва да адаптирате вашия `.git/info/exclude` файл да прави същото като `.bzrignore`. Освен това, ако `.git/info/exclude` преди това е бил символна връзка, първо

ще трябва да я изтриете, да копирате `.bzrignore` в `.git/info/exclude` и да го редактирате за коректна адаптация. Обаче, бъдете внимателни със създаването на файла, защото с Git не е възможно да включите повторно файл, ако родителската директория на този файл е изключена.

Издърпване на промени от отдалечено хранилище

За издърпване на промени се използват стандартните Git команди. Подразбирайки, че промените ви са в `master` клона, вие сливате или пребазирате работата си върху `origin/master` клона:

```
$ git pull --rebase origin
```

Публикуване в отдалечено хранилище

Bazaar също като Git поддържа концепцията за merge къмити, така че няма да има проблем с публикуването на такива. Можете спокойно да работите по отделен клон, да го слеете в `master` и след това да публикувате. Създавате клонове, тествате и къмитвате както обикновено. Накрая изпращате работата си в Bazaar хранилището:

```
$ git push origin master
```

Недостатъци

Remote helper-ите на Git си имат някои ограничения. По-специално, следните команди не работят:

- git push origin :branch-to-delete (Bazaar не може да приема изтриване на референции по този начин)
- git push origin old:new (ще се публикува `old`)
- git push --dry-run origin branch (ще се направи действително публикуване)

Обобщение

Моделите на Git и Bazaar си приличат и по тази причина съвместната им работа не е толкова проблемна. Стига да съблюдавате ограниченията и винаги да помните, че отдалеченото хранилище не е реално Git такова, не би следвало да срещате съществени трудности.

Git и Perforce

Perforce е много популярна version-control система в корпоративните среди. Съществува от 1995 г. и това я прави най-старата от всички, които разглеждаме в тази глава. Като такава, тя е проектирана съобразно нуждите от онези времена и подразбира, че сте свързани към единичен централен сървър и само една версия се пази на локалния диск. За да сме справедливи, нейните функционалности и ограничения обслужват много добре някои

специфични задачи, но на практика съществуват много проекти, които използват Perforce а биха работили много по-добре на Git.

Ако искате да съчетаете използването на Perforce и Git, налични са две опции. Първата, която ще опишем е “Git Fusion” бриджа създаден от авторите на Perforce, който ви позволява да третирате subtrees от Perforce депо като read-write Git хранилища. Втората, git-p4, е клиентски бридж, който позволява да използвате Git като Perforce клиент без да се налага каквато и да било промяна в настройките на Perforce сървъра.

Git Fusion

Perforce предоставя продукта Git Fusion (наличен на http://www.perforce.com/git-fusion), който синхронизира Perforce сървър с Git хранилища от страна на сървъра.

Настройка

За нашите примери ще използваме най-лесния метод за инсталиране на Git Fusion, виртуална машина, в която работят Perforce демона и Git Fusion. Можете да свалите имидж за виртуалната машина от http://www.perforce.com/downloads/Perforce/20-User, и след това да го стартирате в предпочитания от вас виртуализационен софтуер (в случая ползваме VirtualBox).

След стартирането си, виртуалната машина изисква задаване на пароли за три Linux потребителя (`root`, `perforce` и `git`) и предоставя име на инстанцията, което може да се използва за различаване на конкретната инсталация от другите в същата мрежа. Когато всичко това приключи ще видите следното:

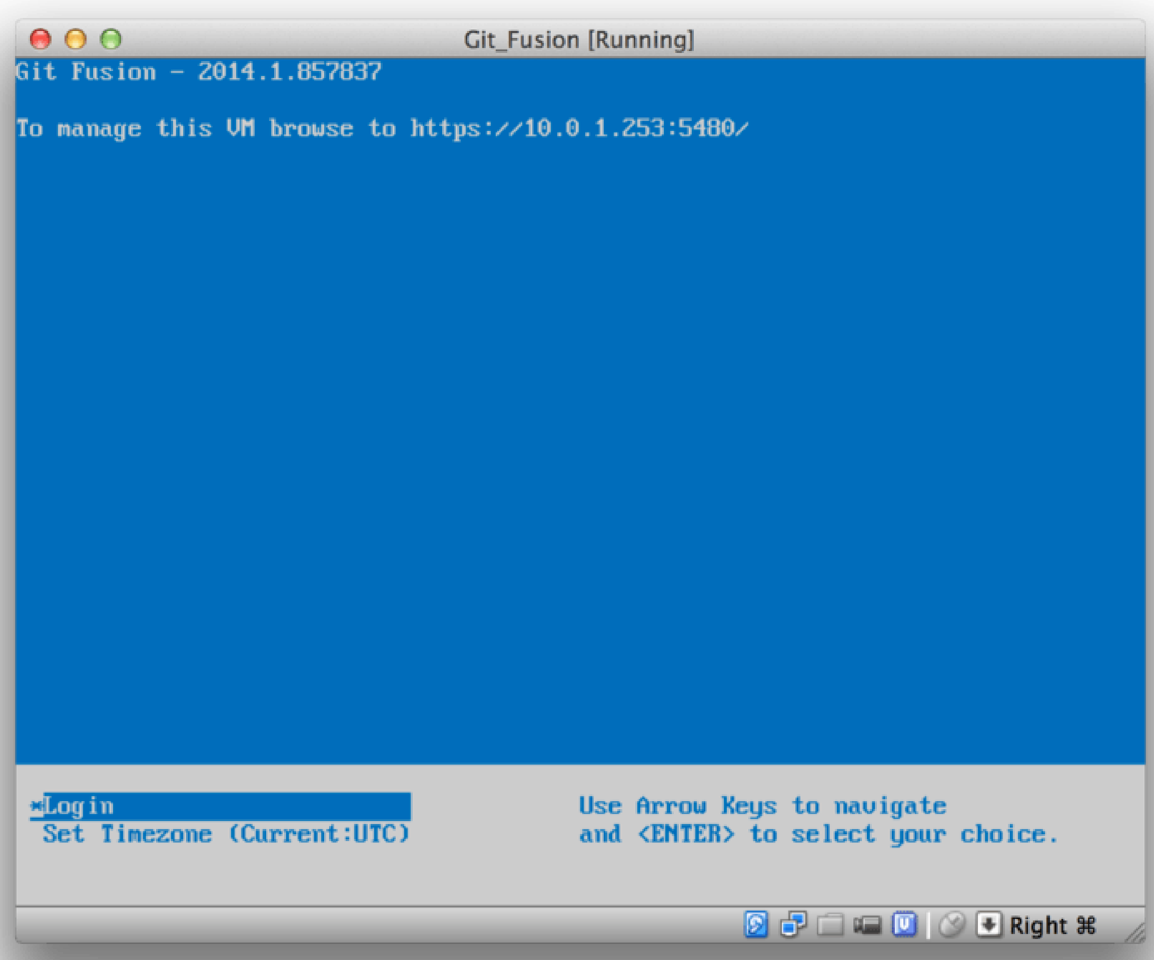


Figure 145. Стартов екран на виртуалната машина на Git Fusion

Запомнете показания тук IP адрес, ще го ползваме по-късно. След това, ще създадем потребител за Perforce. Изберете опцията “Login” в долния край на екрана и натиснете enter (или влезте в машината през SSH), след това влезте като `root`. След това изпълнете тези команди, за да създадете новия потребител:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

Първата ще отвори редактора Vi за настройка на потребителя, можете да приемете настройките по подразбиране изпълнявайки `:wq` и enter. Следващата команда ще ви пита два пъти за паролата. Това е всичко, което правим в конзолата, така че можете да излезете от сесията.

Следващото нещо, което следва да направим, е да кажем на Git да не проверява SSL сертификатите. Виртуалната машина на Git Fusion идва със сертификат за домейн, който няма да съвпада с IP адреса на виртуалната машина и Git ще отхвърля HTTPS връзката. Ако това ще бъде перманентна инсталация, можете да погледнете документацията на Git Fusion

и да видите как да инсталирате различен сертификат. За целите на демонстрацията това няма да е нужно:

```
$ export GIT_SSL_NO_VERIFY=true
```

Сега можем да тестваме дали всичко работи.

```
$ 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.
```

Имиджът идва с инсталиран примерен проект, който може да се клонира. Тук клонираме през HTTPS като потребител `john`, който създадохме преди малко. Git ще пита за име и парола за конекцията първия път, но после credential кешът ще ви спести това неудобство.

Конфигурация на Fusion

След като имате инсталиран Git Fusion, ще променим леко конфигурацията. Това е сравнително лесно с предпочитания от вас Perforce клиент, просто асоциирайте `//.git-fusion` директорията на Perforce сървъра в работното си пространство. Файловата структура изглежда така:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

Директорията `objects` се използва служебно от Git Fusion за асоцииране на Perforce обекти с

Git и обратно, няма да се налага да пипате нищо в нея. В нея има глобален p4gf_config файл, както и по един такъв за всяко хранилище — това са конфигурационните файлове, които определят поведението на Git Fusion. Нека видим файла в главната директория:

```
[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
```

Няма да задълбаваме в подробности за всички флагове тук, просто отбележете, че това е INI-форматиран текстов файл, подобен на тези, които и Git използва за конфигурации. Този файл задава глобалните опции, които могат да бъдат презаписани от съответния файл в конкретно хранилище като например repos/Talkhouse/p4gf_config. Ако отворите подобен такъв, ще видите секцията [@repo] с няколко настройки, които се различават от глобалните. Ще видите и секции като:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Това е мапинг между Perforce клон и Git клон. Тази секция може да се казва както желаете, стига да е уникална. git-branch-name позволява да конвертирате пътя на депо, който може да е объркващ под Git към нещо по-разбираемо. Настройката view определя как Perforce файловете се съотнасят в Git хранилището, използва се стандартния view mapping синтаксис. Както е видно, може да се задава повече от един мапинг:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

По този начин, ако нормалният ви workspace мапинг включва промени в структурата на директориите, може да ги отразите в Git хранилището.

Последният файл, на който обръщаме внимание е `users/p4gf_usermap`, който съотнася Perforce потребители на Git потребители, и от който може въобще да не се нуждаете. Когато конвертира от Perforce changeset към Git къмит, Get Fusion по подразбиране ще потърси Perforce потребителя и ще използва имейл адреса и пълното му име за съответните author/committer полета в Git. В обратната посока, подразбиращото се поведение е да се потърси Perforce потребител с имейл адрес съответстващ на author полето на Git къмита и changeset-а да се запише като направен от този потребител (като се вземат предвид и правата за достъп). В повечето случаи тази схема работи коректно, но погледнете този мапинг файл:

```
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"
```

Всеки ред следва формата `<user> <email> "<full name>"` и дефинира единично съответствие за потребител. Първите две полета асоциират два отделни имейл адреса с един и същи Perforce потребителски акаунт. Това е полезно, ако сте създали Git къмити с различни имейл адреси (или пък сте сменили адреса), но искате да ги присъедините към същия Perforce потребител. Когато създавате Git къмит от Perforce changeset, първият ред съответстващ на Perforce потребителя се използва за попълване на информацията за автора в Git.

Последните два реда маскират действителните имена и имейл адреси на Bob и Joe, от създадените Git къмити. Това е хубаво, в случай че решите да отворите кода на вътрешен проект, но не искате да разкриете списъка с разработчици в компанията ви на останалия свят. Запомнете, че имейл адресите и пълните имена трябва да са уникални, освен ако не искате всички Git къмити да са с фиктивен единичен автор.

Работен процес

Git Fusion е двупосочен бридж между Perforce и Git. Да видим как изглежда работата от гледната точка на Git. Ще допуснем, че сме асоциирали проекта “Jam” с конфигурационен файл като показания по-горе. Клонираме го така:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://john@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 metrowerks on
Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

Първият път, когато направите това, може да отнеме повече време. Това, което се случва е, че Git Fusion конвертира всички приложими changesets в Perforce историята в Git къмити. Това се случва локално на сървъра, така че е сравнително бързо, но все пак зависи от мащаба на историята. Последващите изтегляния правят инкрементално конвертиране, така че наподобяват скоростта на Git, с която сме свикнали.

Както може да видите, хранилището ни изглежда точно като всяко друго с Git. Имаме три клона и Git услужливо е създал локален `master` клон, който проследява `origin/master`. Нека направим малко промени и създадем няколко къмита:

```
# ...
$ 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.
[...]
```

Сега имаме два нови къмита. Следва да проверим дали някой друг колега не е публикувал нови промени:

```
$ 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.
[...]
```

Изглежда има такива. От този изглед няма как да разберете това, но къмитът `6afeb15` е създаден в действителност от Perforce клиент. От гледна точка на Git, този къмит изглежда като нормален Git къмит и точно това е идеята. Нека да видим как Perforce сървърът управлява merge къмит:

```
$ 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
```

Според Git нещата работят. Нека видим историята на `README` файла от гледната точка на Perforce с revision graph опцията на `p4v`:

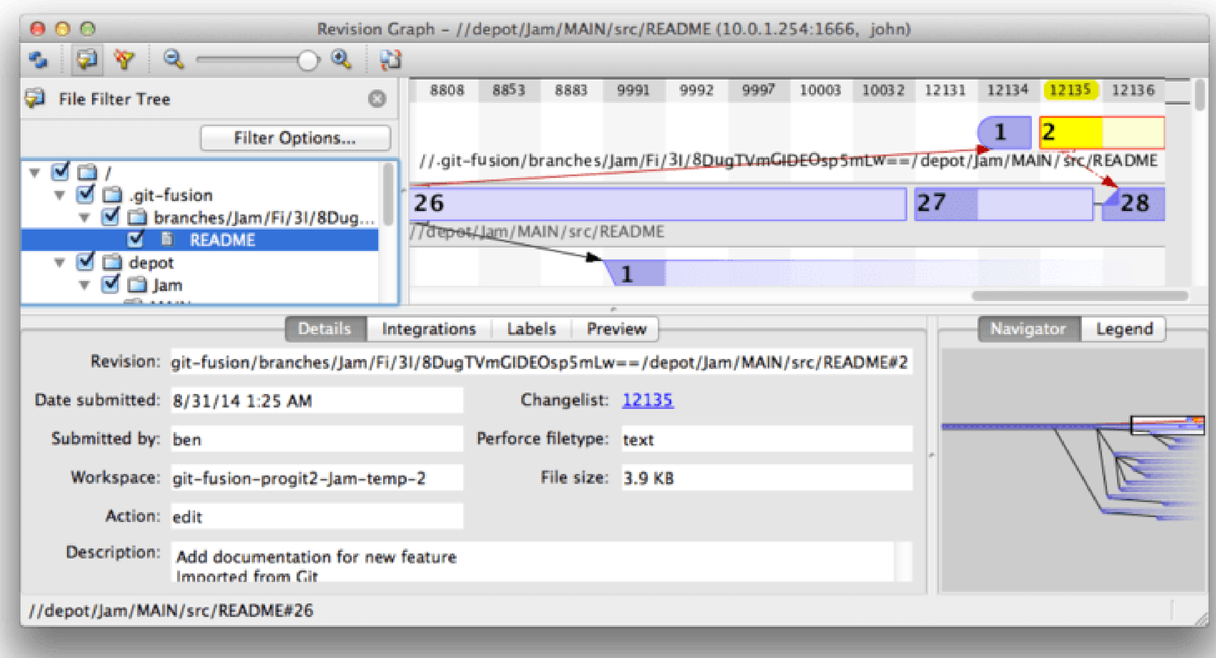


Figure 146. Perforce revision графика след Git push

Ако никога преди не сте виждали такъв изглед, той може да изглежда смущаващо, но показва същите концепции както графичния инструмент на Git за разглеждане на историята. Ние гледаме историята на файла `README` така че дървото горе вляво показва само него както се вижда в различните клонове. Горе вдясно виждаме как се свързват различните версии на файла, а общата картина на тази графика е долу вдясно. Останалата част от картинката е за детайлите на избраната версия (`2` в този случай).

Едно от нещата, които се виждат е, че графиката изглежда точно както тази в Git историята. Perforce няма именуван клон за съхранение на къмитите `1` и `2`, така че е създал “anonymous” клон в директорията `.git-fusion` за тях. Същото ще се случи за именувани Git клонове, които не съвпадат с именувани Perforce клонове (и после можете да ги мапнете към Perforce клон с конфигурационния файл).

Повечето от това се случва задкулисно, но крайният резултат е, че един човек в екипа може да използва Git, друг Perforce и нито един от тях няма да знае за избора на другия.

Git-Fusion обобщение

Ако имате достъп (или може да получите такъв) към вашия Perforce сървър, Git Fusion е чудесен начин да накарате Git и Perforce да си сътрудничат. Има известна работа по конфигурацията, но като цяло не е толкова сложно. Това е една от малкото секции в тази глава, в които няма да видите предупреждения за използването на пълната сила на Git. Това не означава, че Perforce ще е щастлив с всичко, което му изпратите — ако се опитате да промените история, която вече е била публикувана, Git Fusion ще откаже това, но иначе се опитва да бъде максимално близък до Git. Можете дори да използвате Git модули (въпреки, че те ще изглеждат странно за Perforce потребителите) и да сливате клонове (това ще се запише като integration от страната на Perforce).

Ако не успеете да убедите администратора на сървъра ви да инсталира Git Fusion, все пак

съществува възможност да използвате двете системи заедно.

Git-p4

Git-p4 е двупосочен бридж между Git и Perforce. Той работи изцяло във вашето Git хранилище, така че не ви трябва никакъв достъп до Perforce сървър (като изключим потребителските име и парола, разбира се). Git-p4 не е толкова гъвкаво и пълно решение като Git Fusion, но ви позволява да правите повечето неща, които бихте желали, без да се налага да пипате по сървъра.

Трябва да поставите `p4` инструмента някъде в пътищата на `PATH` променливата, за да работите с git-p4. По времето на писането на този текст, той е наличен на http://www.perforce.com/downloads/Perforce/20-User.

Настройка

За целите на примерите, ще използваме Perforce сървъра от Git Fusion OVA както е показано по-горе, но ще заобикаляме Git Fusion сървъра и ще комуникираме директно с Perforce version control системата.

За да използваме `p4` конзолния клиент (от който зависи работата на git-p4), трябва да настроим няколко променливи на обкръжението:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Начало

Както всичко в Git, първата команда е за клониране:

```
$ 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/master
```

Това създава, според терминологията на Git, "shallow" копие; само последната Perforce версия се импортира в Git; спомнете си, че Perforce не е проектиран да предоставя всяка версия на всеки потребител. Това е достатъчно да използваме Git като Perforce клиент, но за други цели не е.

След края на командата, имаме пълнофункционално Git хранилище:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from
the state at revision #head
```

Виждаме, че съществува отдалечена “p4” референция към Perforce сървъра, но всичко друго изглежда като стандартно клонирано копие. В действителност това е малко заблуждаващо, защото такава отдалечена референция не съществува тук.

```
$ git remote -v
```

Не се показват никакви отдалечени референции. Git-p4 е създал няколко референции за да пресъздаде статуса на сървъра и за `git log` те изглеждат като отдалечени такива. Обаче не се управляват от Git и не можете да публикувате в тях.

Работен процес

Нека да направим малко промени. Допускаме, че сте постигнали напредък по много важна функционалност и сте готови са я споделите с екипа.

```
$ 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
revision #head
```

Имаме два нови къмита и искаме да ги изпратим на Perforce сървъра. Проверяваме за междувременно въведени промени от някой друг:

```
$ 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
```

Изглежда, че такива има и клоновете `master` и `p4/master` са разделени. Системата за клонове на Perforce *няма нищо общо* с тази на Git, така че изпращането на merge къмити няма никакъв смисълъл. Git-p4 препоръчва да пребазирате къмитите си и дори предлага кратък път за това:

```
$ 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(-)
```

Вероятно и сами сте забелязали от резултата, но все пак да кажем, че `git p4 rebase` е съкратен вариант на `git p4 sync` последвана от `git rebase p4/master`. Командата дори е една идея по-умна, особено при работа с много клонове, но засега това ни е достатъчно.

Сега историята ни отново е линейна и сме готови да изпратим промените си към Perforce. Командата `git p4 submit` ще се опита да създаде нова Perforce ревизия за всеки Git къмит между `p4/master` и `master`. Изпълнението ѝ ще ни отвори текстовия редактор със следното съдържание:

```
# 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 18:26:05.000000000 0000
@@ -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>
```

Това в голямата си част е същото съдържание, което ще видите при изпълнение на `p4 submit`, с изключение на нещата в края, които git-p4 услужливо е вмъкнала. Git-p4 се опитва

да съблюдава вашите Git и Perforce настройки индивидуално, когато трябва да даде име за къмит или changeset, но в някои случаи ще искате да го редактирате. Например, ако Git къмитът, който импортирате, е бил създаден от потребител, който няма Perforce потребителски акаунт, може все още да искате полученият changeset да изглежда така сякаш е създаден от този потребител (не от вас).

Git-p4 удобно е вмъкнал съобщението от Git къмита като съдържание за този Perforce changeset, така че просто трябва да запишем и да излезем, два пъти (по веднъж за всеки къмит). Резултатът на екрана ще изглежда подобно:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
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
```

Изглежда сякаш сме изпълнили `git push`, най-близкия аналог на това, което реално стана.

По време на този процес всеки Git къмит се превръща в Perforce changeset; ако искате да ги съчетаете в единичен changeset, може да го направите с интерактивно пребазиране преди да изпълните `git p4 submit`. Също така отбележете, че SHA-1 хешовете на всички изпратени като changesets къмити са променени, това е защото git-p4 добавя по ред в края на всеки конвертиран къмит:

```
$ 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]
```

Какво се случва, ако опитате да изпратите merge къмит? Нека опитаме. Ето ситуацията, в която попадаме:

```
$ 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
```

Историята на Git и Perforce се разделя след 775a46f. Git страната има два къмита, след това merge къмит с Perforce head, и след това още един къмит. Ще се опитаме да изпратим всичко това върху единичен changeset от Perforce страната. Какво се случва, ако опитаме да публикуваме:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

Флагът `-n` е съкратено за `--dry-run` и се опитва да докладва какво би се случило, ако submit командата беше реална. В този случай, изглежда че ще се опитаме да създадем три Perforce changeset-а, съответстващи на трите non-merge къмита, които все още не съществуват на Perforce сървъра. Звучи като това, което искаме — нека изпълним реалната команда:

```
$ 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
```

Историята ни стана линейна, точно както ако бяхме пребазирали преди публикуването (което в действителност е реалният резултат). Това значи, че може да се чувствате свободни да създавате, работите, изтривате и сливате клонове локално в Git без да се тревожите, че историята ви ще стане несъвместима с Perforce. Всичко, което можете да пребазирате, може да се изпрати към Perforce сървъра.

Клонове

Ако вашият Perforce проект има много клонове, не е страшно, git-p4 може да се справи с тази ситуация по начин, по който сте свикнали с Git. Нека кажем, че вашето Perforce депо изглежда така:

```
//depot
  └── project
      ├── main
      └── dev
```

И нека приемем, че имате `dev` клон с view spec подобен на това:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 може автоматично да установи ситуацията и да направи правилното:

```
$ 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
```

Отбележете “@all” указателя в пътя на депото, той казва на git-p4 да клонира не само най-новия changeset за това под-дърво, но също и всички changesets, които някога са докосвали тези пътища. Това е близко до концепцията на Git за клон, но може да отнеме време при проекти с голяма история.

Флагът `--detect-branches` казва на git-p4 да използва branch-спецификациите на Perforce за да мапва клоновете с Git референции. Ако тези мапинги не са налични на Perforce сървъра (което е съвсем валиден начин за използване на Perforce), можете вие да кажете на git-p4 какви са мапингите и ще получите същия резултат:

```
$ 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 .
```

Ако за конфигурационната променлива `git-p4.branchList` зададем стойност `main:dev`, това казва на git-p4, че “main” и “dev” са клонове и че вторият произлиза от първия.

Ако сега изпълним `git checkout -b dev p4/project/dev` и направим няколко къмита, git-p4 е достатъчно умен да определи кой е правилния клон при изпълнение на `git p4 submit`. За съжаление, git-p4 не може да смесва shallow копия и множество клонове, ако имате голям проект и искате да работите по повече от един клон, ще трябва да изпълните `git p4 clone` по веднъж за всеки клон, в който искате да публикувате.

За създаването и интеграцията на клонове ще трябва да използвате Perforce клиент. Git-p4 може да синхронизира и публикува към съществуващи клонове и само с един линеен changeset в даден момент. Ако слеете два клона в Git и се опитате да публикувате новия changeset, всичко което ще бъде записано ще множество файлови промени, метаданните с информация за това кои клонове са участвали в интеграцията, ще бъдат загубени.

Git и Perforce - обобщение

Инструментът Git-p4 прави възможно използването на Git работен процес при работа с Perforce сървър и е добър в това. Обаче, важно е да се помни, че Perforce обработва сорс кода и Git се използва само за локална работа. Просто бъдете наистина внимателни при споделяне на Git къмити, ако имате отдалечена референция ползвана и от други хора, не публикувайте никакви къмити, които преди това не са изпратени към Perforce сървъра.

Ако искате свободно да смесвате използването на Perforce и Git като клиенти за сорс контрол и ако успеете да убедите системния ви администратор да го инсталира, то Git Fusion превръща Git в първокласен version-control клиент за Perforce сървър.

Миграция към Git

Ако имате наличен код под друга VCS, но искате да използвате Git, ще трябва да мигрирате проекта по някакъв начин. Тази секция разглежда някои популярни импортиращи инструменти и показва как да си направите свой собствен потребителски importer. Ще научим как се импортират данни от няколко от най-големите SCM системи защото те формират болшинството потребители, които мигрират и защото за тях се предлагат висококачествени инструменти.

Subversion

Ако сте прочели секцията за използването на `git svn`, може да сте ползвали тези инструкции за да направите `git svn clone` към хранилище, след това да сте спрели да използвате Subversion сървъра, да сте публикували в нов Git сървър и да ползвате него занапред. Ако искате историята, може да направите това толкова бързо колкото може да теглите от Subversion сървъра (което може да отнеме доста време).

Обаче, импортът не е перфектен и понеже може да отнеме време, няма да е лошо да го направите правилно. Първият проблем е с информацията за автора. В Subversion, всеки къмитващ автор има акаунт на сървъра, който се записва в къмит информацията. Примерите в предишната секция показваха `schacon` в някои позиции, като например `blame` изхода и `git svn log`. Ако искате да свържете тази информация с информацията за авторите в Git, ще ви трябва някакъв мапинг между Subversion потребителските акаунти и Git авторите. Създайте файл `users.txt`, който прави мапинга така:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

За да получите списък с имената на авторите, които SVN използва:

```
$ svn log --xml --quiet | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

Това генерира лог изхода в XML формат, след това филтрира само редовете с author данни, премахва повторенията и изчиства XML таговете. Очевидно това ще работи при налични

grep, sort, и perl. След това, пренасочете този изход във файла users.txt, така че да добавите еквивалентните Git потребителски данни във всеки ред.

Ако пробвате това на Windows машина, ще срещнете проблем. Microsoft предлага няколко добри съвета и примера на адрес https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git.

Може да подадете този файл на командата git svn за да я подпомогнете в по-акуратното мапване на данните за авторите. Можете също да укажете на git svn да не включва метаданните, които Subversion нормално импортира с флага --no-metadata към командите clone или init. Метаданните включват git-svn-id вграден във всяко къмит съобщение, което Git ще генерира по време на импорта. Това може да задръсти вашия Git лог и да го направи по-неясен.

Ще имате нужда от метаданните, ако искате да клонирате къмитите направени в Git хранилището обратно в оригиналното SVN такова. Ако не желаете синхронизацията във вашия commit-лог, може спокойно да пропуснете параметъра --no-metadata.

Така командата import ще изглежда по следния начин:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

Сега ще имате по-красив Subversion импорт в директорията my_project. Вместо къмити изглеждащи така:

```
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
```

те ще се показват като:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Сега не само полето Author изглежда по-добре, но и git-svn-id информацията няма да се

появява.

Бихте могли да направите и допълнително почистване след импорта. Добре би било да махнете странните референции, които `git svn` настройва. Първо ще преместите таговете, така че да бъдат реални тагове вместо странни отдалечени референции и след това ще преместите останалите клонове, така че да станат локални.

За да превърнете таговете в реални Git тагове, изпълнете:

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git
tag ${t/tags\//} $t && git branch -D -r $t; done
```

Това ще вземе референциите представляващи отдалечени клонове и започващи с `refs/remotes/tags/` и ще ги конвертира в реални олекотени тагове в Git.

След това, преместваме останалите референции от `refs/remotes` като локални клонове:

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch
$b refs/remotes/$b && git branch -D -r $b; done
```

Може да се случи да видите някои допълнителни клонове със суфикс `@xxx` (където xxx е число), докато в Subversion виждате само един клон. Това в действителност е Subversion функция наречена “peg-revisions”, за която Git просто няма синтактичен аналог. По тази причина `git svn` просто добавя svn version номера към името на клона точно по същия начин, по който бихте го написали в svn за да адресирате въпросния peg-revision в този клон. Ако тези peg-revisions не ви интересуват, може просто да ги премахнете:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D
$p; done
```

Сега всички стари клонове са реални Git клонове и всички стари тагове са реални Git такива.

Има още едно последно нещо за коригиране. За съжаление, `git svn` създава допълнителен клон с име `trunk`, който съответства на клона по подразбиране на Subversion, но указателят `trunk` сочи към същото място, към което и `master`. Понеже `master` е нещото, с което сме свикнали с Git, ето как да премахнете допълнителния клон:

```
$ git branch -d trunk
```

Последно, добавяме нашия нов Git сървър като remote и публикуваме в него:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Понеже искаме всичките ни клонове и тагове да се публикуват, можем да изпълним:

```
$ git push origin --all
$ git push origin --tags
```

Сега всичките клонове и тагове ще са в новия ни Git сървър в резултат на един подреден и изчистен импорт.

Mercurial

Mercurial и Git имат подобни модели за представяне на версиите и понеже Git е малко по-гъвкав, конвертирането на хранилище от Mercurial към Git е сравнително лесно чрез инструмента "hg-fast-export", който можете да свалите от:

```
$ git clone https://github.com/frej/fast-export.git
```

Първата стъпка е да се сдобием с пълно копие на хранилището на Mercurial, което ще конвертираме:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Следващата е да създадем author mapping файл. Mercurial е по-малко рестриктивен от Git по отношение на това какво може да се слага в author полето на changeset-ите, така че това е удобен момент за почистване. Създаването на списъка отнема една команда в `bash` шела:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Това ще отнеме няколко секунди в зависимост от дължината на историята на проекта, след което файлът `/tmp/authors` ще изглежда по подобен начин:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

В този конкретен пример, един и същи човек (Bob) е създал changeset-и под четири различни имена, едно от които изглежда коректно и друго, което ще е изцяло невалидно за един Git къмит. Hg-fast-export ни позволява да коригираме това превръщайки всеки ред в правило: `"<input>"="<output>"`, мапвайки `<input>` към `<output>`. В стринговете `<input>` и `<output>`, са позволени всички escape последователности, които се поддържат от `string_escape` енкодинга на python. Ако author mapping файлът не съдържа съответен `<input>`, този автор ще се изпрати към Git непроменен. Ако всички потребителски имена изглеждат добре, то въобще няма да се нуждаем от такъв файл. В този пример искаме файлът ни да

изглежда така:

```
"bob"="Bob Jones <bob@company.com>"
"bob@localhost"="Bob Jones <bob@company.com>"
"bob <bob@company.com>"="Bob Jones <bob@company.com>"
"bob jones <bob <AT> company <DOT> com>"="Bob Jones <bob@company.com>"
```

Същият вид мапинг файлове може да се използва за преименуване на клонове и тагове, когато дадено Mercurial име не е позволено за Git.

След това е време да създадем новото ни Git хранилище и да пуснем експортиращия скрипт:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Флагът `-r` инструктира hg-fast-export къде да намери Mercurial хранилището, което ще се конвертира, а `-A` указва къде е author-mapping файла (съответно за файловете за клонове и тагове се използват флаговете `-B` и `-T`). Скриптът парсва Mercurial changeset-ите и ги конвертира в скрипт за целите на "fast-import" функцията на Git (ще я разгледаме малко по-късно). Това отнема малко време (но за сметка на това е *много* по-бързо, в сравнение с времето необходимо, ако трябваше да се прави по мрежата) и изходът е доста подробен:

```
$ /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 files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602
attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

Това е почти всичко. Всички Mercurial тагове са конвертирани в Git тагове и Mercurial клоновете и bookmarks обектите са превърнати в съответните Git клонове. Сега сте готови да публикувате хранилището в сървъра:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Bazaar

Bazaar е DVCS инструмент подобен на Git и е сравнително лесно да конвертирате Bazaar хранилище в Git такова. За целта ви трябва плъгина `bzr-fastimport`.

Инсталация на bzr-fastimport плъгина

Процедурата е различна под UNIX операционни системи и Windows. В първия случай, най-лесният начин е да се инсталира пакета `bzr-fastimport`, който ще си изтегли и всички необходими допълнителни изисквания.

Например, под Debian и дериватите му, може да използвате:

```
$ sudo apt-get install bzr-fastimport
```

С RHEL варианти командата е:

```
$ sudo yum install bzr-fastimport
```

При Fedora от версия 22 има нов пакетен мениджър, dnf:

```
$ sudo dnf install bzr-fastimport
```

Ако пакетът не е наличен, може да го инсталирате като плъгин:

```
$ mkdir --parents ~/.bazaar/plugins     # създава необходимите директории за
плъгини
$ cd ~/.bazaar/plugins
$ bzr branch lp:bzr-fastimport fastimport   # импортира fastimport плъгина
$ cd fastimport
$ sudo python setup.py install --record=files.txt   # инсталира плъгина
```

За да работи този плъгин, ще ви трябва също и `fastimport` модула за Python. Може да проверите дали е наличен и, ако трябва да го инсталирате, така:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

Ако не е наличен, можете да го изтеглите от https://pypi.python.org/pypi/fastimport/.

Под Windows, `bzr-fastimport` се инсталира автоматично със standalone версията и инсталацията по подразбиране (изберете всички чекбоксове).

На този етап начините за импортиране на Bazaar хранилище се различават според това дали имате само един клон или не.

Проект с единичен клон

Влезте в директорията, съдържаща Bazaar хранилището и инициализирайте Git хранилище:

```
$ cd /path/to/the/bzr/repository
$ git init
```

След това можете просто да експортирате Bazaar хранилището и да го конвертирате в Git така:

```
$ bzr fast-export --plain . | git fast-import
```

Според размера на проекта, процесът може да отнеме секунди или няколко минути.

Проект с главен клон и работен клон

Можете също да импортирате Bazaar хранилище съдържащо клонове. Да кажем, че имате два клона: един главен (myProject.trunk) и един текущ (myProject.work).

```
$ ls
myProject.trunk myProject.work
```

Създайте Git хранилище и влезте в него:

```
$ git init git-repo
$ cd git-repo
```

Изтеглете `master` клона в Git:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject.trunk | \
git fast-import --export-marks=../marks.git
```

Направете същото и за работния клон:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

Сега `git branch` показва `master` и `work` клонове. Проверете логовете за да се уверите, че те са изцяло импортирани и махнете файловете `marks.bzr` и `marks.git`.

Синхронизиране на индексната област

Колкото и клонове да имате и без значение от метода на импортиране, индексната ви област сега не е синхронизирана с `HEAD`. В случая при импортиране на повече от един клон, то това важи и за работната директория. Ситуацията се разрешава лесно с командата:

```
$ git reset --hard HEAD
```

Игнориране на файловете от .bzrignore

Нека сега да видим каква е ситуацията с игнорирането на файлове. Първото нещо за правене е да преименуваме `.bzrignore` в `.gitignore`. Ако файлът `.bzrignore` съдържа един или повече редове започващи с "!!" или "RE:", ще трябва да го коригирате и може би дори да създадете няколко `.gitignore` файлове с цел да игнорирате точно същото съдържание като Bazaar.

Последно ще създадем къмит, който съдържа тези промени за миграцията:

```
$ git mv .bzrignore .gitignore
$ # modify .gitignore if needed
$ git commit -am 'Migration from Bazaar to Git'
```

Изпращане на хранилището към сървъра

Сега можем да публикуваме импортираното хранилище в новия му дом:

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

Новото Git хранилище е готово за ползване.

Perforce

Следващата система, за която ще разгледаме процеса по импортиране на данни, е Perforce. Както видяхме по-рано, Git и Perforce могат да комуникират по два начина: посредством git-p4 и Perforce Git Fusion.

Perforce Git Fusion

Git Fusion прави процеса сравнително безболезнен. Просто задаваме настройките на проекта, мапинга на потребителите и клоновете в конфигурационен файл (виж Git Fusion) и клонираме хранилището. Git Fusion ни изработва резултат подобен на оригинално Git хранилище, което е готово да се публикува на Git сървър. Можем дори да използваме Perforce като Git хост, ако желаем това.

Git-p4

Git-p4 може също да работи като имортиращ инструмент. Като пример, ще импортираме проекта Jam от Perforce публичното депо. За да настроим клиента си, трябва да експортираме P4PORT environment променливата, така че да сочи към Perforce депото:

```
$ export P4PORT=public.perforce.com:1666
```

За да продължите примерните инструкции, се нуждаете от достъп до Perforce депо. Ще използваме публичното такова на адрес public.perforce.com, но може да експериментирате с всяко друго, до което имате достъп.

Изпълняваме командата `git p4 clone` за да импортираме проекта Jam от Perforce сървъра, подавайки ѝ като аргументи депото с пътя до проекта и пътя, в който искаме да го импортираме:

```
$ 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%)
```

Този проект има само един клон, но ако имаме клонове конфигурирани с branch изгледи (или само множество от директории), може да използваме флага `--detect-branches` за да инструктираме `git p4 clone` да импортира всички клонове на проекта. Вижте Клонове за повече подробности.

На този етап сме почти готови. Ако влезем в директорията `p4import` и изпълним `git log`, можем да видим импортираната работа:

```
$ 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]
```

Може да забележите как `git-p4` е оставила идентификатор във всяко къмит съобщение. Добре е той да се запази, в случай че по-късно се наложи да се обърнем към Perforce change number-а по някаква причина. Обаче, ако искаме да махнем идентификатора, сега е времето за това — преди да започнем работа по новото хранилище. За целта използваме командата `git filter-branch`:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Ако пуснем `git log`, ще видим как всички SHA-1 чексуми за къмитите са се променили, но `git-p4` стринговете вече отсъстват от къмит съобщенията:

```
$ 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).
```

Сега импортираното хранилище е готово да се публикува на Git сървър.

Потребителски импортиращ инструмент

Ако системата ви не е сред дотук разгледаните, може да потърсите импортиращ инструмент в Интернет — качествени такива са налични за CVS, Clear Case, Visual Source Safe, дори за директория от архиви. Ако никой от тях не работи в конкретния случай или се нуждаете от по-специфичен процес на импортиране, тогава може да използвате `git fast-import`. Тази команда чете прости инструкции от стандартния вход за да записва специфични Git данни. Много по-лесно е да създавате Git обекти по този начин, вместо да използвате raw Git командите или да се опитвате да записвате raw обекти (вижте Git на ниско ниво за повече информация). По този начин можете да напишете собствен импортиращ скрипт, който чете необходимата информация от системата, от която импортирате и печата последователно инструкции към стандартния изход. Можете да пуснете програмата и да пренасочите изхода ѝ към `git fast-import`.

За демонстрация, ще напишем прост импортиращ инструмент. Да приемем, че работите в директория `current`, редовно архивирате проекта си в отделни директории именувани `back_YYYY_MM_DD` според датата, и в един момент решавате да импортирате всичко това в Git хранилище. Структурата на директориите изглежда така:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

За да импортираме Git директория, трябва да разгледаме как Git съхранява данните си. Както може би помните, Git грубо казано е свързан списък от къмит обекти, които сочат към snapshot-и от съдържание. Всичко, което трябва да укажете на `fast-import` е какво са snapshot-ите със съдържание, какви къмит данни сочат към тях и реда, по който идват. Стратегията е да минаваме по snapshot-ите един след друг и да създаваме къмити със съдържанието на всяка директория свързвайки всеки къмит с предишния.

Както го направихме в Примерна Git-Enforced политика, ще пишем скрипта си на Ruby. Бихте могли да използвате езика, с който вие се чувствате комфортно — просто трябва да печатате съответната информация на `stdout`. Освен това, ако сте под Windows, трябва да внимавате да не вмъквате carriage returns символи в края на редовете — `git fast-import` е чувствителен за това и очаква само line feeds (LF) а не carriage return line feeds (CRLF), което нормално се случва под Windows.

За начало, влизаме в съответната директория и идентифицираме всяка поддиректория, която ще бъде snapshot, който да импортираме. След това, печатаме командите, необходими за експорта. Основният цикъл изглежда по такъв начин:

```
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
```

Изпълняваме `print_export` от всяка директория, което взема манифеста и маркировката на предишния snapshot и връща манифеста и маркировката на текущия. По този начин можем да ги свържем коректно. Маркировката ("Mark") във `fast-import` е термин за идентификатор, който давате на къмит. Когато създавате къмити, вие давате на всеки от тях маркировка, която може да се ползва за свързване към него от други къмити. Така първото нещо, което трябва да направи метода `print_export`, е да генерира маркировка от името на директорията:

```
mark = convert_dir_to_mark(dir)
```

Ще направим това създавайки масив от директориите и ще използваме индексите като маркировки, защото те трябва да са цели числа. Методът изглежда така:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Сега имаме целочислено представяне на къмита и ни трябва дата за неговите метаданни. Понеже казахме, че датата е отразена в името на директорията, ще я извлечем оттам. Следващият ред от `print_export` файла е:

```
date = convert_dir_to_date(dir)
```

където `convert_dir_to_date` се дефинира като:

```
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
```

Това връща целочислена стойност за датата на всяка директория. Последната необходима част за метаданните е информация за автора, която ще хардкоднем в глобална променлива:

```
$author = 'John Doe <john@example.com>'
```

Сега сме готови да започнем печатането на къмит данните за нашия importer. Началните данни твърдят, че създаваме къмит обект с данни за това в какъв клон е той, следвани от генерираната маркировка, информацията за автора и къмит съобщението и след това — предишния къмит, ако има такъв. Кодът изглежда така:

```
# 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
```

Тайм зоната е твърдо зададена на (-0700) за улеснение. Ако импортираме от друга система, трябва да укажем тайм зоната като отместване. Къмит съобщението трябва да се представи в специален формат:

```
data (size)\n(contents)
```

Форматът представлява последователност от думата data, размерът на данните за прочитане, нов ред, и накрая, самите данни. Тук ще използваме helper метод наречен `export_data`, защото по-късно се нуждаем от същия формат за указване на съдържанието на файловете.

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Остана да укажем файловото съдържание на всеки snapshot. Това е лесно, защото имаме всеки един в директория — можете да отпечатате командата `deleteall` последвана от

съдържанието на всеки файл в директорията. Git след това ще запише съответно всеки snapshot:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Заб.: Понеже много системи представят версиите си като промени между два къмита, fast-import може също така да приема команди с всеки къмит, които да указват кои файлове са били добавени, премахнати или модифицирани и какво е новото съдържание. Можете да изчислите разликите между snapshot-ите и да подадете само тези данни, но това е по-сложно и може да оставите на Git да го свърши като просто подадете всички данни. Ако все пак искате това да е ваша работа, погледнете документацията на `fast-import` за повече подробности как точно да я извършите.

Форматът за подаване на ново файлово съдържание или за модифицирано такова е както следва:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Тук, 644 е режимът за файла (ако имате изпълними такива, трябва да ги установите и да ги подадете като 755), а inline казва, че ще предоставите съдържанието веднага след този ред. Методът `inline_data` изглежда така:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Използваме метода `export_data` дефиниран по-рано, понеже форматът е като за данните на къмит съобщенията.

Последното нещо, което трябва да сторим е да върнем текущата маркировка, така че тя да бъде изпратена към следващата итерация:

```
return mark
```

Под Windows трябва да добавите допълнителна стъпка. Както вече казахме, Windows използва CRLF за символите за край на ред, докато `git fast-import` очаква само LF. За да избегнете проблем, ще трябва да укажете на ruby да използва LF вместо CRLF:

```
$stdout.binmode
```

Това е. Ето целия скрипт:

```
#!/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

# 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
```

Ако го изпълним, получаваме съдържание подобно на това:

```
$ 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
(...)
```

За да стартираме importer-а, пренасочваме изхода през `git fast-import` докато сме в Git директорията, в която искаме да импортираме. Може да създадем нова директория, да изпълним `git init` в нея за начална точка и да пуснем скрипта:

```
$ 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
---------------------------------------------------------------------
```

Както се вижда, при успешен завършек получавате подробна статистика за извършените дейности. В този случай сме импортирали 13 обекта за 4 къмита в 1 клон. Сега може да изпълним `git log` за да видим новополучената история:

```
$ 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
```

Получавате чисто ново Git хранилище. Важно е да отбележим, че нищо не е извлечено на този етап — отначало работната директория е празна. За да си получим файловете, трябва да върнем клона си там където е `master` в момента:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Можете да правите още много неща с `fast-import` инструмента — да обработвате различни режими, двоични данни, множество клонове и сливане, тагове, индикатори за прогрес и т.н. Има много примери за по-сложни сценарии в директорията `contrib/fast-import` в сорс кода на Git.

Обобщение

Сега би трябвало да се чувствате комфортно използвайки Git като клиент с други version-control системи или импортирайки почти всяко налично хранилище в Git без да губите данни. В следващата глава ще разгледаме механизмите на Git на по-ниско ниво, така че ако е необходимо да можете да пипнете и последната подробност, която искате да настроите.

Git на ниско ниво

Може да сте стигнали до тази глава прескачайки директно от много по-ранна такава, а може и да сте изчели последователно всичко до тук — и в двата случая стигаме до мястото, в което ще разгледаме как работят вътрешните механизми на Git. Като автори на книгата сме на мнение, че информацията в следващите секции е фундаментално важна, ако искате да оцените изцяло колко полезен и мощен инструмент е Git. От друга страна обаче стоят опасенията, че тази информация може да е смущаваща и ненужно сложна за начинаещите. По тази причина решихме, че е добре главата да е последна в книгата и оставяме на читателя да прецени кога да я прочете.

След като сме стигнали дотук, нека започнем. Първо, да припомним отново, че Git фундаментално представлява вид файлова система, позволяваща адресиране на съдържание, с VCS интерфейс написан върху нея. Ще научим повече за това след малко.

В ранните версии на Git (преди версия 1.5), потребителският интерфейс беше значително по-сложен, понеже ударението беше върху функционалностите на файловата система, вместо върху една по-полирана VCS. В последните няколко години потребителският интерфейс претърпя значителни подобрения, сега е много изчистен и лесен за ползване, въпреки че старите стереотипни схващания за ранните трудни и сложни за овладяване UI версии, все още витаят наоколо.

Content-addressable filesystem слоят на Git е изключително впечатляващ аспект от системата, така че ще започнем с него, а след това ще разгледаме транспортните механизми и възможностите за поддръжка на хранилищата, с които евентуално бихте имали нужда да работите.

Plumbing и Porcelain команди

В книгата дотук използвахме около 30 на брой подкоманди на Git като `checkout`, `branch`, `remote`, и т.н. Но понеже Git отначало беше само набор от инструменти за контрол на версиите, вместо пълнофункционална user-friendly VCS, тя разполага с много подкоманди, които вършат работата на ниско ниво и са проектирани да се използват взаимно в UNIX-стил или да бъдат извиквани от скриптове. Тези команди от по-ниско ниво в Git се наричат "plumbing" команди, докато по-потребителски ориентираните са получили наименованието "porcelain" команди.

Както виждате, в деветте глави дотук, работихме почти само с porcelain командите. В тази глава обаче, акцентът ще е върху plumbing командите, които дават достъп до вътрешните Git механизми и демонстрират как и защо Git се справя със задачите си. Много от тези команди не са предназначени за ръчно използване от командния ред, вместо това те са подходящи за градивни елементи в множество странични инструменти и специфични скриптове.

Изпълнявайки `git init` в дадена директория, Git създава поддиректорията `.git`, която пази почти всичко, което системата съхранява и манипулира. Ако искате да архивирате или клонирате вашето хранилище, то копирането на тази единична директория ви дава почти всичко необходимо за целта. Цялата тази глава от книгата се занимава със съдържанието

на въпросната директория. Ето как изглежда една току що инициализирана `.git` директория:

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

В зависимост от версията на Git, може да видите и допълнителни неща тук, но в общи линии това е прясно `git init` хранилище. Файлът `description` се използва само от програмата GitWeb, не ни засяга в момента. Файлът `config` пази конфигурационните опции за конкретния проект, а директорията `info` съдържа глобалния exclude файл за игнориране на пътища, които не искате да вмъквате в `.gitignore` файл. Директорията `hooks` съхранява клиентските и сървърни hook скриптове, на които обърнахме внимание в Git Hooks.

Остават четири важни елемента: `HEAD` и (все още липсващите) `index` файлове и директориите `objects` и `refs`. Именно тези елементи са в ядрото на Git. Директорията `objects` пази цялото съдържание на вашата база данни, директорията `refs` съхранява указатели в къмит обекти в тези данни (клонове, тагове, remotes и др.). Файлът `HEAD` сочи към клона, съдържанието на който текущо е извлечено в работната директория, а файлът `index` е мястото, където Git пази информацията от индексната област. Сега ще разгледаме всеки от тези елементи в детайли.

Git обекти

Git е content-addressable файлова система. Супер. Какво означава това? Това означава, че по същество Git е просто склад за данни от типа key-value (ключ-стойност). Което от своя страна значи, че можете да вмъкнете произволен тип съдържание в Git хранилище и ще получите за него уникален идентификатор, който можете по-късно да използвате, за да извлечете съдържанието обратно.

За демонстрация, нека погледнем plumbing командата `git hash-object`, която приема някакви данни, съхранява ги в директорията `.git/objects` (*базата данни с обекти*) и ви връща уникалния ключ, сочещ към този информационен обект.

Първо, инициализираме ново Git хранилище и проверяваме, че в директорията `objects` няма нищо:

```
$ 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 създава тази директория `objects` заедно с поддиректориите ѝ `pack` и `info`, но освен тях няма никакви нормални файлове. Нека сега изпълним `git hash-object` за да създадем и запишем в базата данни нов data обект ръчно:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

В най-простата си форма, `git hash-object` ще вземе данните, които сте ѝ подали и само ще върне уникалния ключ. който *ще бъде използван* за съхранението им в базата данни. Флагът `-w` казва на командата не само да върне ключа, но и да запише обекта в базата. Последно, `--stdin` инструктира `git hash-object` да вземе съдържанието, което ще обработва от stdin, в противен случай командата ще очаква като аргумент име на файла със съответното съдържание.

Изходът на екрана е 40-символен стринг. Това е SHA-1 хешът — чексума на съдържанието, което съхранявате плюс един хедър, за който ще научим по-късно. Сега може да видим как Git е съхранил данните ни:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

В директорията ни `objects` вече съществува файл за новото ни съдържание. Първоначално Git запазва данните ни по този начин — като единичен файл за частичка съдържание, с наименование SHA-1 чексумата на съдържанието и хедъра. Поддиректорията (`d6`) се именува с първите два символа от SHA-1 чексумата, а името на файла се формира от останалите 38 символа.

Веднъж записали съдържание в базата данни, можете да го изследвате с командата `git cat-file`. Тази команда е като швейцарско ножче за инспектиране на Git обекти. Параметърът `-p` инструктира командата първо да определи вида на съдържанието и след това да го покаже по съответния начин:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Сега можете да записвате и извличате обратно текстово съдържание в Git. Може да правите

това и със съдържанието на файлове. Например, можете да направите прост контрол на версиите за файл. Първо създаваме нов файл и записваме съдържанието му в базата данни:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

След това, записваме нови данни във файла и го записаме отново:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Сега базата с обекти съдържа и двете версии на този нов файл (първоначалното текстово съдържание също е тук):

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Сега можете спокойно да изтриете локалното копие на файла `test.txt`, а след това да използвате Git за да извлечете коя да е от версиите му от базата данни:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

ако искате втората версия:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Но запомнянето на SHA-1 ключа за всяка версия на файла едва ли е практично и освен това вие не пазите името на файла, а само съдържанието му. Този тип обект се нарича *blob*. Можете да кажете на Git да ви извлече типа на всеки обект по дадена SHA-1 стойност с `git cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree обекти

Следващият тип обект в Git е *tree*, който решава проблема със записа на името на файла и също така позволява да запазвате група файлове заедно. Git пази съдържанието по маниер подобен на UNIX файлова система, но една идея по-опростено. Цялото съдържание се съхранява под формата на tree и blob обекти, като дърветата играят ролята на съответните UNIX директории и blob-обектите съответстват малко или много на inodes или файлово съдържание. Единичен tree обект съдържа един или повече tree елемента, всеки от които съдържа SHA-1 указател към blob или поддърво със съответните име на файл, режим и тип. Например, най-новото дърво в проект може да изглежда по такъв начин:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

Синтаксисът `master^{tree}` указва tree обекта, към който сочи последния къмит в `master` клона. Забелязваме, че `lib` поддиректорията не е blob, а указател към друго дърво:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

В зависимост от шела, който използвате, може да срещнете грешки при използване на синтаксиса `master^{tree}`.

В CMD под Windows, символът `^` се използва за escaping, така че трябва да го въведете два пъти: `git cat-file -p master^^{tree}`. Когато използвате PowerShell, параметрите съдържащи символите {} трябва да се оградят в кавички, за да не се интерпретират погрешно: `git cat-file -p 'master^{tree}'`.

Ако имате ZSH, тогава символът `^` се използва за globbing, така че трябва да оградите целия израз с кавички: `git cat-file -p "master^{tree}"`

Концептуално, данните които Git запазва изглеждат така:

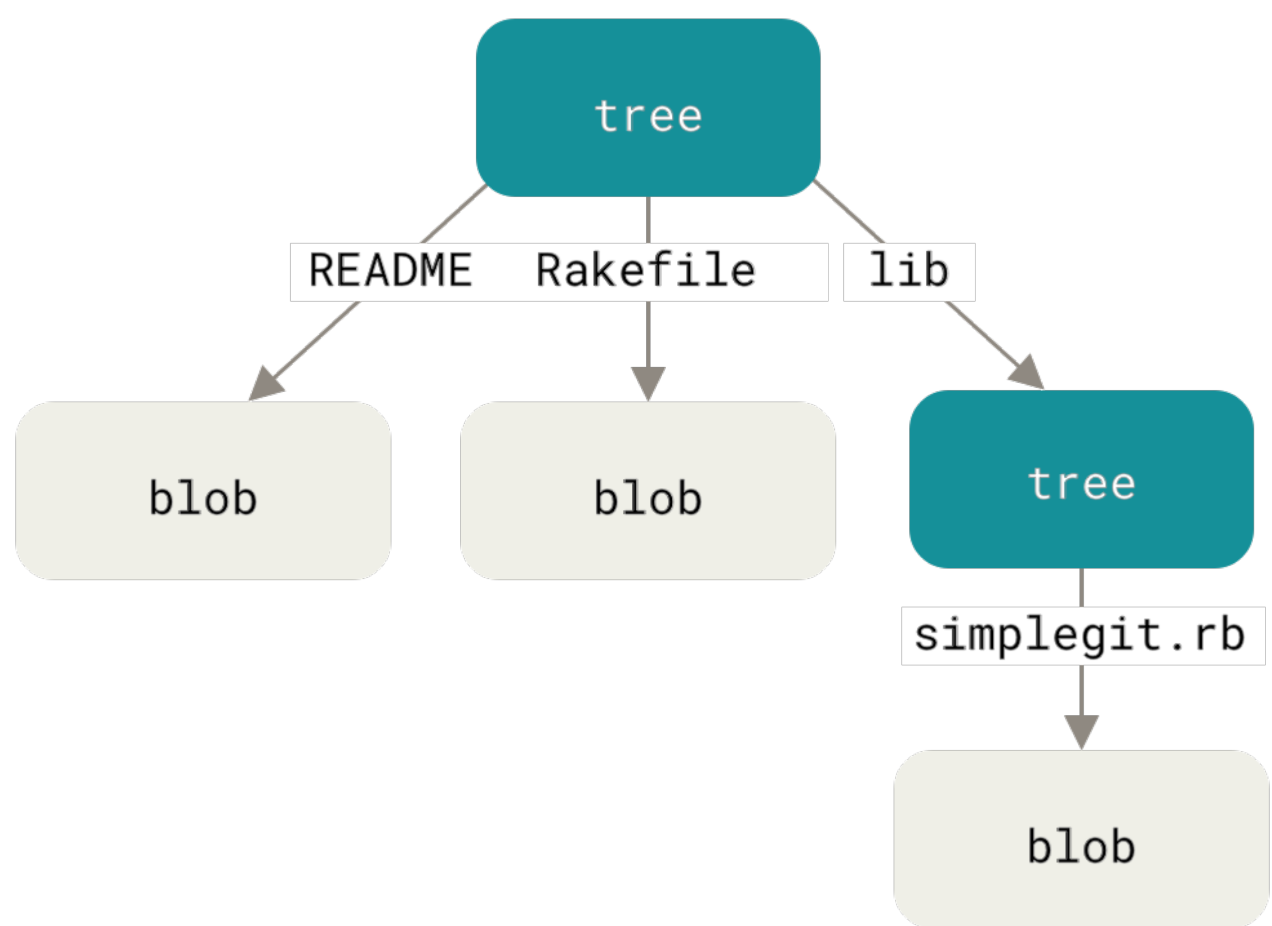


Figure 147. Проста версия на data модела на Git

Можете сравнително лесно да създадете собствено дърво. Git нормално създава дърво вземайки статуса на индексната област и записвайки серия от tree обекти от нея. Така, за да създадете tree обект, първо трябва да инициализирате индекса с някакви файлове. За да създадете индекс с един елемент — първата версия на файла `test.txt`, може да използвате plumbing командата `git update-index`. Използвайки я, добавяте изкуствено най-ранната версия на файла към нова индексна област. Трябва да подадете аргумента `--add`, защото файлът все още не съществува в индекса (и дори самият индекс още не съществува) и също `--cacheinfo`, защото файлът, който добавяте не е в работната директория, а е в базата данни. След това подавате режима, SHA-1 стойността и името на файла:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

В този случай указваме режим `100644`, което ще рече, че това е обикновен файл. Другите опции са `100755` за изпълним файл и `120000`, който указва символна връзка (symlink). Режимът се взема съобразно стандартните UNIX правила, но е много по-малко гъвкав — това са единствените три режима, които са валидни за файлове (blobs) в Git (въпреки че други режими се използват за директории и подмодули).

Сега може да използвате `git write-tree` за да запишете индексната област в tree обект. Не се изисква флаг `-w` — изпълнението на тази команда автоматично създава tree обект съобразно статуса на индекса, ако такова дърво вече не съществува:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Може да проверите, че това е tree обект със същата `git cat-file` команда от по-рано:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Сега ще създадем ново дърво с втората версия на `test.txt` файла и също така още един нов файл:

```
$ echo 'new file' > new.txt
$ git update-index --add --cacheinfo 100644 \
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index --add new.txt
```

Сега индексът ни има новата версия на `test.txt`, както и новия файл `new.txt`. Записваме това дърво в tree обект и поглеждаме как изглежда:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Забелязваме, че сега това дърво съдържа и двата файла, и че SHA-1 стойността на `test.txt` вече е “version 2” от по-рано (`1f7a7a`). Само за идеята, ще добавим първото дърво като поддиректория в това. Може да прочитаме дървета от индекса с командата `git read-tree`. В този случай, може да прочетем дърво от индекса като поддърво с параметъра `--prefix` към командата:

```
$ 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
```

Ако създадем работна директория от новозаписаното дърво, ще получим в нея двата файла в корена ѝ и поддиректория `bak`, в която е първата версия на файла `test.txt`. Може да пресъздадем данните, които Git пази за тези структури, така:

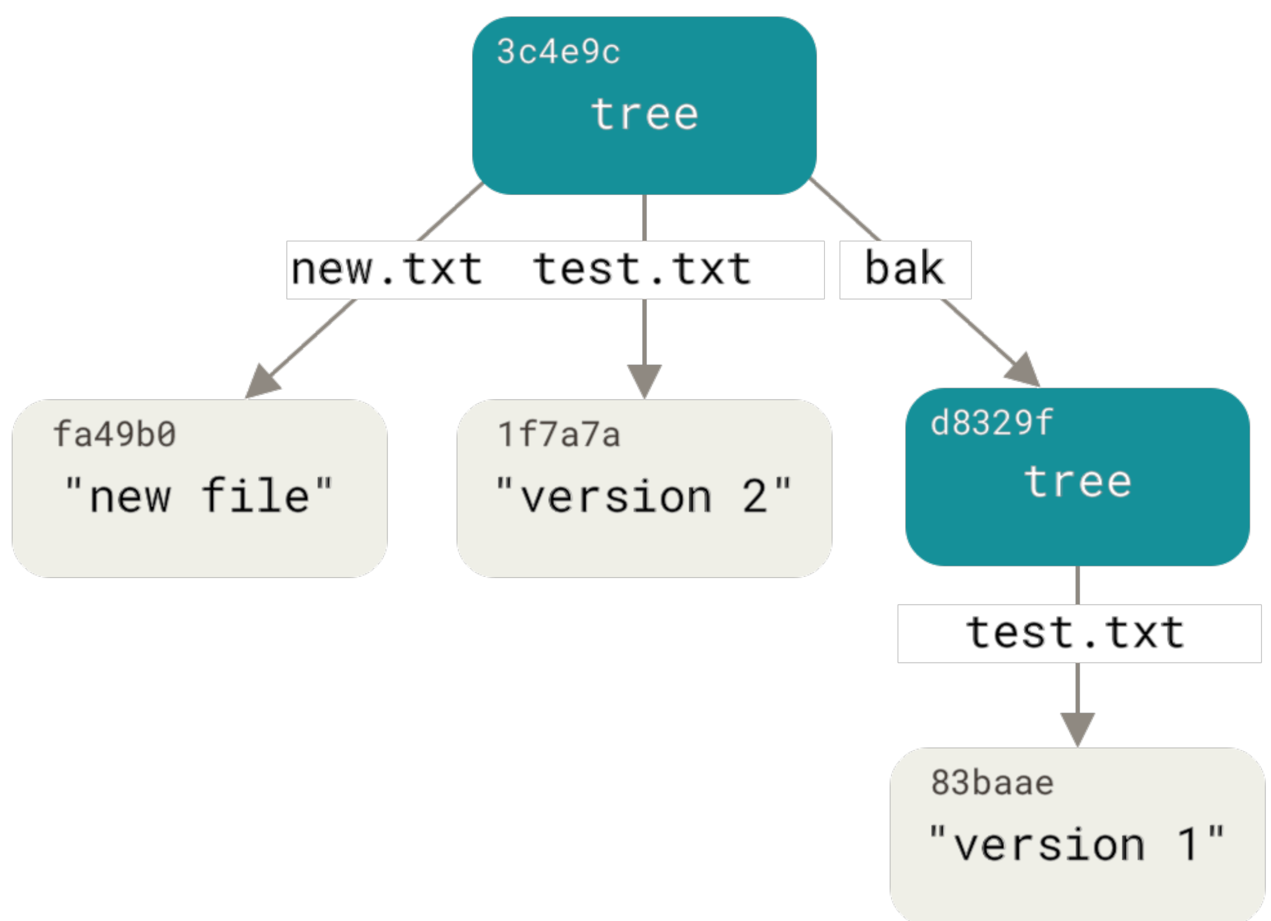


Figure 148. Структура на съдържанието на текущите Git данни

Commit обекти

Ако сте изпълнили всички стъпки досега, разполагате с три дървета, които представят различните snapshot-и на проекта, който искате да следите. Но проблемът все още стои: трябва да помните всичките три SHA-1 стойности, за да извлечете съдържанието им. Също така, нямате никаква информация за това кой е съхранил тези snapshot-и, кога са записани или защо са записани. Това е информацията, която съхраняват къмит обектите.

За да създадем такъв обект, използваме командата `commit-tree`, която очаква като аргументи SHA-1 хеша на единично дърво и също така, кой къмит обект (ако има такъв) директно го предшества. Започваме с първото съхранено дърво:

```
$ echo 'First commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Получаваме различна хеш стойност поради различното време на създаване на обекта и данните за автора. Освен това, въпреки че по принцип всеки къмит обект може да се пресъздаде прецизно с тези данни, хронологичните особености на конструкцията на тази книга означават, че отпечатаните къмит хешове може да не съответстват на дадените къмити. По-натам в главата, замествайте commit и tag хешовете с вашите собствени чексуми.

Сега можем да разгледаме новия ни къмит обект с командата `git 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
```

Форматът е прост: той указва top-level дървото за snapshot-а на проекта в този момент; родителските къмити, ако съществуват такива (описаният отгоре обект няма никакви родители); author/committer информацията (ще се използват текущите конфигурационни настройки за `user.name` и `user.email` и timestamp); следва празен ред и накрая е къмит съобщението.

Следва да запишем другите два къмит обекта, всеки от които сочи към къмита дошъл директно преди него:

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'Third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Всеки от трите къмит обекта сочи към едно от трите snapshot дървета, които създадохме. Сега вече имаме и реална Git история, която може да видим с `git log`, ако я пуснем за SHA-1 стойността на последния къмит:

```
$ 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(+)
```

Чудесно. Току що извършихме операции от ниско ниво за да си изградим Git история без да използваме нито една porcelain команда. По същество това прави Git, когато изпълните `git add` и `git commit` — записва blob обекти за променените файлове, обновява индекса, записва дървета и записва къмит обекти, които сочат към top-level дърветата и къмитите дошли непосредствено преди тях. Тези три основни Git обекта — blob, tree, и commit, първоначално се съхраняват като отделни файлове в директорията `.git/objects`. Ето всички обекти в примерната ни директория с коментар за това, което съхраняват:

```
$ 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
```

Ако проследим всички вътрешни указатели, получаваме графика от този вид:

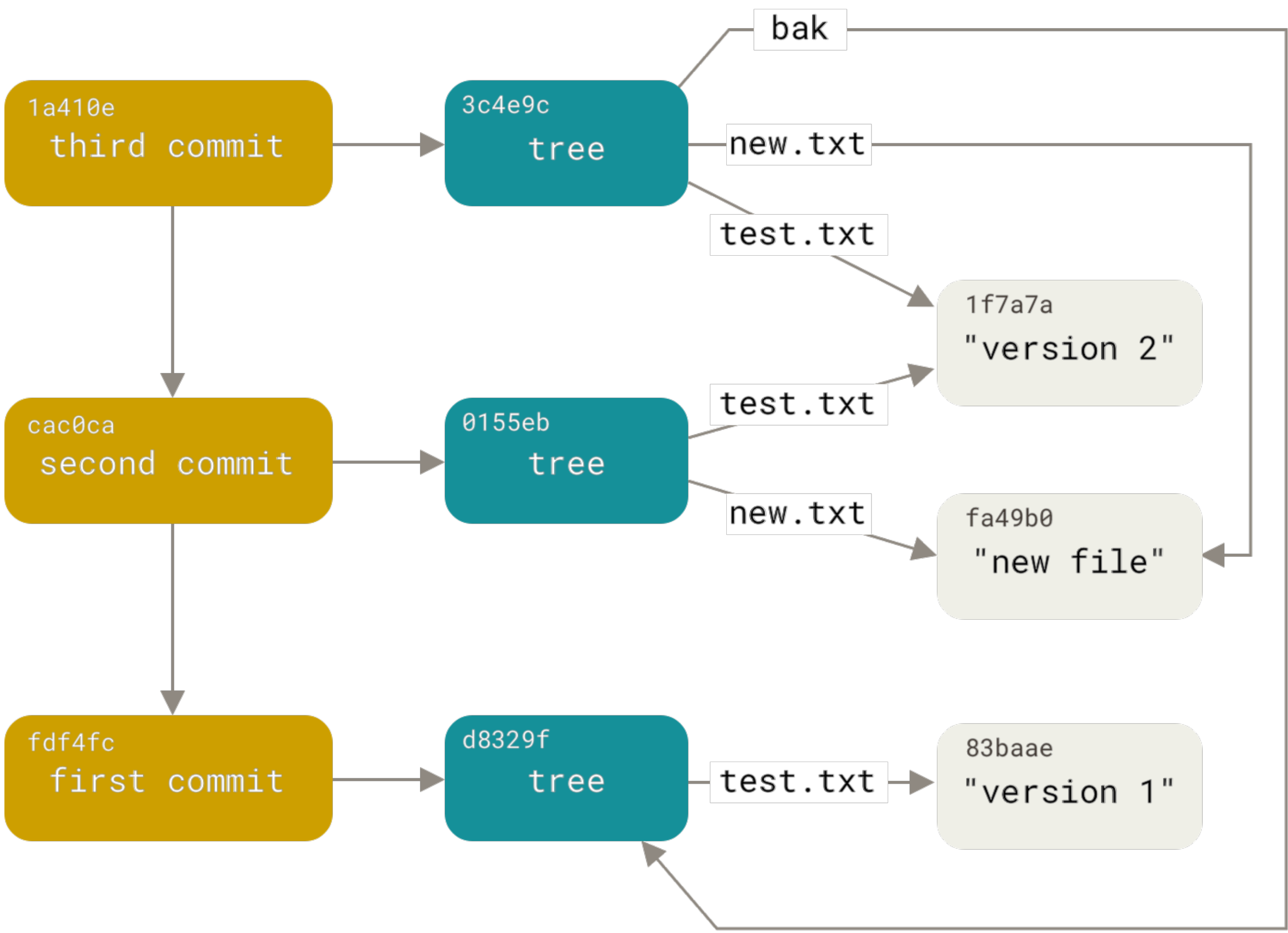


Figure 149. Всички достъпни обекти в Git директорията

Съхранение на обектите

По-рано казахме, че с всеки обект, който къмитваме в базата данни се пази и по един хедър. Нека видим как Git съхранява обектите си. Ще видим как да съхраним blob обект, в този случай стринга “what is up, doc?” — интерактивно в Ruby scripting езика.

Може да стартирате интерактивния Ruby режим с `irb` командата:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git първо конструира хедър, който започва с идентифициране на типа на обекта — в този случай blob. Към тази първа част от хедъра, Git добавя интервал последван от размера на съдържанието в байтове и финален null байт:

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\u0000"
```

Git конкатенира хедъра и оригиналното съдържание, след което калкулира нова SHA-1 стойност на резултата. В Ruby, можете да калкулирате SHA-1 стойност на стринг като инклуднете SHA1 digest библиотеката с `require` команда и след това викайки метода `Digest::SHA1.hexdigest()` със стринга:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Нека сравним това с изхода от `git hash-object`. Тук използваме `echo -n` за да избегнем добавянето на нов ред към входа.

```
$ echo -n "what is up, doc?" | git hash-object --stdin
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Git компресира новото съдържание със zlib, което може да направите в Ruby с библиотеката zlib. Първо инклудвате библиотеката и след това изпълнявате `Zlib::Deflate.deflate()` за съдържанието:

```
>> 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"
```

Последно, записате zlib-deflated съдържанието в обект на диска. Установявате пътя за обекта (първите два символа от SHA-1 стойността са името на поддиректорията, останалите 38 са името на файла в нея). В Ruby може да използвате функцията `FileUtils.mkdir_p()` за да създадете директорията, ако тя не съществува. След това, отворете файла с `File.open()` и запишете в него zlib-компресираното съдържание с `write()` повикване към получения файлов указател:

```
>> 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
```

Нека проверим съдържанието на обекта с `git cat-file`:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
---
```

Това е, вече създадохте валиден Git blob обект.

Всички Git обекти се записват по същия начин, само с различни типове — вместо със стринга blob, хедърът ще започва с commit или tree. Също така, въпреки че blob съдържанието може да е практически всякакво, то commit и tree съдържанията се форматират строго специфично.

Git референции

Ако искате да видите историята на хранилището си достъпна през къмит `1a410e` например, може да изпълните нещо като `git log 1a410e`, но все пак трябва да сте запомнили, че именно `1a410e` е къмитът, който ви интересува. Вместо това, би било по-лесно ако имахте файл, в който да съхраните тази SHA-1 стойност под някакво смислено име и след това да използвате това име като изходна точка.

В Git тези опростени имена се наричат “references” или просто “refs” и може да намерите файловете, които ги съхраняват в директорията `.git/refs`. В текущия ни проект тази директория не съдържа файлове, но съдържа проста структура:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

За да създадете проста референция, може да направите това:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

Сега можете да използвате head референцията, която току що създадохте вместо SHA-1 стойността в Git командите:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Не се препоръчва тези файлове да бъдат редактирани ръчно, вместо това Git предоставя по-безопасната команда `git update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Това в общи линии е един клон в Git: прост указател към head на линия работа. За да създадете клон от втория къмит:

```
$ git update-ref refs/heads/test cac0ca
```

Този клон сега ще съдържа само работата от този къмит назад:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Сега Git базата данни концептуално изглежда така:

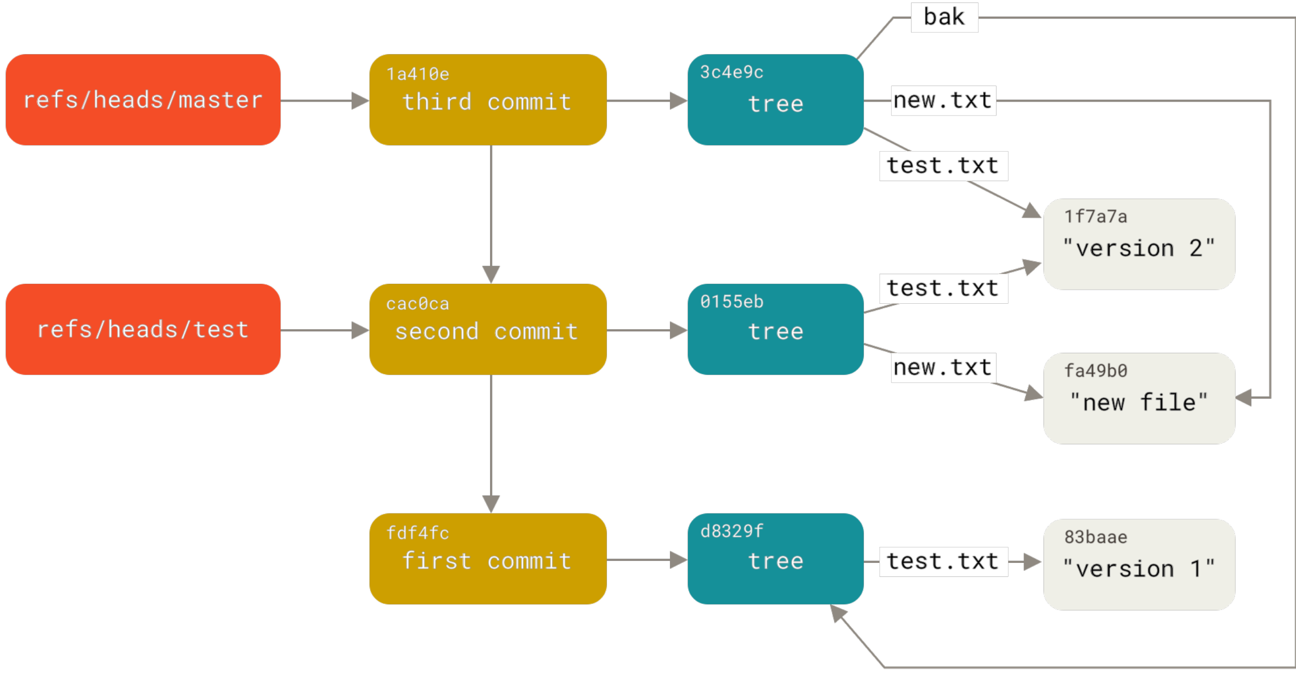


Figure 150. Обекти в Git директорията с включени head референции за клонове

Когато изпълнявате команда като `git branch <branch>`, Git всъщност изпълнява `update-ref` за да добави SHA-1 чексумата на последния къмит на текущия клон в референцията, която искате да създадете.

HEAD

Сега изниква въпроса как при изпълнение на `git branch <branch>` Git знае коя е SHA-1 стойността на последния къмит? Отговорът е файла HEAD.

HEAD файлът е symbolic референция към текущия клон. Под symbolic референция се има предвид, че за разлика от нормалните референции, тя не съдържа SHA-1 стойност, а вместо това указател към друга референция.

В някои редки случаи обаче, HEAD файлът все пак може да съдържа SHA-1 стойност на git обект. Това се случва при извличане на таг, къмит или отдалечен клон, при което хранилището ви попада в "detached HEAD" състояние.

Ако погледнете файла, обикновено виждате нещо такова:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Ако изпълним `git checkout test`, Git обновява файла така:

```
$ cat .git/HEAD
ref: refs/heads/test
```

При изпълнение на `git commit` се създава къмит обект и се указва, че родителят на този къмит обект съответства на SHA-1 стойността, към която сочи референцията в HEAD.

Можете и ръчно да редактирате този файл, но и тук съществува по-безопасна алтернатива под формата на командата `git symbolic-ref`. Можете да прочетете стойността на HEAD така:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Със същата команда можете и да я промените:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Не можете да направите symbolic референция извън стила на refs:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Тагове

Току що разгледахме трите основни обектни типа в Git (*blobs*, *trees* и *commits*), но има и още един. Обектът *tag* е много подобен на commit обекта — съдържа информация за човека създал тага, дата, указател и съобщение. Основната разлика е, че таг обекта сочи към къмит, вместо към дърво. Това е подобно на branch референция, но никога не се премества — винаги сочи към един и същи къмит, но му дава по-информативно име.

Както видяхме в Основи на Git, има два вида тагове: annotated и lightweight. Можете да направите lightweight таг така:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Това е lightweight тагът — референция, която не се мести. Annotated таговете, обаче, са малко по-сложни. Ако създадете annotated таг, Git създава таг обект и след това създава референция сочеща към него, вместо директно към къмита. Може да видите това създавайки annotated таг (с опцията `-a`):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

Ето SHA-1 стойността на създадения обект:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Сега, изпълнете `git cat-file -p` върху тази стойност:

```
$ 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
```

Отбележете как реда object сочи към SHA-1 стойността на къмита, който беше тагнат. Също така отбележете, че не е необходимо да сочи към къмит, можете да тагвате всеки Git обект. В сорс кода на Git например, поддържащият проекта разработчик е добавил своя GPG public key като blob обект и след това го е тагнал. Можете да видите публичния ключ изпълнявайки следното в клонирано хранилище:

```
$ git cat-file blob junio-gpg-pub
```

В хранилището на Linux ядрото също има таг обект, който не сочи към къмит — първият създаден таг сочи към първоначалното дърво на импорта на сорс кода.

Remotes

Третият тип референции са remote референциите. Ако добавите remote и публикувате към него, Git записва последната публикувана стойност за всеки клон в директорията `refs/remotes`. Можете да добавите remote `origin` и да публикувате `master` клона в него:

```
$ 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
```

След това, можете да видите стойността за `master` клона в отдалечената `origin` референция последния път, когато сте комуникирали със сървъра във файла `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote референциите се отличават от клоновете (`refs/heads` референциите) основно по това, че се третират като read-only. Можете да изпълните `git checkout` към такава референция, но Git няма да насочи HEAD към нея, така че никога няма да я обновите с `commit` команда. Git ги управлява като bookmarks към последния известен статус на клоновете им в съответните сървъри.

Packfiles

Ако сте следвали инструкциите от предната секция сега трябва да имате Git хранилище с 11 обекта — 4 blob обекта, 3 tree обекта, 3 къмита, и 1 таг:

```
$ 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 ги компресира със zlib и резултатът не отнема много място, всички файлове заедно отнемат 925 байта. Сега ще добавим малко по-обемисто съдържание в хранилището, за да демонстрираме една интересна функционалност на Git. Добавяме файла `repo.rb` от библиотеката Grit — това е сорс код с размер около 22K:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'Create repo.rb'
[master 484a592] Create repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Поглеждайки в полученото дърво, можем да видим SHA-1 стойността за новия `repo.rb` blob обект:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Сега може да използваме `git cat-file` за да видим колко голям е обекта:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Променяме малко този файл и къмитваме:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'Modify repo.rb a bit'
[master 2431da6] Modify repo.rb a bit
 1 file changed, 1 insertion(+)
```

Проверяваме дървото създадено от този последен къмит и виждаме нещо интересно:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Обектът blob сега е различен, което значи, че въпреки добавения само един ред в края на един файл от 400 реда, Git съхранява новото съдържание като изцяло нов обект:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Имаме два почти идентични 22K обекта на диска (всеки компресиран до около 7K). Дали няма да е по-добре, ако Git можеше да съхрани единия от тях изцяло, а другия само като разлика между двата?

Оказва се, че може. Първоначалният формат, в който Git записва обектите на диска се нарича “loose” формат. Обаче, в даден момент Git пакетира набор такива обекти в единичен binary файл известен като “packfile” с цел да спести място и да е по-ефективен. Git прави това, ако имате твърде много loose обекти, ако изпълните ръчно командата `git gc` или ако публикувате към отдалечен сървър. За да видите какво се случва, може да изпълните `git gc` за да накарете Git ръчно да пакетира обектите си:

```
$ 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)
```

Ако сега погледнете в директорията `objects`, ще установите че повечето от обектите ви са изчезнали и се е появил чифт нови файлове:

```
$ 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
```

Останалите тук обекти са blob обектите, към които не сочи нито един къмит — в нашия случай съдържанието от примерите “what is up, doc?” и “test content”. Понеже никога не са добавяни в къмит, те се третират като висящи и не се пакетират в новия packfile.

Останалите файлове са новия packfile и един индекс. Този packfile е единичен файл съхраняващ съдържанието на всички обекти, които са изтрити от файловата система. Индекс файлът съдържа отстъпите на обектите в packfile файла, така че лесно да търсите специфичен обект. Ценното в случая е фактът, че преди да изпълните `gc` командата, размерът на обектите беше около 15K, докато новият packfile е само 7K. Пакетирането съкрати наполовина използваното дисково пространство.

Как Git прави това? В процеса на пакетиране, Git търси файлове с подобни имена и размери и съхранява само разликите от една версия на файла към друга. Полученият packfile може да се изследва, за да видим какво точно е направено от Git. Командата `git verify-pack` позволява да видим какво е било пакетирано:

```
$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Тук blob обектът `033b4`, който ако си спомняте беше първата версия на файла`repo.rb`, сочи към blob обекта `b042a`, който беше втората версия. Третата колона от изхода е размера на обекта в пакета. Така може да видите, че `b042a` заема 22K от файла, но `033b4` заема само 9 байта. Друг интересен факт е, че запазената изцяло версия на файла е втората такава, докато първата се пази само като разлика — това е защото се предполага, че най-вероятно ще се нуждаете от бърз достъп до най-актуалната му версия.

Наистина хубавото нещо във всичко това е, че по всяко време може да се направи ново пакетиране. Git от време на време автоматично препакетира базата данни като се опитва винаги да ви спести пространство. Но винаги можете да направите това и ръчно с командата `git gc`.

Refspec спецификации

В книгата дотук използвахме просто съпоставяне от отдалечени клонове към локални референции, но те могат да бъдат и по-сложни. Допускаме, че сте следвали последните няколко секции и сте създали малко Git хранилище, сега искате да добавите *remote* към него:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Изпълнението на командата отгоре добавя секция във файла `.git/config`, която указва името на този remote (`origin`), URL-а на отдалеченото хранилище и *refspec* спецификацията, която да се използва за изтегляне:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Форматът е символът `+` (който е опция), последван от `<src>:<dst>`, където `<src>` е израз за референциите от отдалечената страна на връзката и `<dst>` указва къде тези референции ще се проследяват локално. Символът `+` казва на Git да обновява референцията дори, когато тя не е fast-forward.

В случаите по подразбиране, това се записва автоматично от командата `git remote add origin`, Git издърпва всички референции от `refs/heads/` на сървъра и ги записва в `refs/remotes/origin/` локално. Ако на сървъра има `master` клон, можете да получите достъп до историята му локално с коя да е от следните команди:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Те са еквивалентни, защото Git ги разширява до `refs/remotes/origin/master`.

Ако искате Git да изтегля само `master` клона всеки път, а другите не, можете да промените fetch реда така:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Това е точно подразбиращата се refspec спецификация за `git fetch` за този remote. Ако искате да правите само еднократно изтегляне, можете да укажете специфичната refspec спецификация също и от командния ред. За да изтеглите клона `master` от сървъра локално в `origin/mymaster`, може да изпълните:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Може да указвате и множество refspecs. От командния ред изтегляте няколко клона така:

```
$ 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
```

В този случай, изтеглянето на `master` клона беше отказано, защото той не е посочен като fast-forward референция. Може да преодолеете това със символа `+` преди съответната refspec.

Може да укажете множество refspecs за издърпване и в конфигурационния файл. Ако винаги искате да теглите клоновете `master` и `experiment` от `origin` сървъра, добавете два реда:

```
[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
```

От Git 2.6.0 може да използвате частични globs в израза за търсене на повече клонове, така че това работи:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Дори по-добре, можете да използвате namespaces (или директории) за да направите същото по-структурирано. Ако имате QA екип, който публикува серии от клонове и желаете да получавате клона `master`, всеки от QA клоновете и нищо друго, използвайте подобна конфигурационна секция:

```
[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/*
```

Ако сте в сложен работен процес, при който има QA екип, разработчици и интеграционни екипи, които публикуват и сътрудничат по отдалечени клонове, можете по този начин по-лесно да ги поставяте в съответни namespaces.

Публикуване на Refspecs

Добре е, че можете да издъпвате namespaced референции по този начин, но как QA екипа поставя клоновете си в `qa/` namespace на първо време? Това се прави с помощта на push refspecs спецификации.

Ако QA екипът иска да публикува техния `master` клон в `qa/master` на сървъра, биха могли да изпълнят:

```
$ git push origin master:refs/heads/qa/master
```

Ако искат Git да прави това автоматично всеки път при `git push origin`, могат да добавят `push` елемент в конфигурационния си файл:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Така при `git push origin` локалният `master` клон ще се публикува в отдалечения `qa/master` по подразбиране.

Не можете да използвате refspec за да теглите от едно хранилище и да изпращте към друго. За пример как би могло да стане това, погледнете Актуализиране на вашето публично GitHub хранилище.

Изтриване на референции

Можете също да използвате refspec за да изтривате референции от отдалечен сървър така:

```
$ git push origin :topic
```

Понеже спецификацията е във формат `<src>:<dst>`, пропускането на `<src>` частта ще направи така, че `topic` клонът в сървъра да е нищо — което ефективно ще го изтрие.

Или може да използвате по-новия синтаксис (след Git v1.7.0):

```
$ git push origin --delete topic
```

Транспортни протоколи

Git може да обменя данни между две хранилища по два основни начина: с "dumb" протокола и със "smart" протокола. Тук ще видим накратко как работят те.

Dumb протокол

Ако настройвате хранилище само за четене през HTTP, тогава dumb протоколът е вероятния избор. Този вид протокол се нарича "dumb", защото не изисква Git-specific код от страна на сървъра по време на транспортния процес. Процесът е серия от HTTP `GET` заявки, при които клиентът може да предположи какво е разположението на Git хранилището в сървъра.

Този вид протокол вече се използва сравнително рядко. При него е трудно да се гарантира защитата на данните, така че повечето Git хостове (cloud-based и on-premises) ще откажат използването му. Препоръчително е да се използва smart протокола, който ще разгледаме след малко.

Нека преминем през `http-fetch` процеса за библиотеката simplegit:

```
$ git clone http://server/simplegit-progit.git
```

Първото нещо, което прави командата, е да изтегли файла `info/refs`. Този файл се записва от командата `update-server-info`, ето защо трябва да разрешите това като `post-receive` hook за да може HTTP транспорта да работи коректно:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Сега имате списък на отдалечените референции и SHA-1 стойности. След това поглеждаме към какво сочи референцията HEAD, така че да знаем какво да извлечем в работната директория, когато сме готови:

```
=> GET HEAD
ref: refs/heads/master
```

Това значи, че ще разпакетираме клона `master` в края. В този момент сме готови да пуснем процеса. Понеже изходната точка е къмит обекта `ca82a6`, който виждаме във файла `info/refs`, започваме с неговото изтегляне:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Получаваме обратно обект — този обект е в loose формат на сървъра и сме го изтеглили през статична HTTP GET заявка. Можете да използвате zlib за да го декомпресирате, да премахнете хедъра и да видите съдържанието на къмита:

```
$ 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

Change version number
```

Следва да се изтеглят още два обекта — `cfda3b`, което е дървото на съдържанието, към което сочи току що изтегления къмит, и родителския къмит `085bb3`:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Това ни дава следващия къмит обект. Изтегляме tree обекта:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Получаваме отговор 404 — изглежда, че tree обектът не е в loose формат на сървъра. За това може да има различни причини — обектът може да е в алтернативно хранилище или пък може да е в packfile в текущото. Git първо проверява за евентуални заместители:

```
=> GET objects/info/http-alternates
(empty file)
```

Ако получим списък от алтернативни URL-и, Git проверява за loose файлове и packfile файлове в тях — това е чудесен начин за проекти, които са forks на други да споделят обекти на диска. Само че, в този случай нямаме такъв списък, така че обектът трябва да е пакетиран в packfile. За да видим какви packfiles са налични на сървъра трябва да вземем файла `objects/info/packs`, който съдържа списък с тях (генерира се също от `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Има само един packfile на сървъра, така че обектът очевидно е там, но все пак проверяваме и в индекса за да се уверим в това. Такава проверка е полезна и ако имате повече packfiles на сървъра за да намерите кой точно от тях съдържа търсения обект:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Сега имаме packfile индекса и можем да видим дали обектът ни е посочен в него, индексът съдържа списък с SHA-1 чексумите и позициите (отместванията) на обектите от packfile файловете. Обектът ни е тук и следва да изтеглим целия packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Получихме tree обекта и можем да продължим стъпките през къмитите. Те също са в packfile пакета, който току що изтеглихме, така че не се налага да правим повече заявки към сървъра. Git извлича работно копие от клона `master`, към който сочеше HEAD референцията изтеглена в началото.

Протоколът Smart

Протоколът dumb е прост, но не много ефективен и освен това не може да обслужва изпращането на данни от клиента към сървъра. Протоколът smart е много по-подходящ за трансфер на данни, но изисква процес в отдалечената страна на връзката, който да е

запознат с Git — той трябва да може да прочита локалните данни, да определи какво има и от какво се нуждае клиента и да му генерира custom packfile като резултат. Съществуват две множества процеси за трансфер на данни: чифт за изпращане и още един чифт за приемане.

Изпращане на данни

За да изпраща данни към отдалечен процес, Git използва процесите `send-pack` и `receive-pack`. Процесът `send-pack`, очаквано, работи от страна на клиента и се свързва с `receive-pack` процеса в другия край на връзката.

SSH

За пример, да кажем, че изпълнявате `git push origin master` в проекта си и `origin` е дефинирана като URL, за който се използва SSH протокола. Git стартира `send-pack` процеса, който инициира конекция към сървъра през SSH. Този процес опитва да изпълни команда на сървъра през SSH повикване, което изглежда по подобен начин:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

Командата `git-receive-pack` незабавно отговаря с по един ред за всяка референция, която има — в този случай само клона `master` и неговата SHA-1 чексума. Първият ред също така подава списък с поддържаните от сървъра възможности (в случая `report-status`, `delete-refs`, и някои други, вкл. идентификация на клиента).

Данните се предават на части (chunks). Всяка част започва с 4-символна шестнайсетична стойност, която указва колко дълъг е остатъкът от нея (вкл. 4-те байта на стойността). Частите обикновено съдържат един ред данни следвани от символ за нов ред. Първата ви част започва с 00a5, което десетично е 165 и означава, че на този ред остават 165 байта. Следващият ред започва с 0000, което индикира, че сървърът няма какви други референции да показва.

След като вече знае статуса на сървъра, процесът `send-pack` определя локалните къмити, които не присъстват на него. За всяка референция, която това публикуване ще актуализира, процесът `send-pack` подава на `receive-pack` съответната информация. Например, ако обновявате `master` клона и добавяте клон `experiment`, `send-pack` отговорът може да изглежда така:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
\
    refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
\
    refs/heads/experiment
0000
```

Git изпраща по ред за всяка референция, която обновява съдържащ дължината на реда, старата SHA-1 стойност, новата SHA-1 стойност и името на съответната референция. Може да видите, че първият ред също така подава като информация поддържаните от клиента функционалности (report status). SHA-1 стойността състояща се само от нули означава, че там преди не е имало нищо — понеже добавяте липсващата до момента референция experiment. Ако изтривате референция, ще видите обратното, всички нули ще са в дясната SHA-1 чексума.

След това клиентът изпраща packfile на всички обекти, които сървърът няма. Последно, сървърът отговаря с индикация за успех или грешка:

```
000eunpack ok
```

HTTP(S)

Този процес в голямата си част е същия като HTTP, въпреки че handshaking механизмът е по-различен. Връзката се инициира с тази заявка:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

Това е краят на първата client-server обмяна. Клиентът прави още една заявка, този път по метода `POST` с данните, които `send-pack` доставя.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

Заявката `POST` включва изхода от `send-pack` и самия packfile. Сървът след това отговаря с резултата за успех или отказ.

Имайте предвид, че HTTP протоколът може по-натам да постави тази информация в chunked трансферно кодиране.

Изтегляне на данни

Когато теглите данни, участват процесите `fetch-pack` и `upload-pack`. Клиентът инициира `fetch-pack` процес, който се свързва с `upload-pack` такъв в другия край на връзката за да уговори какви данни ще се изтеглят.

SSH

Ако теглите през SSH, `fetch-pack` изпълнява нещо подобно:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

След като `fetch-pack` се свърже, `upload-pack` изпраща обратно нещо такова:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

Това е много подобно на отговора, който връща `receive-pack`, но функционалностите се различават. Процесът връща в добавка къде сочи HEAD (`symref=HEAD:refs/heads/master`), така че клиентът да знае какво да разпакетира, ако това е клониране.

На този етап процесът `fetch-pack` проверява какви обекти има и отговаря с обектите, които му трябват изпращайки "want" и след това SHA-1 стойностите, които иска. Той изпраща с "have" обектите, които има и техните SHA-1 стойности. В края на този списък той подава "done" за да инструктира `upload-pack` да започне изпращането на packfile пакета данни, който се търси:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

Handshake механизмът за fetch операцията използва две HTTP заявки. Първата е `GET` към същия endpoint използван при dumb протокола:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Това е много подобно на повикване към `git-upload-pack` през SSH връзка, но втората размяна на данни се изпълнява като отделна заявка:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Отново, това е същия формат като по-горе. Отговорът на тази заявка индикира успех или отказ и включва данните в packfile.

Обобщение

Тази секция съдържа много съкратен преглед на транспортните протоколи. Протоколът включва и много други възможности като `multi_ack` или `side-band` поддръжка, но те са извън темата за книгата. Опитахме да опишем основната идея в механизмите на комуникацията между клиент и сървър, ако се нуждаете от повече подробности, може да разгледате сорс кода на Git.

Поддръжка и възстановяване на данни

От време на време може да се налага почистване на хранилището — за да стане то по-компактно, за да се почисти импортирано такова, или за да се възстанови загубена информация. В тази секция ще разгледаме няколко сценария от този сорт.

Поддръжка

В дадени моменти, Git автоматично изпълнява команда наречена “auto gc”. През повечето време, тя не прави нищо. Обаче, ако има твърде много loose обекти (такива, които са извън packfile) или пък твърде много packfiles, Git стартира пълнокръвна `git gc`. Фразата “gc” идва от garbage collect и командата прави много неща: тя събира всички loose обекти и ги поставя в packfiles, консолидира packfiles в един по-голям packfile, и премахва обекти, които са недостъпни през никой къмит и са по-стари от няколко месеца.

Можете да пуснете auto gc ръчно:

```
$ git gc --auto
```

Отново, това в повечето случаи не прави нищо. Трябва да имате приблизително 7,000 loose обекта или повече от 50 packfiles, за да се извърши реално изпълнение на gc командата. Можете да промените тези лимити с `gc.auto` и `gc.autopacklimit` конфигурационните настройки.

Другото нещо, което `gc` ще направи, е да пакетира референциите в единичен файл. Да кажем, че хранилището съдържа следните клонове и тагове:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Ако изпълним `git gc`, за целите на ефективността тези файлове ще бъдат изтрити от `refs` и ще се преместят в единичен `.git/packed-refs`, който изглежда така:

```
$ 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
```

При обновяване на референция, Git не редактира този файл, а създава нов в `refs/heads`. За да вземе SHA-1 чексума за дадена референция, Git първо проверява в директорията `refs` и след това проверява `packed-refs` файла като резервен вариант. Така че, ако не можете да намерите референция в `refs` директорията, тя вероятно е в `packed-refs` файла.

Обърнете внимание на последния ред от файла, който започва с `^`. Това показва, че тагът на реда отгоре е анотиран таг и че този ред е къмита, към който сочи анотирания таг.

Възстановяване на данни

В някой хубав момент при работата си с Git може по невнимание да загубите къмит. Обикновено това се случва, ако направите форсирано изтриване на клон с работа в него или ако направите hard-reset на клон. Ако това се случи, как да си върнете къмитите?

Ето един пример, който прави hard-rest на клона `master` в тестово хранилище до по-стар къмит и след това възстановява загубените къмити. Първо, нека разгледаме хранилището в момента:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Сега преместваме `master` към средния къмит:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Сега ефективно изтрихме горните два къмита — нямаме клон, от който да са достъпни. За да ги възстановим обратно ни трябва SHA-1 чексумата на последния и след това ще добавим клон сочещ към него. Работата е да намерим въпросната чексума — едва ли я запомнихте само от примера, нали?

Често най-бързият начин за това е инструмента `git reflog`. Докато работите, Git задкулисно записва къде сочи HEAD при всяка промяна. Всеки път, когато къмитвате или сменяте клонове, reflog-ът се опреснява. Това става с командата `git update-ref`, което е още една причина да я използваме, вместо да пишем SHA-1 стойностите в ref файловете, както видяхме в Git референции. Можете да видите къде сте били по всяко време с `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: Modify repo.rb a bit
484a592 HEAD@{2}: commit: Create repo.rb
```

Тук виждаме двата къмита, които бяхме разпакетирали, но отделно от това няма много друга информация. За да видим същата информация в много по-полезен вид, може да изпълним `git log -g`, която ни показва това:

```
$ 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

       Modify repo.rb a bit
```

Изглежда сякаш долният къмит е този, който сме загубили, така че можем да го върнем създавайки нов клон базиран на него. Създаваме клон с име `recover-branch` от този къмит (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Чудесно, сега имаме клон `recover-branch` който сочи там, където сочеше `master` преди да го върнем и двата ни къмита са достъпни отново. Сега да допуснем, че загубата е резултат от причина извън reflog-а — можете да симулирате това изтривайки `recover-branch` и самия reflog. Сега двата къмита не са достъпни отникъде:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Понеже reflog данните се пазят в директорията `.git/logs/`, сега практически нямате reflog. Как сега да си върнем къмита? Един начин е да се използва инструмента `git fsck`, който проверява интегритета на базата данни. Ако изпълните командата с аргумента `--full` тя ще ви покаже всички обекти, към които никой друг обект не сочи:

```
$ 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
```

В този случай може да видите липсващия къмит след стринга “dangling commit”. След това можем да възстановим къмита по гореописания начин, добавяйки нов базиран на него клон.

Изтриване на обекти

Една от особеностите на Git, която би могла да доведе до проблеми, е фактът, че при клониране на хранилище `git clone` издърпва цялата история на проекта, включително всяка версия на всеки файл. В това няма нищо лошо, ако става дума само за сорс код, защото Git е силно оптимизиран да компресира ефективно подобен тип данни. Обаче, ако някой в произволен момент от историята на проекта е добавил единичен голям файл, тогава всяко клониране ще трябва да изтегля този файл дори и той да е бил премахнат от проекта още със следващия го къмит. Файлът е достъпен през историята, следователно винаги ще е тук.

Това може да се превърне в огромен проблем, когато конвертирате Subversion или Perforce хранилища в Git. Понеже при тези системи не изтегляте цялата история, такъв тип добавка има последици. Ако сте импортирали от друга система или по друг начин установите, че хранилището ви е много по-голямо отколкото би следвало, ето как да намерите и премахнете големи обекти.

Предупреждаваме: тази техника е деструктивна по отношение на историята на къмитите. Тя пренаписва всеки къмит обект от най-ранното необходимо дърво натам. Ако го направите веднага след импорта, преди никой друг да е базирал работата си на даден къмит, тогава няма проблем — в противен случай следва да уведомите всички колеги, че трябва да пребазират работата си върху вашите нови къмити.

За пример ще добавим един голям файл в тестовото хранилище, ще го премахнем в следващ къмит, след което ще го намерим и ще го изтрием перманентно. Първо го добавяме:

```
$ 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
```

Осъзнаваме, че не искаме този огромен архив в проекта и трябва да се отървем от него:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'Oops - remove large tarball'
[master dadf725] Oops - remove large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Сега правим `gc` на базата данни и преглеждаме използваното дисково пространство:

```
$ 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)
```

Може да изпълните `count-objects` командата за бърз преглед на използваното пространство:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

Редът `size-pack` показва размера на вашите packfiles в килобайти, така че в случая се използват почти 5MB. Преди последния къмит използвахме около 2K — очевидно изтриването на файла от предишния къмит не го е извадило от историята. В това състояние на нещата, всеки път когато някой клонира това хранилище ще трябва да изтегли всичките 5MB, въпреки че проектът е мъничък. И това само защото по невнимание добавихме този голям файл. Нека видим как да го разкараме.

Първо, трябва да го намерим. В този конкретен пример знаем какъв е този файл. Но това може и да не е така и следователно се нуждаем от начин по който да идентифицираме големите файлове в проекта си. Ако изпълните `git gc` всички обекти отиват в packfile и може да използвате друга plumbing команда за да идентифицирате големите. Командата е `git verify-pack` и може да я изпълним сортирайки изхода ѝ по третата колона, която представлява размера на файла. Можете също да пренасочите изхода през `tail`, тъй като търсите само последните няколко големи файла:

```
$ 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
```

Големият обект се показва най-отдолу: 5MB. За да разберете на кой файл отговаря, използвайте командата `rev-list`, която бегло видяхме в Налагане на специфичен Commit-Message формат. Ако ѝ подадете параметъра `--objects`, `rev-list` ще отпечата SHA-1 стойностите на blob обектите с асоциираните към тях пътища на файлове. Може да изпълните нещо такова за да намерите името на файла за този blob:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Сега трябва да изтрием този файл от всички дървета назад във времето. Може лесно да намерим кои къмити са модифицирали този файл:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - remove large tarball
7b30847 Add git tarball
```

Сега трябва да пренапишем всички къмити от `7b30847` назад за да премахнем изцяло файла от историята. За целта ще ползваме командата `filter-branch`, която вече пускахме в Манипулация на историята:

```
$ 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
```

Опцията `--index-filter` е подобна на `--tree-filter`, която използвахме в Манипулация на историята с изключение на това, че вместо да подаваме команда модифицираща файлове от работната директория сега променяме индексната област при всяка итерация.

Вместо да премахваме специфичен файл с нещо като `rm file`, трябва да го извадим от индекса с `git rm --cached`. Трием от индексната област, не от диска. Причината да правим нещата по този начин е скоростта — Git не трябва да разпакетира всяка версия на диска преди да приложи филтъра ни и целият процес е много по-бърз. Разбра се, същата цел може да постигнем и с `--tree-filter`, ако желаем. Флагът `--ignore-unmatch` към `git rm` указва да не се печатат грешки, ако подадения в израза обект не е намерен. Последно, указваме на `filter-branch` да пренапише историята от къмита `7b30847` натам, понеже знаем, че там е възникнал проблема. В противен случай командата започва отначало и ще отнеме

ненужно дълго време.

Сега историята не съдържа референции към този файл. Обаче, reflog информацията и новото множество референции, които Git създаде при изпълнението на `filter-branch` все още пазят данни за него, така че за да е цялостно завършен процеса, трябва да премахнем и тях, след което да препакетираме базата данни. Трябва да се освободим от всичко, което има указатели към тези стари къмити преди препакетирането:

```
$ 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)
```

Време е да видим какво дисково пространство освободихме.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Пакетираното хранилище сега е сведено до 8K, което е доста по-малко от предишните 5MB. От реда size се вижда, че големият обект е все още в loose обектите, така че не е изчезнал съвсем, но по-важното е, че той няма да се изпраща при публикуване или последващо клониране, което е целта ни. Ако окончателно искаме да го изтрием, може да изпълним `git prune` с параметъра `--expire`:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Environment променливи

Git работи в `bash` шел и използва множество environment променливи за да регулира поведението си. Понякога е добре да знаем кои са те и как могат да се използват за фина настройка на Git. Това не е пълен списък с всички използвани променливи, но ще прегледаме най-полезните от тях.

Глобално поведение

Някои от аспектите от работата на Git като компютърна програма зависят от environment променливите.

`GIT_EXEC_PATH` определя къде Git търси своите подпрограми (като `git-commit`, `git-diff`, и т.н.). Може да проверите текущата стойност с `git --exec-path`.

`HOME` нормално не се третира като настройваема променлива (защото твърде много други неща зависят от нейната стойност), от нея се определя къде Git търси глобалния си конфигурационен файл. Ако искате изцяло portable Git инсталация, завършена с глобални конфигурации, може да презапишете `HOME` в portable shell профила на Git.

`PREFIX` е подобна, но се използва за system-wide конфигурации. Git търси този файл в `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, ако е зададена, забранява използването на system-wide конфигурационния файл. Това е полезно, ако системната конфигурация е в конфликт с командите ви, но нямате достъп до нея.

`GIT_PAGER` контролира програмата, която се използва за показване на дълъг изход в командния ред. Ако не е зададена, ще се използва стойността от `PAGER`.

`GIT_EDITOR` задава редактора, който Git ще стартира при нужда от редакция на текст (например за къмит съобщение). Ако не е зададена, използва се посоченото в `EDITOR` променливата.

Локации в хранилища

Git използва няколко environment променливи за да определи как да оперира с текущото хранилище.

`GIT_DIR` е пътя за директорията `.git`. Ако не е указан, Git се качва по дървото с директориите докато стигне до `~` или `/` търсейки за `.git` на всяка стъпка.

`GIT_CEILING_DIRECTORIES` контролира маниера на търсене за `.git` директория. Ако посещавате директории, достъпа до които е бавен (например такива на лентово устройство или през бавна мрежова връзка), може да искате да инструктирате Git да спре да опитва по-рано отколкото би могъл.

`GIT_WORK_TREE` е root локацията на работната директория за non-bare хранилище. Ако са указани `--git-dir` или `GIT_DIR`, но нито едно от `--work-tree`, `GIT_WORK_TREE` и `core.worktree`, тогава за текуща работна директория се счита най-горното ниво на работното ви дърво.

`GIT_INDEX_FILE` е пътя до индексния файл (само за non-bare хранилища).

`GIT_OBJECT_DIRECTORY` може да се използва за задаване на мястото на директорията, която нормално е `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` е разделен с двуеточия списък (форматиран като `/dir/one:/dir/two:…`), който казва на Git къде да проверява за обекти, които не са в локацията от `GIT_OBJECT_DIRECTORY`. Ако се случи да имате много проекти с идентични големи файлове, това може да се използва като начин да избегнете съхранението на твърде много на брой техни копия.

Pathspecs

Терминът “pathspec” се отнася до това как указвате пътища до неща в Git, включително използването на wildcards. Такива се използват в `.gitignore` файла, но също и в командния ред (`git add *.c`).

`GIT_GLOB_PATHSPECS` **и** `GIT_NOGLOB_PATHSPECS` контролират поведението по подразбиране на wildcard символите в pathspecs. Ако `GIT_GLOB_PATHSPECS` е със стойност 1, wildcard символите работят като wildcards (така е по подразбиране). Ако `GIT_NOGLOB_PATHSPECS` е 1, тогава wildcard символите само съвпадат със самите тях, което значи, че `*.c` ще намери само файл *с име* “*.c”, вместо всеки файл с окончание `.c`. Можете да променяте това в индивидуални случаи започвайки дадения pathspec с `:(glob)` или `:(literal)`, както например `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` забранява и двете горни поведения и никакви wildcard символи няма да работят, override префиксите също се забраняват.

`GIT_ICASE_PATHSPECS` указва всички pathspecs да работят в case-insensitive маниер.

Къмитване

Финалното създаване на Git къмит обект обикновено се прави с `git-commit-tree`, която използва тези environment променливи като основен източник на информация, използвайки конфигурационните стойности като резервен вариант, ако те не съществуват:

`GIT_AUTHOR_NAME` е human-readable име за полето “author”.

`GIT_AUTHOR_EMAIL` е имейл адреса за “author”.

`GIT_AUTHOR_DATE` е timestamp за “author” полето.

`GIT_COMMITTER_NAME` задава human име за полето “committer”.

`GIT_COMMITTER_EMAIL` е имейл адреса за “committer”.

`GIT_COMMITTER_DATE` е timestamp за “committer” полето.

`EMAIL` се използва като резервен имейл адрес, ако нямаме стойност за конфигурацията `user.email`. Ако и *тази променлива* не е зададена, Git използва системните user и host имена.

Мрежови функции

Git използва библиотеката `curl` за мрежови активности през HTTP. Променливата **`GIT_CURL_VERBOSE`** указва на Git да печата всички съобщения, генерирани от библиотеката. Това е подобно на `curl -v` в командния ред.

`GIT_SSL_NO_VERIFY` казва на Git да не проверява SSL сертификати. Това понякога е необходимо, ако използвате self-signed сертификати за да обслужвате Git хранилища през HTTPS или ако изграждате Git сървър, но все още не сте инсталирали валиден сертификат.

Ако скоростта на HTTP операция е по-ниска от указаните в **`GIT_HTTP_LOW_SPEED_LIMIT`** байта в секунда за повече от **`GIT_HTTP_LOW_SPEED_TIME`** секунди, Git ще откаже тази операция. Тези стойности са с приоритет пред `http.lowSpeedLimit` и `http.lowSpeedTime` конфигурационните настройки.

`GIT_HTTP_USER_AGENT` определя user-agent стринга, който Git използва при комуникация през HTTP. По подразбиране се използва нещо като `git/2.0.0`.

Diffing и Merging

`GIT_DIFF_OPTS` е леко заблуждаващо наименование. Единствено валидните стойности са `-u<n>` или `--unified=<n>`, което определя броя на context редовете показвани от команда `git diff`.

`GIT_EXTERNAL_DIFF` се използва за приоритет пред конфигурационната стойност `diff.external`. Ако е зададена, Git ще стартира съответната програма при изпълнение на `git diff`.

`GIT_DIFF_PATH_COUNTER` и **`GIT_DIFF_PATH_TOTAL`** са полезни за използване в програмата указана от `GIT_EXTERNAL_DIFF` или `diff.external`. Първата указва върху кой файл от серия такива ще се изпълни diff (започвайки с 1), а втората задава общия брой файлове за всяка batch операция.

`GIT_MERGE_VERBOSITY` контролира изхода за recursive merge стратегия. Позволените стойности са:

- 0 не извежда нищо с изключение на евентуално единично съобщение за грешка.
- 1 показва само конфликти.
- 2 също показва и файлови промени.
- 3 показва, когато файлове се пропускат, защото не са променени.
- 4 показва всички пътища по време на обработка.
- 5 и нагоре, показва детайлна debugging информация.

По подразбиране стойността е 2.

Дебъгване

Искате *наистина* да знаете с какво се занимава Git? Git има сравнително завършен набор от вградени traces и просто трябва да ги включите. Възможните стойност за тези променливи са както следва:

- “true”, “1”, или “2” — trace категорията се изпраща към stderr.
- Абсолютен път започващ с `/` — trace изходът ще се записва в този файл.

`GIT_TRACE` управлява общи traces, които не попадат в нито една специфична категория. Това включва разширяването на aliases, и делегиране към други подпрограми.

```
$ 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' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
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` контролира трасирането на packfile достъп. Първото поле е съответния packfile, второто е отстъпа в него:

```
$ 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 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35175
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

`GIT_TRACE_PACKET` позволява packet-level tracing за мрежови операции.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-upload-
pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_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/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46           packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# […]
```

`GIT_TRACE_PERFORMANCE` контролира записа на данните за производителността. Изходът показва колко време е отнела всяка отделна `git` команда.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
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-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git command: 'git'
'gc'
```

`GIT_TRACE_SETUP` показва информация за това какво знае Git за хранилището и обкръжението, с които комуникира.

```
$ 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
```

Други променливи

`GIT_SSH`, ако е указана, е програмата, която ще се използва вместо `ssh`, когато Git опитва да комуникира с SSH хост. Тя се извиква така: `$GIT_SSH [username@]host [-p <port>] <command>`. Това не е най-лесният начин да се промени настройката за повикване на `ssh` и няма да поддържа допълнителни параметри за командния ред, така че може би ще е по-добре да си напишете малък wrapper script и да зададете него като стойност на `GIT_SSH`. Вероятно е по-лесно просто да се използва `~/.ssh/config` файла.

`GIT_ASKPASS` се използва за приоритет пред конфигурационната настройка `core.askpass`. Това е програмата, която се пуска всеки път когато Git трябва да пита потребителя за логин информация, тя трябва да върне отговора в `stdout`. (Вижте Credential Storage система за повече информация).

`GIT_NAMESPACE` контролира достъпа до namespaced refs и е еквивалентна на флага `--namespace`. Това е най-полезно от страна на сървъра, където може да искате да съхранявате множество forks на единични хранилище в едно хранилище, пазейки отделни само референциите.

`GIT_FLUSH` може да се използва за да накарате Git да използва non-buffered I/O, когато пише инкрементално към stdout. Стойност 1 ще накара Git да flush-ва по-често, а 0 означава, че целият изход се буферира. Правилото по подразбиране (ако променливата не е настроена) е да се избере подходящата буферираща схема според активността и output режима.

`GIT_REFLOG_ACTION` позволява да укажете описателния текст, който се записва в reflog. Например:

```
$ 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
```

Обобщение

На този етап вече знаете доста подробности за това какво прави Git на заден план и също така, до известна степен — как е имплементирано това. Разгледахме множество plumbing команди — команди от по-ниско ниво и по-прости от porcelain командите, които ползвахме до момента. Разбирането на това как работи Git вътрешно ще ви помогне да разберете по-добре защо системата прави нещата точно по даден начин, а също и ще ви помогне да

напишете по-лесно собствени инструменти и помощни скриптове съобразени със спецификата на вашия работен процес.

Като content-addressable файлова система Git е много мощен инструмент, който можете да използвате за повече задачи освен за контрол на версиите. Надяваме се, вече можете да използвате новите си знания за Git за да имплементирате собствени разработки базирани на технологията и улесняващи работата на вас и вашия екип.

Appendix A: Git в други среди

Ако сте прочели цялата книга сте научили много за използването на Git от командния ред. Можете да работите с локалните си файлове, да свързвате хранилището си с отдалечени такива през мрежа, да работите ефективно с колегите си. Но историята не свършва тук. Git често се използва като част от по-големи екосистеми и терминалът не винаги е най-добрия начин за работа. Сега ще разгледаме няколко работни среди, в които Git може да е полезна и също как други приложения (вкл. вашите) работят в синхрон с Git.

Графични интерфейси

Нативната среда на Git е терминала. Новите функционалности се появяват първо там и само командния ред е мястото, където Git ви предоставя пълната си сила. Но чистият текст не винаги е подходящия избор за всички задачи, понякога визуалното представяне е необходимо и отделно от това има много потребители, които се чувстват по-комфортно с мишката.

Важно е да посочим, че различните интерфейси са проектирани за различни работни процеси. Някои клиенти дават достъп само до внимателно подбрано подмножество от функционалностите на Git с цел да подпомогнат специфичен начин на работа, който авторът на интерфейса намира за ефективен. Погледнато от този аспект, никой от тези инструменти не може да се нарече "по-добър" от кой да е друг, те просто са фокусирани за конкретна цел. Също така подчертаваме, че няма дейност извършвана от графичните клиенти, която да не може да се реализира в команден ред.

`gitk` и `git-gui`

Когато инсталирате Git, получавате неговите визуални инструменти `gitk` и `git-gui`.

`gitk` е графичен инструмент за разглеждане на историята. Мислете за него като за мощен GUI заместител на `git log` и `git grep`. Това е инструментът, който ви трябва, ако се опитвате да намерите нещо случило се в миналото или искате да визуализирате историята на проекта.

Gitk най-лесно се стартира от командния ред. Просто влезте в Git хранилище и изпълнете:

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

Gitk приема много аргументи от командния ред, повечето от които се изпращат към съответните `git log` операции. Един от най-полезните е флагът `--all`, който инструктира gitk да показва къмитите достъпни от *всяка* референция, не само от HEAD. Интерфейсът изглежда така:

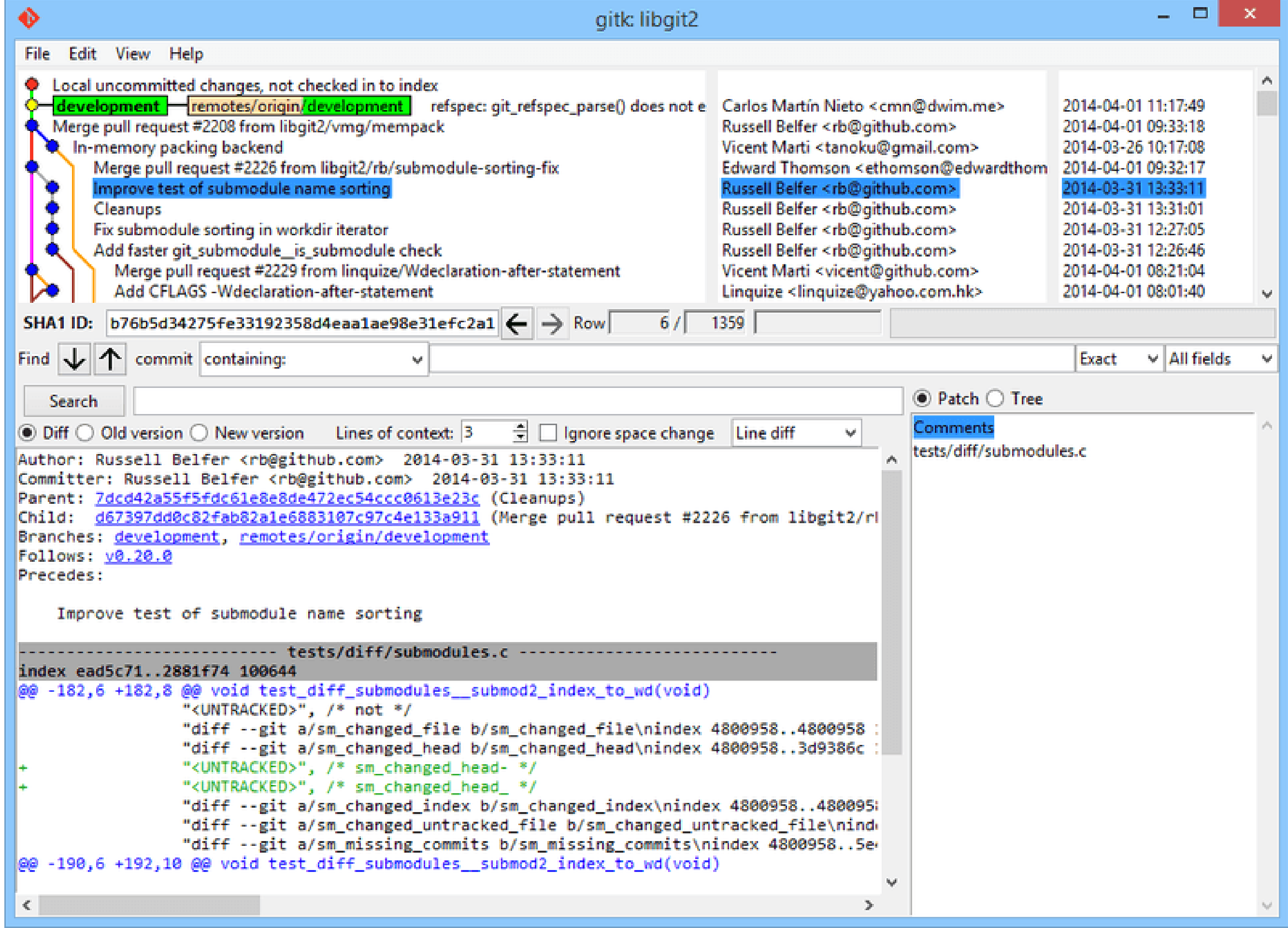


Figure 151. `gitk` history viewer

Най-отгоре виждаме нещо наподобяващо изхода на `git log --graph`, всяка точка представлява къмит, линиите показват родителските връзки и референциите се показват като оцветени кутии. Жълтата точка представя HEAD, а червената точка са промените, които все още не са къмитнати. В долния край имате преглед на избрания къмит, коментарите и пача отляво и обобщение вдясно. Между тези две секции има колекция от контроли за търсене в историята.

`git-gui`, от друга страна, е предимно инструмент за работа с къмити. И той се стартира най-лесно от конзолата:

```
$ git gui
```

И изглежда така:

`git-gui` commit tool image::images/git-gui.png[`git-gui` commit tool]

Отляво на екрана виждаме неиндексираните промени отгоре, под тях са индексираните. Можете да местите цели файлове между двете области щраквайки върху иконите им или можете да изберете файл за преглед като щракнете върху името му.

Горе вдясно е diff изгледът, който показва промените по текущо селектирания файл. Можете да индексирате индивидуални hunks (или отделни редове) с дясно щракване в тази област.

Долу вдясно е областта за дейности и съобщения. Напишете съобщението си в кутията и натиснете “Commit” за да направите нещо подобно на `git commit`. Можете също да изберете да откажете последния къмит с радио бутона `‘Amend’', което ще опресни областта “Staged Changes” със съдържанието му. След това можете просто да индексирате и деиндексирате някои промени, да промените къмит съобщението и да натиснете бутона “Commit” повторно, за да замените стария къмит с нов.

`gitk` и `git-gui` са примери за task-oriented инструменти. Всеки от тях е насочен към специфична задача (съответно за преглед на история и промяна на къмити) и не предлагат функционалности, които не я засягат.

GitHub за macOS и Windows

GitHub предоставя два workflow-ориентирани Git клиента, по един за Windows и macOS. Тези клиенти са добър пример за workflow-ориентирани инструменти — вместо да предоставят *цялата* Git функционалност, те са фокусирани върху подбрани части от нея, които работят добре заедно и които потребителите често използват. Те изглеждат така:

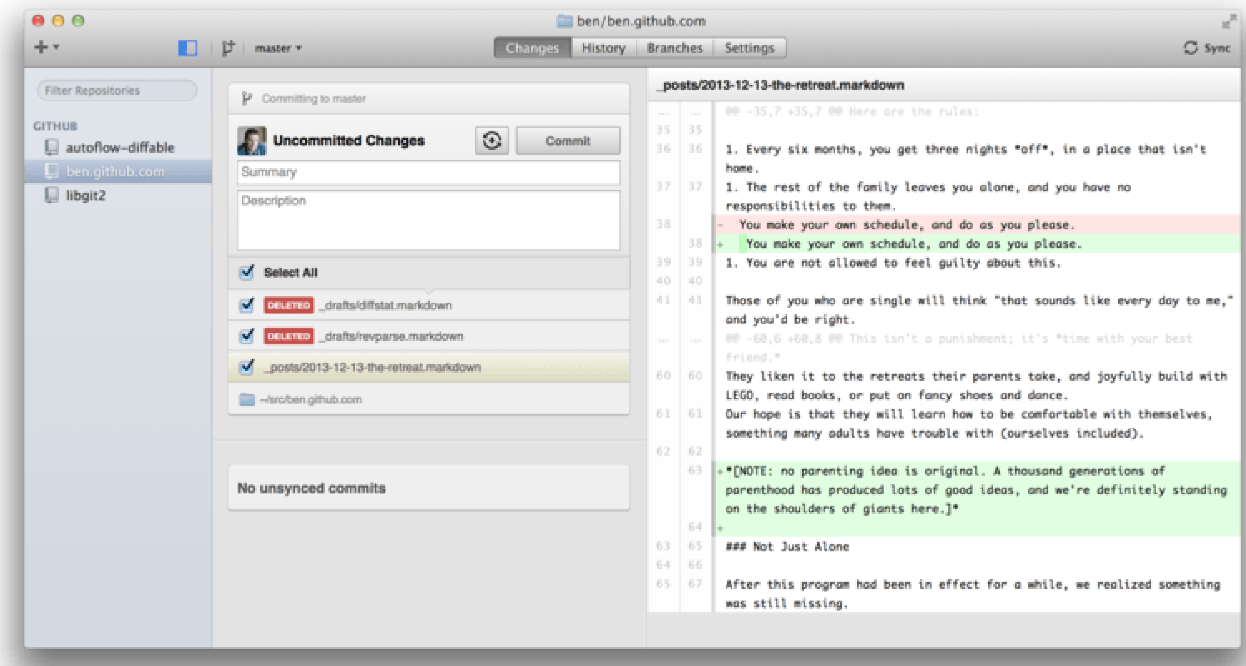


Figure 152. GitHub за macOS

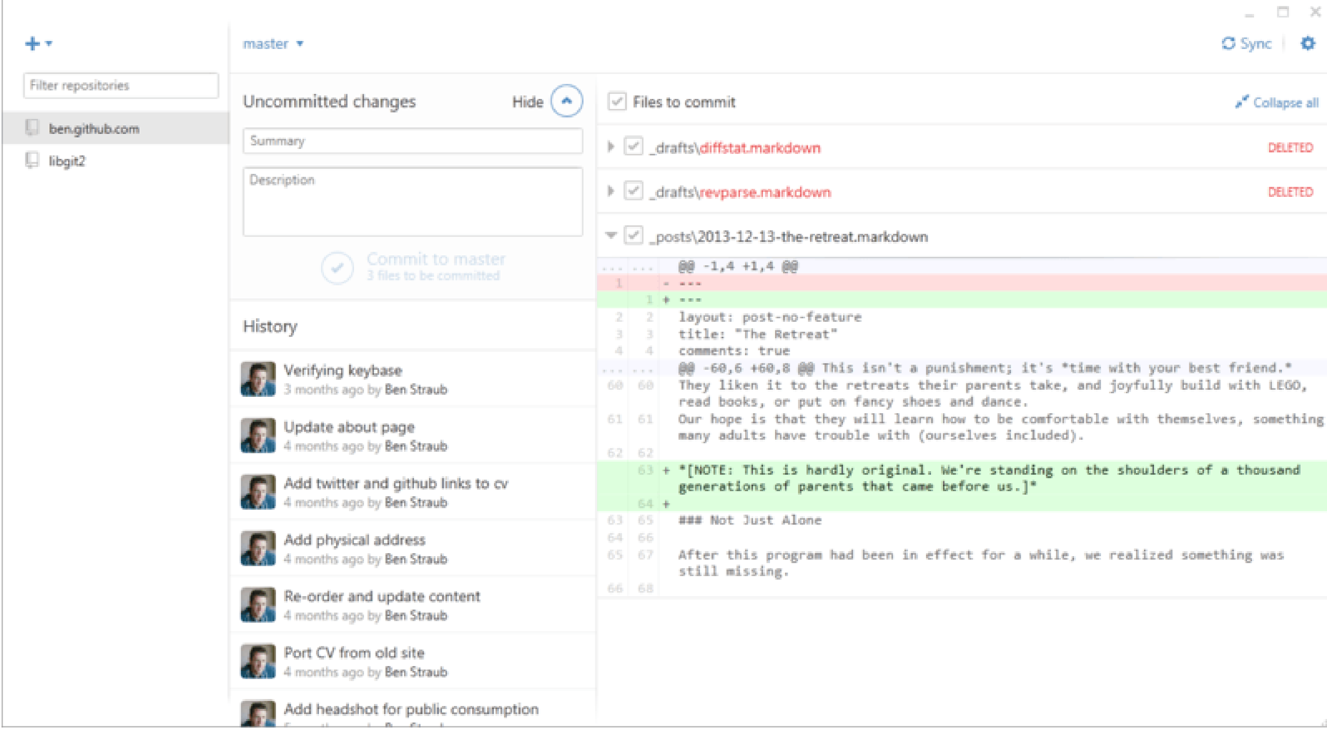


Figure 153. GitHub за Windows.

Проектирани са да изглеждат и работят почти еднакво, така че ще ги разгледаме като един продукт. Няма да навлизаме в детайли (имат подробна документация), а ще разгледаме набързо изгледа “changes” (в който ще прекарвате повечето си време).

- Вляво са хранилищата, които клиентът следи; можете да добавяте хранилища (с клониране или локално прикачване) с иконата “+” в горния край.
- В центъра е commit-input областта, която позволява да въведете къмит съобщение и да изберете кои файлове да се включат. Под Windows, историята на къмитите се показва директно отдолу, под macOS е в отделен таб.
- В дясно е diff изгледа, който показва промените в работната директория или кои промени са включени в избрания къмит.
- Последно имаме бутон “Sync” в горната дясна част, който е основния начин за комуникация през мрежата.

Не се нуждаете от GitHub акаунт за да ползвате тези програми. Въпреки, че са проектирани да се възползват максимално добре от услугите на GitHub и препоръчвания от него тип работен процес, те ще си работят добре с всяко хранилище и ще контактуват по мрежата с всеки Git хост.

Инсталация

GitHub for Windows може да се изтегли от https://windows.github.com, а GitHub for macOS от https://mac.github.com. При първото им стартиране приложенията ви водят през първоначална настройка на Git, вкл. конфигуриране на вашето име и имейл адрес, след което задават много настройки по подразбиране като credential кеширането и CRLF обработката.

И двете са винаги актуални - обновяванията им се издърпват и инсталират на заден план

докато приложението работи. Тъй като включват и вградена версия на Git, тя също автоматично се обновява и няма нужда да го правите на ръка. Под Windows, клиентът включва shortcut за стартиране на PowerShell с Posh-git, за който ще говорим малко по-късно.

Следващата стъпка е да дадете на приложенията хранилища, с които да работят. Те ви показват списък с хранилищата ви в GitHub и ви позволяват да ги клонирате в една стъпка. Ако имате локално хранилище, просто издърпайте с мишката директорията му от Finder или Windows Explorer в клиентския прозорец на програмата и то ще бъде добавено към списъка с хранилища вляво.

Препоръчителен работен процес

Веднъж инсталиран и конфигуриран, GitHub клиентът може да се използва за много ежедневни Git задачи. Заложеният работен процес понякога се нарича "GitHub Flow." Вече разгледахме това в Работния процес в GitHub, като цяло идеята е (a) да къмитвате в клон, и (b) да синхронизирате с отдалечено хранилище сравнително редовно.

Управлението на клоновете е една от областите, в които двете версии се различават. В Mac, имате бутон за създаване на нов клон в горната част:

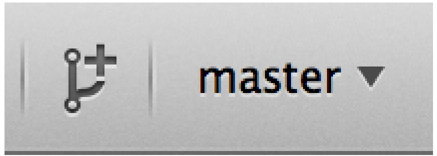


Figure 154. "Create Branch" бутон под macOS

Под Windows, това се прави като напишете името на новия клон в branch-switching уиджета:

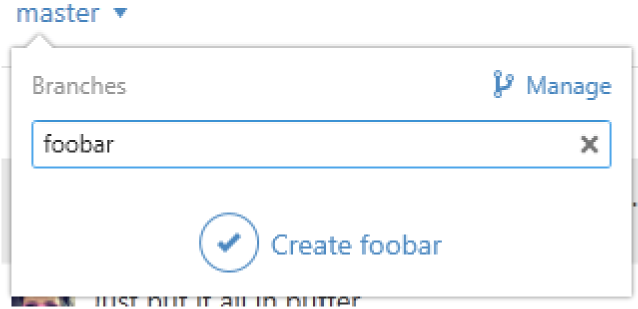


Figure 155. Създаване на клон под Windows

След като клонът е създаден, правенето на нови къмити е сравнително лесно. Направете някакви промени в работната директория и когато се върнете в прозореца на GitHub клиента, той ще ви покаже кои файлове са били променени. Въвеждате къмит съобщение, избирате файловете, които искате да бъдат включени и натискате бутона "Commit" (ctrl-enter или -enter).

Основният начин, по който контакувате с други хранилища през мрежата е чрез функцията "Sync". Git вътрешно си има отделни операции за публикуване, издърпване, сливане и пребазиране, но GitHub клиентите ги обединяват в единична функция с няколко стъпки. Ето какво се случва, когато натиснете бутона Sync:

1. `git pull --rebase`. Ако това не успее поради merge конфликт, тогава алтернативата е `git pull --no-rebase`.
2. `git push`.

Това е най-често срещаната последователност от действия при работа в такъв стил, така че обединяването на командите в една спестява много време.

Обобщение

Тези инструменти са много добре окомплектовани за работния процес, за който са проектирани. Те позволяват бързо изграждане на механизъм за съвместна работа между разработчици и странични лица при съблюдаване на добрите практики. Обаче, ако вашият работен процес е различен или пък искате повече контрол върху мрежовите операции, бихме препоръчали друг клиент или директно командния ред.

Други графични инструменти

Съществуват множество други графични Git клиенти, при това с широк диапазон от функционалности — от тясно специализирани върху конкретни дейности до такива, които се опитват да въплатят всичко, което Git поддържа. Официалният сайт на Git съдържа списък с най-популярните клиенти на адрес https://git-scm.com/downloads/guis. Още по-пълен списък е наличен от Git wiki сайта на адрес https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git във Visual Studio

След Visual Studio 2013 Update 1, Visual Studio потребителите имат Git клиент вграден директно в IDE средата. Visual Studio разполага със source-control функционалности от доста време, но в миналото те бяха ориентирани към централизирани, file-locking системи и Git не се вписваше добре в този стил на работа. Поддръжката на Git във Visual Studio 2013 е отделена от тези по-стари функции и резултатът е много по-добра интеграция между Studio и Git.

За да видите Git в средата, отворете проект под контрола на Git (или изпълнете `git init` в съществуващ такъв), след което изберете View > Team Explorer от менюто. Ще видите изгледа "Connect":

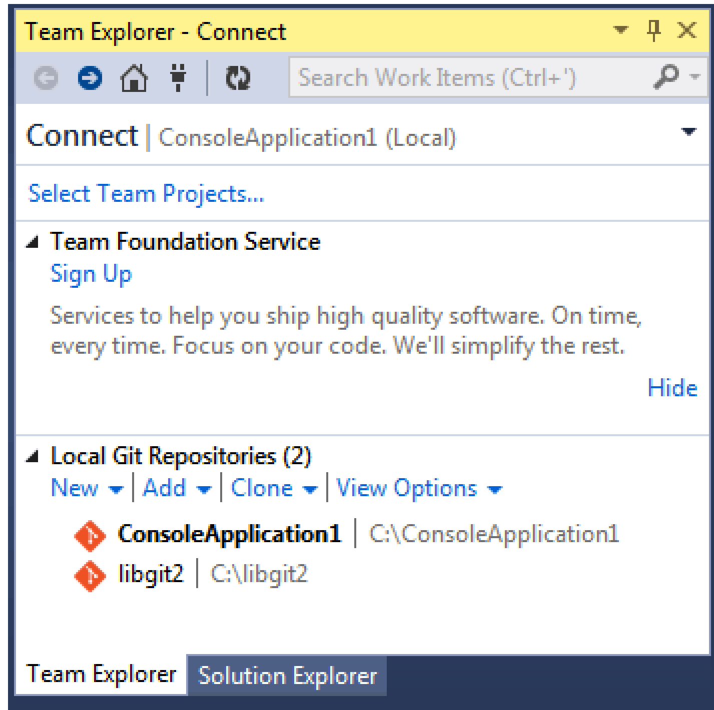


Figure 156. Свързване към Git хранилище от Team Explorer.

Visual Studio запомня всички Git проекти отваряни по-рано и ги показва в списъка в долната част. Ако не виждате желания от вас, натиснете линка "Add" и напишете пътя до работната директория. Когато щракнете два пъти върху някое от локалните Git хранилища, ще бъдете прехвърлени към изгледа Home, изглеждащ като на екрана Изгледът Home за Git хранилище във Visual Studio. Това е нещо като хъб за изпълнение на Git операции, когато *пишете* код вероятно ще прекарвате повечето си време в изгледа "Changes", но когато дойде време да издърпате промените направени от колегите ви, ще използвате изгледите "Unsynced Commits" и "Branches".

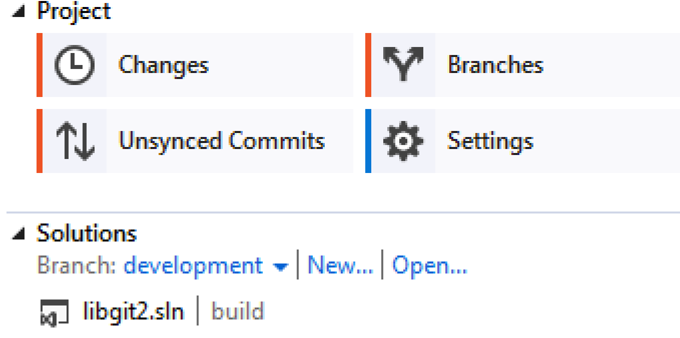


Figure 157. Изгледът Home за Git хранилище във Visual Studio.

Visual Studio сега има мощен, фокусиран върху задачите, графичен интерфейс за Git. Той включва преглед на линейна история, diff viewer, поддръжка за отдалечени команди и много други възможности. Повече за използването на Git с Visual Studio ще намерите на адрес: https://docs.microsoft.com/en-us/azure/devops/repos/git/command-prompt?view=azure-devops.

Git във Visual Studio Code

Visual Studio Code има вградена git поддръжка. Нуждаете се само от инсталиран git версия

2.0.0 или по-нова.

Основни функционалности:

- Виждате diff на текущо редактирания файл в gutter-а.
- Git Status Bar-ът (долу вляво) показва текущия клон, dirty индикатори, входни и изходни къмити.
- Можете да изпълнявате повечето стандартни git операции от редактора:
    - Инициализиране на хранилище.
    - Клониране на хранилище.
    - Създаване на клонове и тагове.
    - Индексиране и къмитване на промени.
    - Push/pull/sync с отдалечени клонове.
    - Разрешаване на merge конфликти.
    - Преглед на diffs.
- Чрез допълнително разширение, можете също така да обслужвате GitHub Pull Request-и: https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github.

Официалната документация е на адрес: https://code.visualstudio.com/Docs/editor/versioncontrol.

Git in IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine

JetBrains IDEs (such as IntelliJ IDEA, PyCharm, WebStorm, PhpStorm, RubyMine, and others) ship with a Git Integration plugin. It provides a dedicated view in the IDE to work with Git and GitHub Pull Requests.

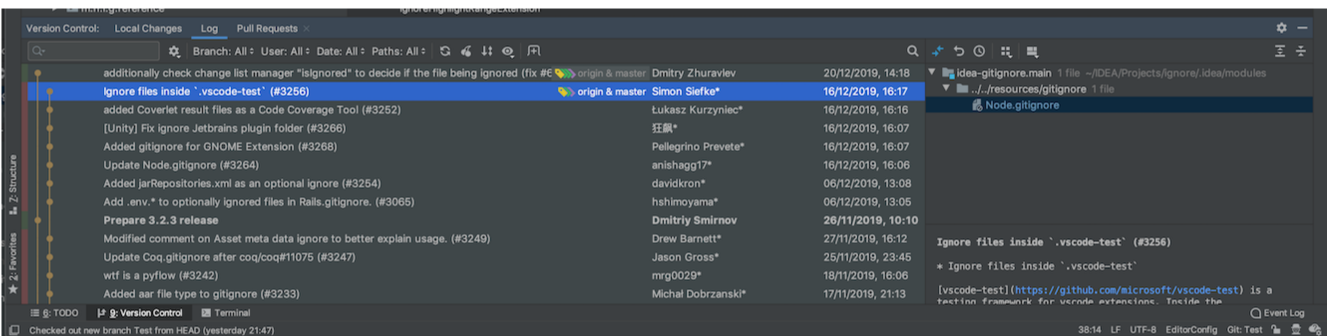


Figure 158. Version Control ToolWindow in JetBrains IDEs

The integration relies on the command-line git client, and requires one to be installed. The official documentation is available at https://www.jetbrains.com/help/idea/using-git-integration.html.

Git в Sublime Text

От версия 3.2, Sublime Text предлага git интеграция в редактора.

Възможностите са:

- Страничният панел ще показва git статуса на файловете и директориите с бадж/икона.
- Файловете и директориите, които са в .gitignore файла ще бъдат показвани избледнели в страничния панел.
- В статус лентата можете да виждате текущия git клон и колко промени сте направили.
- Всички промени по файл сега са видими чрез маркери в gutter-а.
- Може да използвате част от функционалността на Sublime Merge git клиента в Sublime Text. Това изисква инсталиран Sublime Merge. Вижте: https://www.sublimemerge.com/.

Официалната документация на Sublime Text е на адрес: https://www.sublimetext.com/docs/3/git_integration.html.

Git в Bash

Ако сте Bash потребител, може да промените леко настройките на шела, така че работата ви с Git да стане много по-приятна. Git в действителност има плъгини за няколко шела, но това не е активно по подразбиране.

Първо, трябва да се сдобиете с копие от файла `contrib/completion/git-completion.bash` от сорс кода на Git. Копирайте го на удобно място, например в домашната си директория и добавете това към вашия `.bashrc`:

```
. ~/git-completion.bash
```

След това, влезте в Git хранилище и напишете:

```
$ git chec<tab>
```

… и Bash ще допълни командата автоматично до `git checkout`. Това работи с всички подкоманди на Git, с параметрите за командния ред, а също и с имената на референциите, където е подходящо.

Полезно е също така да настроите вашия промпт да показва информация за Git хранилището в текущата директория. Тази информация може да е кратка или по-подробна, но в общи линии има няколко основни данни, които повечето хора искат да имат, като текущия клон или статуса на работната директория. За да направите това, копирайте файла `contrib/completion/git-prompt.sh` от сорс кода на Git в домашната си директория и добавете нещо такова във файла `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

Съчетанието `\w` означава да се печата текущата работна директория, `\$` извежда частта `$` от промпта, а `__git_ps1 " (%s)"` вика функцията осигурена от `git-prompt.sh` с форматиращ аргумент. Сега вашият bash промпт ще изглежда така, когато сте в Git проект:

Figure 159. Специално настроен `bash` промпт

И двата скрипта имат полезна документация, просто погледнете в съдържанието на файловете `git-completion.bash` и `git-prompt.sh`.

Git в Zsh

Zsh също има tab-completion библиотека за Git. За да я ползвате, просто изпълнете `autoload -Uz compinit && compinit` във вашия `.zshrc` файл. Интерфейсът на Zsh е малко по-гъвкав от този на Bash:

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

Както виждате, възможните варианти не само се показват, но се и описват и можете графично да минавате през тях натискайки клавиша табулация. Това работи за Git команди, за аргументите им и за имена в рамките на хранилище (refs и remotes), а също така и за имена на файлове и за всички други неща, които Zsh знае как автоматично да допълва.

Zsh има също така framework за извличане на информация от version control системи, наречена `vcs_info`. За да включите името на клона отдясно в промпта, добавете тези редове в `~/.zshrc`:

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_'%# '
zstyle ':vcs_info:git:*' formats '%b'
```

В резултат на това ще имате името на клона в дясната част на прозореца, когато сте в Git хранилище. Разбира се, поддържа се и лявата страна, просто разкоментирайте присвояването към PROMPT. Изглежда приблизително така:

Figure 160. Потребителски `zsh` *промпт*

За повече информация за vcs_info, погледнете документацията в `zshcontrib(1)` man страницата или онлайн на адрес http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information.

Вместо `vcs_info`, може да изберете prompt customization скрипта, който е част от Git и се нарича `git-prompt.sh`; вижте https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh за подробности. `git-prompt.sh` е съвместим с Bash и Zsh.

Zsh е достатъчно мощен и има цели frameworks предназначени да го направят още по-добър. Една от тях е "oh-my-zsh", налична от https://github.com/robbyrussell/oh-my-zsh. Плъгин системата на oh-my-zsh идва с git tab-completion и има разнообразни промпт "теми", много от които показват version-control информация. Изображението от фигурата Пример за oh-my-zsh тема е само един вариант на това какво може да се направи с тази система.

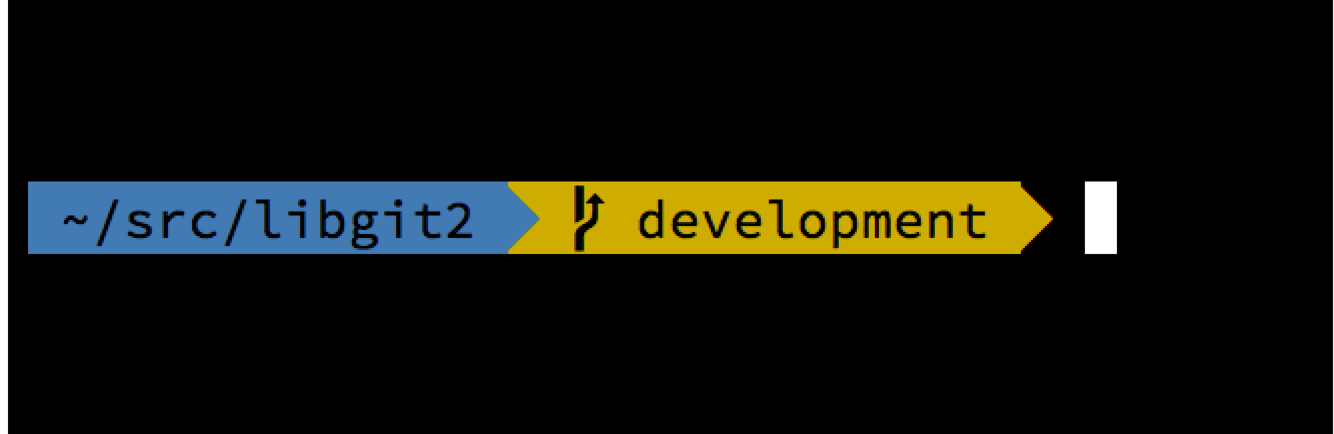


Figure 161. Пример за oh-my-zsh тема

Git в PowerShell

Стандартният команден терминал в Windows (`cmd.exe`) не е особено подходящ за комфортна работа с Git, но ако използвате PowerShell, тогава нещата са други. Това също работи ако използвате PowerShell Core на друга платформа, например Linux или macOS. Пакетът posh-git (https://github.com/dahlbyk/posh-git) осигурява мощна tab-completion функционалност, както и подобрен промпт за вашите хранилища. Изглежда така:

Figure 162. PowerShell с Posh-git

Инсталация

Изисквания (само за Windows)

Преди да можете да изпълнявате PowerShell скриптове, трябва да настроите вашата локална ExecutionPolicy на RemoteSigned (всичко с изключение на Undefined и Restricted). Ако изберете AllSigned вместо RemoteSigned, също и локалните скриптове (вашите собствени) трябва да бъдат цифрово подписани, за да могат да бъдат изпълнявани. С RemoteSigned, само скриптовете със "ZoneIdentifier" настроен на Internet (тоест, изтеглени от мрежата) е необходимо да са подписани, другите не. Ако сте администратор и искате да го зададете за всички потребители, използвайте "-Scope LocalMachine". Ако сте нормален потребител без административни права, използвайте "-Scope CurrentUser" за да зададете настройката само за вас.

Повече за PowerShell Scopes: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes.

Повече за PowerShell ExecutionPolicy: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy.

За да зададете стойност `RemoteSigned` на `ExecutionPolicy` за всички потребители, използвайте командата:

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

PowerShell Gallery

Ако имате поне PowerShell 5 или PowerShell 4 с PackageManagement, можете да използвате пакетния мениджър за да издърпате posh-git.

Повече информация за PowerShell Gallery: https://docs.microsoft.com/en-us/powershell/scripting/gallery/overview.

```
> Install-Module posh-git -Scope CurrentUser -Force
> Install-Module posh-git -Scope CurrentUser -AllowPrerelease -Force # Newer beta
version with PowerShell Core support
-----

Ако искате да инсталирате posh-git за всички потребители, подайте "-Scope
AllUsers" вместо това и изпълнете командата от elevated PowerShell конзола.
Ако втората команда завърши с грешка от рода на `Module 'PowerShellGet' was not
installed by using Install-Module`, ще трябва първо да изпълните друга такава:

[source,powershell]
```

> Install-Module PowerShellGet -Force -SkipPublisherCheck

```
След това може да се върнете и да опитате отново.
Това се случва, защото модулите, които идват с Windows Powershell са подписани
с различни сертификати.

===== Обновяване на PowerShell промпта

За да включите Git информация в промпта, posh-git модулът трябва да бъде
импортиран.
За да се импортира posh-git всеки път, когато се пусне PowerShell, изпълнете
командата Add-PoshGitToProfile, която ще добави импортиращия израз във вашия
$profile скрипт.
Той се изпълнява всеки път, когато отворите нов PowerShell промпт.
Не забравяйте, че има няколко $profile скриптове.
Например един за конзолата и друг отделен за ISE.

[source,powershell]
```

> Import-Module posh-git Add-PoshGitToProfile -AllHosts

```
===== От изходен код

Просто изтеглете posh-git версия от (https://github.com/dahlbyk/posh-git[]), и я
разархивирайте.
След това импортирайте модула използвайки пълния път до файла posh-git.psd1:

[source,powershell]
```

> Import-Module <path-to-uncompress-folder>\src\posh-git.psd1 Add-PoshGitToProfile -AllHosts

```
Това ще добави правилния ред във файла `profile.ps1` и posh-git ще е активен
следващия път, когато отворите вашия промпт.

За описание на Git status summary информацията, която се вижда в промпта,
погледнете: https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-
summary-information[]

=== Обобщение

Научихте как да използвате силата на Git от инструментите, които ползвате в
ежедневната си работа и как да получите достъп до Git хранилище от
собствените си програми.

[[B-embedding-git-in-your-applications]]
[appendix]
== Вграждане на Git в приложения
```

Ако вашето приложение е предназначено за разработчици, твърде вероятно е интеграцията на сорс контрол възможности да му е от полза.
Дори и non-developer приложенията, като редактори на документи например, могат потенциално да се възползват от version-control функционалностите и моделът на Git работи добре в много и разнообразни сценарии.

Ако трябва да интегрирате Git във вашите собствени приложения, по същество разполагате с две опции: да извиквате шел сесия и да се обръщате към `git` от командния ред или да вградите Git библиотека в приложението си.
Тук ще погледнем интеграцията от команден ред и някои от най-популярните Git библиотеки, които могат да се вграждат.

=== Git от команден ред

Една възможност е да се стартира шел процес и да се използват Git командите за необходимите дейности.
Предимството на такъв подход е в последователността и всички Git функционалности са налице.
Освен това е сравнително лесно, защото повечето runtime среди разполагат със сравнително лесен механизъм за извикване на процеси с аргументи от команден ред.
Обаче, подходът си има и недостатъци.

Един от тях е, че изходът е в чист текст.
Това означава, че трябва да се обработва изходния формат на Git за прочитане на прогреса на дадено действие и информацията от резултатите, което може да е неефективен и податлив на грешки процес.

Друго неудобство е липсата на поддръжка за възстановяване на състоянието при възникнали грешки.
Ако дадено хранилище по някакъв начин се повреди или потребителят е подал неправилно форматирана конфигурационна стойност, то Git просто ще откаже да изпълни много операции.

Друг аспект е управлението на процесите.
Git изисква да поддържате шел обкръжение в отделен процес, което може да добави нежелана сложност.
Опитът да координирате много такива процеси (особено когато потенциално манипулирате едно и също хранилище от много процеси) може да бъде сериозно предизвикателство.

=== Libgit2

(((libgit2)))((("C")))
Библиотеката Libgit2 е друга опция на ваше разположение.
Libgit2 е dependency-free имплементация на Git, фокусирана в предоставянето на добър API за ползване от външни програми.
Налична е от https://libgit2.org[].

Първо нека видим как изглежда един C API.
Накратко:

```
[source,c]
```

git_repository *repo; int error = git_repository_open(&repo, "/path/to/repository");

git_object **head_commit; error** = **git_revparse_single(&head_commit, repo, "HEAD^{commit}"); git_commit *commit** = **(git_commit**)head_commit;

printf("%s", git_commit_message(commit)); const git_signature *author = git_commit_author(commit); printf("%s <%s>\n", author→name, author→email); const git_oid *tree_id = git_commit_tree_id(commit);

git_commit_free(commit); git_repository_free(repo);

```
Първите няколко реда отварят Git хранилище.
Типът `git_repository` представлява указател към хранилище с кеш в паметта.
Това е най-простият метод за случаите, когато знаете точния път към работната
директория на хранилище или директорията `.git`.
Съществува и `git_repository_open_ext`, където имаме опции за търсене,
`git_clone` и подобни команди за правене на локално копие на отдалечено
хранилище, както и `git_repository_init` за създаване на изцяло ново хранилище.

Следващият елемент от кода използва rev-parse синтаксис (вижте <<ch07-git-
tools#_branch_references>> за подробности) за да вземе къмита, към който сочи
HEAD.
Върнатият тип е `git_object` указател, който дава достъп до съдържанието на
обектната база данни в Git хранилище.
`git_object` в действителност е ``parent'' тип за няколко различни вида обекти,
разположението в паметта за всеки от ``child'' типовете е същото като на
`git_object`, така че може безопасно да се cast-ва до правилния такъв.
В този случай, `git_object_type(commit)` ще върне `GIT_OBJ_COMMIT`, така че е
възможно да се cast-не към `git_commit` указател.

Следващата част от кода показва как да се получи достъп до свойствата на
къмита.
Последният ред използва типа `git_oid`, което е Libgit2 представянето на SHA-1
хеш.

От този пример можем да направим следните изводи:

* Ако декларирате указател и изпратите референция към него в Libgit2
повикване, това повикване вероятно ще върне целочислен код за грешка.
  Стойност `0` индикира успех, всичко по-малко е грешка.
* Ако Libgit2 инициализира указател за вас, ваша е отговорността да го
освободите.
* Ако Libgit2 върне `const` указател от повикване, не трябва да го
освобождавате, но той ще стане невалиден, когато обектът, към който
принадлежи бъде освободен.
* Писането на код на C може да бъде доста болезнено.

(((Ruby)))
Последното означава, че не е много вероятно да пишете на C, когато използвате
Libgit2.
За щастие, налични са много language-specific bindings, които правят сравнително
лесно да работите с Git хранилища от вашия специфичен език за програмиране и
среда.
Нека видим примера отгоре написан с помощта на Ruby bindings за Libgit2,
наречени Rugged и налични от https://github.com/libgit2/rugged[].

[source,ruby]
```

repo = Rugged::Repository.new(*path/to/repository*) commit = repo.head.target puts commit.message puts "{commit.author[:name]} <{commit.author[:email]}>" tree = commit.tree

```
Както се вижда, кодът е доста по-прегледен.
Първо, Rugged използва изключения (exceptions), може да подава неща като
`ConfigError` или `ObjectError` за да сигнализира за грешки.
Второ, няма изрично освобождаване на ресурси, понеже Ruby е garbage-collected.
Нека видим по-сложен пример: създаване на къмит от нулата

[source,ruby]
```

blob_id = repo.write("Blob contents", :blob) # <1>

index = repo.index index.read_tree(repo.head.target.tree) index.add(:path ⇒ *newfile.txt*, :oid ⇒ blob_id) # <2>

sig = { :email ⇒ "bob@example.com", :name ⇒ "Bob User", :time ⇒ Time.now, }

commit_id = Rugged::Commit.create(repo, :tree ⇒ index.write_tree(repo), # <3> :author ⇒ sig, :committer ⇒ sig, # <4> :message ⇒ "Add newfile.txt", # <5> :parents ⇒ repo.empty? ? [] : [repo.head.target].compact, # <6> :update_ref ⇒ *HEAD*, # <7>) commit = repo.lookup(commit_id) # <8>

```
<1> Създаваме нов blob, който пази съдържанието на нов файл.
<2> Попълваме индекса с дървото на къмита на head и добавяме новия файл в
пътя `newfile.txt`.
<3> Това създава ново дърво в ODB (базата данни с обекти) и го използва за
новия къмит.
<4> Използваме една и съща сигнатура за author и committer полетата.
<5> Къмит съобщението.
<6> Когато създаваме къмит, трябва да укажем родителите му.
    В случая използваме върха на HEAD за единичен родител.
<7> Rugged (и Libgit2) може по желание да обнови референция, когато се прави
къмит.
<8> Върнатата стойност е SHA-1 хеша на новия къмит обект и може да се
използва за получаване на `Commit` обект.

Ruby кодът е чист и приятен, но понеже Libgit2 върши тежката работа, той също
така ще работи и много бързо.
Ако не сте привърженик на Ruby, показваме накратко и някои други bindings в
секцията <<_libgit2_bindings>>.

==== По-сложни функционалности

Libgit2 има доста възможности, които са извън обхвата на същността на Git.
Един пример е pluggability поддръжката: Libgit2 ви позволява да подадете
специализирани ``backends'' за няколко различни типа операции, така че можете
да съхранявате неща по различен начин от Git.
Libgit2 позволява custom backends за конфигурация, съхранение на референции и
обектната база данни.

Нека видим как работи това.
Кодът отдолу е взаимстван от множеството backend примери, които екипът на
Libgit2 предоставя (на адрес https://github.com/libgit2/libgit2-backends[]).
Ето как се настройва custom backend за базата данни с обекти:

[source,c]
```

git_odb *odb; int error = git_odb_new(&odb); // <1>

git_odb_backend **my_backend; error = git_odb_backend_mine(&my_backend,** /…*/); // <2>

error = git_odb_add_backend(odb, my_backend, 1); // <3>

git_repository *repo; error = git_repository_open(&repo, "some-path"); error = git_repository_set_odb(repo, odb); // <4>

```
_Отбележете, че грешките се прихващат, но не се обработват. Надяваме се
кодът ви да е по-добър от нашия._

<1> Инициализираме празен object database (ODB) ``frontend,'' който ще служи за
контейнер за ``backend-те'', които всъщност вършат реалната работа
<2> Инициализираме custom ODB backend.
<3> Добавяме backend-а към frontend-а.
<4> Отваряме хранилище и го настройваме да използва нашата ODB за търсене на
обекти.

Какво е `git_odb_backend_mine`?
Това е конструкторът за собствената ни ODB имплементация и тук може да
правим каквото си искаме, стига да попълваме коректно структурата
`git_odb_backend`.
Ето как _би могъл_ да изглежда:

[source,c]
```

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;
}
```

```
Неуловимото ограничение тук е, че първият член на `my_backend_struct` трябва да е `git_odb_backend` структура -- това гарантира, че разположението в паметта е такова, каквото Libgit2 кода очаква.
Останалото е по избор, тази структура може да е толкова голяма или малка, колкото е нужно.

Инициализиращата функция запазва малко памет за структурата, настройва custom контекст и след това попълва членовете на `parent` структурата, която поддържа.
Погледнете файла `include/git2/sys/odb_backend.h` от сорс кода на Libgit2 за пълния набор от call signatures, вашият специфичен случай ще ви помогне да изберете коя точно ще искате да поддържате.

[[_libgit2_bindings]]
==== Други Bindings

Libgit2 има bindings за много езици.
Тук показваме малък пример за използване на някои от по-завършените (към момента на писането на книгата) bindings пакети. Библиотеки съществуват за много други платформи, включително C++, Go, Node.js, Erlang, и JVM, всяка от тях на различен етап от развитието си.
Официалната колекция bindings може да се намери като разгледате хранилищата на адрес https://github.com/libgit2[].
Кодът, който пишем ще върне къмит съобщението на къмита, към който сочи HEAD (нещо като `git log -1`).

===== LibGit2Sharp

(((.NET)))(((C#)))(((Mono)))
Ако пишете .NET или Mono приложение, LibGit2Sharp (https://github.com/libgit2/libgit2sharp[]) е нещото, което ви трябва.
Самите bindings са написани на C# и е обърнато сериозно внимание на добрата синхронизация между чистите Libgit2 повиквания с native-feeling CLR API-та.
Ето как би изглеждала примерната ни програма:

[source,csharp]
```

new Repository(@"C:\path\to\repo").Head.Tip.Message;

```
За desktop Windows приложения дори има и NuGet пакет, който помага да почнете
по-лесно.

===== objective-git

(((Apple)))(((Objective-C)))(((Cocoa)))
Ако приложението ви работи на Apple платформа, вероятно ще използвате
Objective-C като език за имплементация.
Objective-Git (https://github.com/libgit2/objective-git[]) е името на Libgit2
binding-те за тази среда.
Примерна програма:

[source,objc]
```

GTRepository *repo = [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] error:NULL]; NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];

```
Objective-git е напълно оперативно съвместим със Swift, така че не се
страхувайте, ако сте оставили Objective-C в миналото.

===== pygit2

(((Python)))
Binding-ите на Libgit2 за Python се наричат Pygit2, достъпни на
https://www.pygit2.org[].
Примерна програма:

[source,python]
```

pygit2.Repository("/path/to/repo") # отваряме хранилище .head # вземаме текущия клон .peel(pygit2.Commit) # преминаваме към къмита .message # четем съобщението

```
==== Допълнителна информация

Разбира се, пълният преглед на Libgit2 възможностите е извън обхвата на
книгата.
Ако се нуждаете от повече информация за самата Libgit2 имате API документация
на адрес https://libgit2.github.com/libgit2[], както и набор от ръководства на
https://libgit2.github.com/docs[].
За другите bindings, погледнете файла README и тестовете, често там има малки
указания и насоки за получаване на допълнителна информация.

=== JGit

(((jgit)))(((java)))
Ако искате да ползвате Git от Java програма, налична е пълнофункционалната
библиотека JGit.
JGit е Git имплементация написана на Java и е много популярна в Java
общността.
Проектът JGit е под шапката на Eclipse и е на адрес
https://www.eclipse.org/jgit/[].

==== Настройка

Има няколко начина да свържете проекта си с JGit.
Вероятно най-лесният е да използвате Maven -- интеграцията се извършва с
добавяне на следното в `<dependencies>` тага на файла pom.xml:

[source,xml]
```

<dependency> <groupId>org.eclipse.jgit</groupId> <artifactId>org.eclipse.jgit</artifactId> <version>3.5.0.201409260305-r</version> </dependency>

```
Елементът `version` вероятно ще е различен по времето, когато четете това,
проверете https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit[] за
актуална информация за хранилището.
След като това бъде направено, Maven автоматично ще намери и използва JGit
библиотеките, които ви трябват.

Ако вместо това сами управлявате binary зависимостите, компилирани JGit
binaries има на адрес https://www.eclipse.org/jgit/download[].
Може да ги интегрирате в проекта си с команди от рода на:

[source,console]
```

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 има две основни API нива: plumbing и porcelain.
Терминологията им идва от самия Git и JGit е разделена на приблизително
същите видове области: porcelain API-тата са friendly front-end за основните
user-level действия (нещата, които нормално потребителят би използвал с Git в
командния ред), докато plumbing API-тата са за директен контакт с low-level
обекти в хранилище.

Отправната точка за повечето JGit сесии е класът `Repository` и първата ни
задача е да го инстанциираме в обект.
За хранилище от файловата система (да, JGit позволява и други storage модели),
това се прави с помощта на `FileRepositoryBuilder`:

[source,java]
```

Repository newlyCreatedRepo = FileRepositoryBuilder.create(new File("/tmp/new_repo/.git")); newlyCreatedRepo.create();

Repository existingRepo = new FileRepositoryBuilder() .setGitDir(new File("my_repo/.git")) .build();

```
Builder-ът има чудесен API за да осигури всички неща, необходими за намиране
на Git хранилище без значение дали програмата ви знае къде точно се намира
то.
Може да използва environment променливи (`.readEnvironment()`), да започне от
място в работната директория и да търси (`.setWorkTree(…).findGitDir()`), или
просто да отвори известна `.git` директория.

След като вече имате инстанция на `Repository`, можете да правите всякакви
неща с обекта.
Бърз пример:

[source,java]
```

Ref master = repo.getRef("master");

ObjectId masterTip = master.getObjectId();

ObjectId obj = repo.resolve("HEAD^{tree}");

ObjectLoader loader = repo.open(masterTip); loader.copyTo(System.out);

RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1"); createBranch1.setNewObjectId(masterTip); createBranch1.update();

RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1"); deleteBranch1.setForceUpdate(true); deleteBranch1.delete();

Config cfg = repo.getConfig(); String name = cfg.getString("user", null, "name");

```
Тук има доста неща, нека ги разгледаме подред.

Първият ред взема указател към `master` референцията.
JGit автоматично намира _действителната_ `master` референция, която се пази в
`refs/heads/master`, и връща обект, който позволява да извличате информация за
нея.
Може да получите името ѝ (`.getName()`) както и целевия обект на директна
референция (`.getObjectId()`) или референцията сочена от symbolic ref
(`.getTarget()`).
Ref обектите се използват също за представяне на tag refs и objects, така че
можете да питате дали тагът е ``peeled,'' което значи че сочи към финалната
цел на (потенциално дълъг) стринг от таг обекти.

Вторият ред взема целта на `master` референцията, която се връща като ObjectId
инстанция.
ObjectId представлява SHA-1 хеш на обект, който може да съществува или не в
базата данни с обекти на Git.
Третият ред е подобен, но показва как JGit обработва rev-parse синтаксиса (за
повече информация погледнете в <<ch07-git-tools#_branch_references>>), можете да
подадете произволен object specifier, който Git разбира, и JGit ще върне или
валиден ObjectId за този обект или `null`.

Следващите два реда показват как да заредите raw съдържанието на обект.
В този пример ние извикваме `ObjectLoader.copyTo()` за да пратим съдържанието
на обекта директно към stdout, но ObjectLoader има също методи за четене на
типа и размера на обекта и може също така да ги върне като byte масив.
За големи обекти (където `.isLarge()` връща `true`), може да извикате
`.openStream()` за да получите подобен на InputStream обект способен да чете
object данни без да ги изтегля в паметта изцяло.

Следващите няколко реда показват какво е необходимо за създаване на клон.
Създаваме RefUpdate обект, конфигурираме малко параметри и извикваме
`.update()` за да активираме промяната.
Директно след това идва кодът за изтриване на същия клон.
Отбележете, че `.setForceUpdate(true)` е необходимо условие за това, в противен
случай `.delete()` повикването ще върне `REJECTED` и няма да се случи нищо.

Последният пример показва как да извлечем конфигурационна стойност на Git,
`user.name`.
Тази Config инстанция използва отвореното по-рано хранилище за локалната
конфигурация, но също така може да установи автоматично глобалните и
системни конфигурационни файлове и да чете и от тях.

Това е само малка част от plumbing API-тата, съществуват много други методи и
класове.
Тук също не показахме как JGit обработва грешки, това става с exceptions.
JGit API-тата понякога хвърлят стандартни Java exceptions (като например
`IOException`), но съществуват и JGit-специфични типове изключения (като
`NoRemoteRepositoryException`, `CorruptObjectException`, и `NoMergeBaseException`).
```

```
==== Porcelain

Plumbing API-тата са сравнително изчерпателни, но понякога може да е тромаво
да се използват за постигане на тривиални задачи като добавяне на файл в
индекса или създаването на нов къмит.
JGit предлага набор API-та от по-високо ниво посредством класа `Git`:

[source,java]
```

Repository repo; Git git = new Git(repo);

```
Git класът притежава чудесна колекция от high-level методи в _builder_ стил,
които могат да се използват за реализиране на доволно сложни сценарии.
Да видим пример -- ще направим нещо еквивалентно на `git ls-remote`:

[source,java]
```

CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw0rd"); Collection<Ref> remoteRefs = git.lsRemote() .setCredentialsProvider(cp) .setRemote("origin") .setTags(true) .setHeads(false) .call(); for (Ref ref : remoteRefs) { System.out.println(ref.getName() + " → " + ref.getObjectId().name()); }

```
Това е стандартно използване на Git класа, методите връщат команден обект,
който ви позволява да правите chaining на повикванията им за да задавате
параметри и да ги стартирате с `.call()`.
В този случай ние питаме `origin` референцията за тагове, но не и за heads.
Също така, обърнете внимание на обекта `CredentialsProvider`, който се ползва
за автентикация.

Много други команди са достъпни през класа Git, включително `add`, `blame`,
`commit`, `clean`, `push`, `rebase`, `revert`, и `reset`.

==== Повече информация

Това е само малка демонстрация на възможностите на JGit.
Ако се интересувате и искате да научите повече, ето къде да потърсите
допълнителна информация и вдъхновение:

* Официалната JGit API документация на адрес
https://www.eclipse.org/jgit/documentation/[].
  Това са стандартни Javadoc, така че любимата ви JVM IDE ще може да ги
инсталира и локално.
* Хранилището JGit Cookbook на адрес https://github.com/centic9/jgit-cookbook[]
съдържа много примери за извършване на специфични дейности с JGit.

=== go-git

(((go-git)))(((Go)))
В случай, че желаете да интегрирате Git в услуга написана на Go, съществува
pure Go библиотечна имплементация.
Тази имплементация няма никакви нативни зависимости и по тази причина е
неподатлива на manual memory management грешки.
Също така е прозрачна за стандартните Golang performance analysis
инструментариуми като CPU, Memory profilers, race detector, и т.н.

go-git акцентира върху разширяемостта и съвместимостта и поддържа повечето
plumbing APIs, документирани на https://github.com/go-git/go-
git/blob/master/COMPATIBILITY.md[].

Ето прост пример за използването на Go APIs:

[source, go]
```

import "github.com/go-git/go-git/v5"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{ URL: "https://github.com/go-git/go-git", Progress: os.Stdout, })

```
След като имате инстанция `Repository`, можете да четете информация и да
правите промени по нея:

[source, go]
```

ref, err := r.Head()

commit, err := r.CommitObject(ref.Hash())

history, err := commit.History()

for _, c := range history { fmt.Println(c) }

```
==== Разширена функционалност

go-git има няколко advanced възможности, които си заслужава да се посочат.
Една от тях е pluggable storage системата, подобна на Libgit2 backend-ите.
Имплементацията по подразбиране е in-memory storage и тя е много бърза.

[source, go]
```

r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{ URL: "https://github.com/go-git/go-git", })

```
Pluggable storage системата осигурява много интересни опции.
Например, https://github.com/go-git/go-git/tree/master/_examples/storage[]
позволява да съхранявате референции, обекти и конфигурационни настройки в
Aerospike база данни.

Друга функция е гъвкавата абстракция на файловата система.
Посредством https://pkg.go.dev/github.com/go-git/go-billy/v5?tab=doc#Filesystem[] е
лесно да се съхраняват всички файлове по различен начин, например като се
пакетират в единичен архив на диска или като се запазват всички in-memory.

Друг advanced use-case включва HTTP клиент с фина настройка, като този от
https://github.com/go-git/go-git/blob/master/_examples/custom_http/main.go[].

[source, go]
```

customClient := &http.Client{ Transport: &http.Transport{ // accept any certificate (might be useful for testing) TLSClientConfig: &tls.Config{InsecureSkipVerify: true}, }, Timeout: 15 * time.Second, // 15 second timeout CheckRedirect: func(req *http.Request, via []*http.Request) error { return http.ErrUseLastResponse // don’t follow redirect }, }

client.InstallProtocol("https", githttp.NewClient(customClient))

r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})

```
==== Допълнителна информация

Извън обхвата на тази книга е да разглеждаме всички поддържани от go-git възможности.
Ако се нуждаете от подробности, налична е API документация на адрес
https://pkg.go.dev/github.com/go-git/go-git/v5[], както и комплект от примери на
https://github.com/go-git/go-git/tree/master/_examples[].

=== Dulwich

(((Dulwich)))(((Python)))
Съществува и Git имплементация за Python - Dulwich.
Проектът се хоства на адрес https://www.dulwich.io/
Целта му е да предостави интерфейс за достъп до git хранилища (локални и отдалечени), който не се обръща към git директно, а вместо това използва чист Python.
Той също съдържа и опционални C разширения, които значително подобряват производителността.

Dulwich следва дизайна на git и разделя двете базови API: plumbing и porcelain.

Ето пример за използване на API от по-ниско ниво за достъп до съобщението на последния къмит:

[source, python]
```

from dulwich.repo import Repo r = Repo(.) r.head() # *57fbe010446356833a6ad1600059d80b1e731e15*

c = r[r.head()] c # <Commit 015fc1267258458901a94d228e39f0a378370466>

c.message # *Add note about encoding.\n*

```
За да отпечатате commit log чрез porcelain API-то от по-високо ниво, може да използвате:

[source, python]
```

from dulwich import porcelain porcelain.log(., max_entries=1)

#commit: 57fbe010446356833a6ad1600059d80b1e731e15 #Author: Jelmer Vernooij <jelmer@jelmer.uk> #Date: Sat Apr 29 2017 23:57:34 +0000

```
==== Допълнителна информация

 * Официалната API документация е налична на
https://www.dulwich.io/apidocs/dulwich.html[].
 * Официалното ръководство https://www.dulwich.io/docs/tutorial[] предлага много примери за това как да извършвате специфични задачи в Dulwich.
```

```
[[C-git-commands]]
[appendix]
== Git команди

В книгата представихме много Git команди и се опитвахме това да бъде
възможно най-информативно като постепенно добавяхме нови такива.
Обаче, това ни оставя с примери разпръснати из цялото съдържание.

В това приложение ще преминем накратко през всички изброени дотук Git
команди и ще се постараем да ги групираме, доколкото е възможно, според
предназначението им.
Ще говорим за това какво е общото предназначение на всяка команда и след
това ще посочваме къде в книгата може да намерите примери с нея.

[TIP]
====
Можете да съкращавате дългите опции.
Например, може да напишете `git commit --a`, което е еквивалентно на `git
commit --amend`.
Това работи само, ако символите след `--` са уникални за опцията.
Използвайте пълната опция в скриптове.
====

=== Настройки и конфигурация

Две от командите в Git се използват почти ежедневно, `config` и `help`.

==== git config

Git има начини за изпълнение по подразбиране на стотици операции.
За много от тях, можете да инструктирате Git да прави по подразбиране нещата
по малко по-различен начин.
Това включва всичко, от това да кажете на Git какво е името ви до това да
укажете специфични цветове в терминала или кой е редакторът ви, който
предпочитате.
Тази команда чете и пише в няколко различни файла, така че можете да
задавате стойности глобално или за определени хранилища.

Командата `git config` се използва в почти всяка глава от книгата.

В <<ch01-getting-started#_first_time>> я ползвахме за задаване на име и имейл на
потребителя, както и за указване на текстовия ни редактор -- преди още да
бяхме започнали да използваме Git.

В <<ch02-git-basics-chapter#_git_aliases>> показахме как бихте могли да я
използвате за създаването на съкратени варианти на команди, които автоматично
се разширяват до пълните еквиваленти, така че да не се налага да ги въвеждате
изцяло всеки път.
```

```
В <<ch03-git-branching#_rebasing>> използвахме командата за да направим `--
rebase` опция по подразбиране при изпълнение на `git pull`.

В <<ch07-git-tools#_credential_caching>> я използвахме за определяне на default
store за HTTP пароли.

В <<ch08-customizing-git#_keyword_expansion>> показахме как се създават smudge and
clean филтри за съдържанието влизащо или излизащо от Git.

Почти всичко в <<ch08-customizing-git#_git_config>> е посветено на тази команда.

[[_core_editor]]
==== git config core.editor команди

Съгласно конфигурационните инструкции в <<ch01-getting-started#_editor>>, много
редактори могат да се настроят както следва:

.Пълен списък на `core.editor` конфигурационните команди
[cols="1,2",options="header"]
|==============================
|Редактор | Конфигурационна команда
|Atom |`git config --global core.editor "atom --wait"`
|BBEdit (Mac, with command line tools) |`git config --global core.editor "bbedit -w"`
|Emacs |`git config --global core.editor emacs`
|Gedit (Linux) |`git config --global core.editor "gedit --wait --new-window"`
|Gvim (Windows 64-bit) |`git config --global core.editor "'C:\Program
Files\Vim\vim72\gvim.exe' --nofork '%*'"` (вижте забележката отдолу)
|Kate (Linux) |`git config --global core.editor "kate"`
|nano |`git config --global core.editor "nano -w"`
|Notepad (Windows 64-bit) |`git config core.editor notepad`
|Notepad++ (Windows 64-bit) |`git config --global core.editor "'C:\Program
Files\Notepad++\notepad++.exe' -multiInst -notabbar -nosession -noPlugin"` (вижте
забележката отдолу)
|Scratch (Linux)|`git config --global core.editor "scratch-text-editor"`
|Sublime Text (macOS) |`git config --global core.editor "/Applications/Sublime\
Text.app/Contents/SharedSupport/bin/subl --new-window --wait"`
|Sublime Text (Windows 64-bit) |`git config --global core.editor "'C:\Program
Files\Sublime Text 3\sublime_text.exe' -w"` (вижте забележката отдолу)
|TextEdit (macOS)|`git config --global --add core.editor "open -W -n"`
|Textmate |`git config --global core.editor "mate -w"`
|Textpad (Windows 64-bit) |`git config --global core.editor "'C:/Program Files/TextPad
5/TextPad.exe' -m` (вижте забележката отдолу)
|UltraEdit (Windows 64-bit) | `git config --global core.editor Uedit32`
|Vim |`git config --global core.editor "vim"`
|Visual Studio Code |`git config --global core.editor "code --wait"`
|VSCodium (Free/Libre Open Source Software Binaries of VSCode) | `git config --global
core.editor "codium --wait"`
|WordPad |`git config --global core.editor '"C:\Program Files\Windows
NT\Accessories\wordpad.exe"'"`
|Xi | `git config --global core.editor "xi --wait"`
|==============================
```

```
[NOTE]
====
Ако имате 32-битов редактор на 64-битов Windows, програмата ще бъде
инсталирана в `C:\Program Files (x86)\` вместо в `C:\Program Files\`.
====

==== git help

Командата `git help` се използва за показване на документацията на командите в
Git.
Ние правим кратък преглед тук, но ако искате пълния списък с всички възможни
аргументи и флагове за коя да е команда, можете винаги да изпълните `git help
<command>`.

Представихме `git help` в <<ch01-getting-started#_git_help>> и показахме как да я
използвате за да намерите повече информация за `git shell` в <<ch04-git-on-the-
server#_setting_up_server>>.


=== Издърпване и създаване на проекти

Има два начина за сдобиване с Git хранилище.
Единият е да го копираме от налично такова в мрежата или където и да се
намира. Другият е да си го създадем от налична директория.

==== git init

За да вземем произволна директория и да я превърнем в Git хранилище, просто
изпълняваме `git init`.

Показахме това първо в <<ch02-git-basics-chapter#_getting_a_repo>>, където
създадохме ново хранилище, по което да работим.

Накратко споменахме как можем да сменим името на клона по подразбиране от
``master'' до друго такова в <<ch03-git-branching#_remote_branches>>.

Използваме тази команда и за създаване на празно bare хранилище на сървъра в
<<ch04-git-on-the-server#_bare_repo>>.

Последно, погледнахме какво командата прави задкулисно в <<ch10-git-
internals#_plumbing_porcelain>>.

==== git clone

Командата `git clone` по същество е нещо като wrapper около няколко други
команди.
Тя създава нова директория, влиза в нея и изпълнява `git init` за да създаде
празно Git хранилище, след това добавя remote (`git remote add`) към URL-а,
който ѝ подавате (по подразбиране с име `origin`), изпълнява `git fetch` от
това отдалечено хранилище и извлича в работната ви директория най-новия
```

```
къмит с `git checkout`.

`git clone` се използва на цял куп места в книгата, ще изброим само най-
интересните.

Командата бе представена и обяснена в <<ch02-git-basics-chapter#_git_cloning>>,
където дадохме няколко примера.

В <<ch04-git-on-the-server#_getting_git_on_a_server>> погледнахме опцията `--bare`
за създаване на Git хранилище без работна директория.

В <<ch07-git-tools#_bundling>> я използвахме за да възстановим пакетирано Git
хранилище.

В <<ch07-git-tools#_cloning_submodules>> научихме за опцията `--recurse-
submodules`, с чиято помощ улесняваме клонирането на хранилище с подмодули.

Въпреки, че командата се използва на много други места в книгата,
гореизброените са тези, при които тя се използва по по-различен и специфичен
начин.


=== Snapshotting

За основния работен процес по индексиране на съдържание и къмитването му в
историята, съществуват само няколко основни команди.

==== git add

Командата `git add` добавя съдържание от работната директория в staging area
(или ``индексната област'') за следващия къмит.
Когато се изпълни `git commit`, по подразбиране тя гледа какво има само в
индекса, така `git add` се използва за определяне на това какво точно искате да
включите в следващия snapshot.

Това е много важна за Git команда и се споменава десетки пъти в книгата.
Ето по-важните места.

Представихме `git add` първо в <<ch02-git-basics-chapter#_tracking_files>>.

Показахме как да я използваме за разрешаване на конфликти при сливане в
<<ch03-git-branching#_basic_merge_conflicts>>.

Демонстрирахме как да я използваме интерактивно за да индексираме само
специфични части от модифициран файл в <<ch07-git-tools#_interactive_staging>>.

Последно, емулирахме я на ниско ниво в <<ch10-git-internals#_tree_objects>>, така
че да получите представа какво се случва под повърхността.

==== git status
```

```
Командата `git status` ще ви покаже различните статуси на файловете в
работната директория и индексната област.
Кои файлове са променени и неиндексирани и кои са индексирани, но все още не
са къмитнати.
В нормалната си форма, тя също така подава основни съвети за това как да
премествате файлове между тези етапи.

Видяхме `status` за първи път в <<ch02-git-basics-chapter#_checking_status>>.
Използваме я почти навсякъде в книгата, но почти всичко, което можете да
вършите с нея е обяснено там.

==== git diff

Командата `git diff` се използва, когато искате да видите разликите между кои
да е две дървета.
Това би могло да бъде разликата между работната област и индексната (което
прави `git diff` без аргументи), между индексната област и последния къмит
(`git diff --staged`), или между два къмита (`git diff master branchB`).

За пръв път срещнахме `git diff` в <<ch02-git-basics-chapter#_git_diff_staged>>,
където показахме как да видим кои промени са индексирани и кои все още не
са.

Използвахме я и за търсене на възможни whitespace проблеми преди къмитване с
опцията `--check` в <<ch05-distributed-git#_commit_guidelines>>.

Видяхме как да проверим за разлики между клонове по ефективен начин
използвайки синтаксиса `git diff A...B` в <<ch05-distributed-
git#_what_is_introduced>>.

Използвахме я за да филтрираме whitespace разлики с флага `-b` а също и за
сравнение на различни етапи от конфликтни файлове с `--theirs`, `--ours` и `--
base` в <<ch07-git-tools#_advanced_merging>>.

Накрая, използвахме я за ефективно сравнение на submodule промени със `--
submodule` опцията в <<ch07-git-tools#_starting_submodules>>.

==== git difftool

Командата `git difftool` просто стартира външен инструмент за показване на
разликите между две дървета, в случай че предпочитате нещо различно от
вградената `git diff`.

Споменахме я в <<ch02-git-basics-chapter#_git_diff_staged>>.

==== git commit

Командата `git commit` взема съдържанието на всички файлове индексирани
преди това с `git add` и записва нов перманентен snapshot в базата данни, след
което премества указателя на текущия клон към него.
```

```
Основите на къмитването показахме в <<ch02-git-basics-
chapter#_committing_changes>>.
Там също така показахме как с удобния флаг `-a` можем да прескочим `git add`
стъпката в ежедневния работен процес и как с `-m` да подадем къмит
съобщението директно от командния ред, вместо да пускаме редактора.

Във <<ch02-git-basics-chapter#_undoing>> разгледахме опцията `--amend` за да
направим отново последния къмит.

В <<ch03-git-branching#_git_branches_overview>> навлязохме в повече детайли за
това какво прави `git commit` и защо го прави по дадения начин.

Видяхме как да подписваме къмити криптографски с флага `-S` в <<ch07-git-
tools#_signing_commits>>.

Последно, погледнахме какво прави зад кулисите `git commit` и как това е
имплементирано в <<ch10-git-internals#_git_commit_objects>>.

==== git reset

Командата `git reset` се използва основно за отмяна на действия, както
подсказва името ѝ.
Тя премества указателя на `HEAD` и по избор може да променя индексната
област, както и работната директория с опцията `--hard`.
Тази последна опция дава възможност за загуба на данни, ако се използва
неправилно, така че трябва да я разберете добре преди да я ползвате.

За пръв път се срещнахме с простата форма на `git reset` в <<ch02-git-basics-
chapter#_unstaging>>, където я използвахме за да извадим от индекса файл, върху
който е изпълнена `git add`.

След това я разгледахме в повече детайли в <<ch07-git-tools#_git_reset>>,
секцията беше изцяло посветена на нея.

Използвахме `git reset --hard` за да отменим сливане в <<ch07-git-
tools#_abort_merge>>, където също така използвахме и `git merge --abort`, която е
един вид wrapper за `git reset`.

==== git rm

Използваме `git rm` за изваждане на файлове от индексната област и работната
директория в Git.
Тя е подобна на `git add` в това, че индексира файл (само че за изваждане) от
следващия къмит.

Погледнахме я в малко детайли в <<ch02-git-basics-chapter#_removing_files>>,
включително за рекурсивно изтриване на файлове и също така за изтриване на
файлове само от индекса, но не и от работната директория с опцията `--cached`.

Единственият вид различно приложение на `git rm` в книгата е в <<ch10-git-
internals#_removing_objects>>, където използвахме и обяснихме опцията `--ignore-
```

```
unmatch` при изпълнение на `git filter-branch`, която просто игнорира грешката,
ако файлът който се опитваме да изтрием не съществува.
Това може да е полезно за ползване в скриптове.

==== git mv

Командата `git mv` е просто удобен заместител за ситуациите, когато трябва да
преместите файл, да изпълните `git add` за новия файл и след това `git rm` за
стария.

Само я споменахме накратко в <<ch02-git-basics-chapter#_git_mv>>.

==== git clean

Командата `git clean` се използва за премахване на нежелани файлове от
работната директория.
Това може да включва премахването на временни обекти от компилиране или
merge conflict файлове.

Разглеждаме много от опциите и сценариите, в които може да се използва clean
командата в <<ch07-git-tools#_git_clean>>.

=== Клонове и сливане

Няколко команди имплементират по-голямата част от branching и merging
функционалностите в Git.

==== git branch

`git branch` по същество е инструмент за управление на клоновете в Git.
Може да създава, изброява, изтрива и преименува клонове.

Повечето от написаното в <<ch03-git-branching#ch03-git-branching>> е посветено на
`branch` командата и тя се използва в цялата глава.
Първо я представихме в <<ch03-git-branching#_create_new_branch>> и преминахме
през повечето ѝ възможности (печатане и изтриване) в <<ch03-git-
branching#_branch_management>>.

В <<ch03-git-branching#_tracking_branches>> използваме `git branch -u` за да укажем
tracking клон.

Последно, погледнахме какво прави тя на заден план в <<ch10-git-
internals#_git_refs>>.

==== git checkout

Командата `git checkout` се използва за превключване на клонове и за извличане
на съдържание в работната директория.

За пръв път я срещнахме в <<ch03-git-branching#_switching_branches>> заедно с
командата `git branch`.
```

```
Видяхме как да я използваме за да започнем да следим клонове с флага `--
track` в <<ch03-git-branching#_tracking_branches>>.

Използваме я за повторно въвеждане на конфликти във файлове с опцията `--
conflict=diff3` в <<ch07-git-tools#_checking_out_conflicts>>.

Навлизаме в по-дълбоки подробности за връзката ѝ с `git reset` в <<ch07-git-
tools#_git_reset>>.

Детайли по имплементацията ѝ показахме в <<ch10-git-internals#ref_the_ref>>.

==== git merge

Инструментът `git merge` се използва за сливане на един или повече клонове в
клона, който е текущо извлечен.
След сливането, текущият клон се премества напред с резултата от сливането.

Командата `git merge` видяхме за пръв път в <<ch03-git-
branching#_basic_branching>>.
Въпреки, че се използва на различни места в книгата, тя има много малко на
брой вариации -- в общи линии само `git merge <branch>` с името на единичен
клон, който искаме да слеем.

Видяхме как се прави squashed сливане (където Git слива работата, но го прави
под формата на единичен обединяващ къмит, вместо да записва цялата история
на сливащия се клон) в края на <<ch05-distributed-git#_public_project>>.

Преминахме през доста детайли по merge процеса и самата команда, включително
`-Xignore-space-change` командата и флага `--abort` за отказ на проблематично
сливане в <<ch07-git-tools#_advanced_merging>>.

Видяхме как да проверяваме сигнатури преди сливане, ако проектът ви използва
GPG подписване в <<ch07-git-tools#_signing_commits>>.

Накрая научихме за Subtree сливането в <<ch07-git-tools#_subtree_merge>>.

==== git mergetool

Командата `git mergetool` служи за стартиране на външен merge helper в случай,
че не харесвате вграденото сливане в Git.

Погледнахме го набързо в <<ch03-git-branching#_basic_merge_conflicts>> и
навлязохме в детайли за това как да имплементирате собствен външен merge tool
във <<ch08-customizing-git#_external_merge_tools>>.

==== git log

Командата `git log` се използва за показване на достъпна записана история на
проект от най-късно записания къмит snapshot назад.
Тя по подразбиране ще покаже само историята на текущия клон, но може да ѝ
```

```
се подадат различни или дори повече heads или клонове, от които да трасира.
Също често се използва за показване на разлики между два или повече клона на
ниво къмит.

Тази команда се среща в почти всяка глава в книгата за демонстрация на
различни аспекти от историята на проект.

Представихме я първо в <<ch02-git-basics-chapter#_viewing_history>>.
Там разгледахме аргументите ѝ `-p` и `--stat` за да получим представа какво е
било въведено във всеки къмит, както и `--pretty` и `--oneline` опциите за да
видим историята по-стегнато заедно с някои прости опции за филтриране по
дата и автор.

В <<ch03-git-branching#_create_new_branch>> я използваме с аргумента `--decorate`
за лесно визуализиране на това къде сочат указателите на клоновете, а също и
с опцията `--graph` за да видим как изглеждат разклонени истории.

В <<ch05-distributed-git#_private_team>> и <<ch07-git-tools#_commit_ranges>>
обяснихме синтаксиса `branchA..branchB` за да инструктираме `git log` да намери
кои къмити са уникални за даден клон в сравнение с друг клон.
В <<ch07-git-tools#_commit_ranges>> разгледахме това сравнително обстойно.

В <<ch07-git-tools#_merge_log>> и <<ch07-git-tools#_triple_dot>> използваме формата
`branchA...branchB` и `--left-right` синтаксиса, за да видим какво е налично в
един от двата клона, но не и в двата едновременно.
В <<ch07-git-tools#_merge_log>> също видяхме как да използваме `--merge` опцията
за помощ при изследване на merge конфликти, както и `--cc` опцията за търсене
на merge commit конфликти в историята.

В <<ch07-git-tools#_git_reflog>> демонстрирахме флага `-g` за да видим Git
reflog-а през този инструмент вместо да правим branch traversal.

В <<ch07-git-tools#_searching>> използвахме флаговете `-S` и `-L` за търсене на
неща, които са се случили хронологично в кода във времето, например историята
на дадена функция.

В <<ch07-git-tools#_signing_commits>> виждаме как да използваме `--show-signature`
за да добавим верификационен стринг към всеки къмит в изхода от `git log` в
зависимост от това дали той е валидно подписан или не.

==== git stash

Командата `git stash` се използва за временно съхранение/скриване на
некъмитната работа с цел да се изчисти работната директория без да е
необходимо къмитване на недовършената работа в клона.

Това е обяснено изцяло в <<ch07-git-tools#_git_stashing>>.

==== git tag

`git tag` командата предоставя възможност за създаване на перманентен маркер
```

```
(bookmark) към специфична точка в историята на кода.
В общи линии се използва за маркиране на неща от рода на releases.

Тази команда е представена и разгледана в подробности в <<ch02-git-basics-
chapter#_git_tagging>> и я използваме в действие в <<ch05-distributed-
git#_tagging_releases>>.

Погледнахме как да създадем GPG signed таг с флага `-s` и също така как да
проверим такъв с `-v` в <<ch07-git-tools#_signing>>.


=== Споделяне и обновяване на проекти

Не са много командите в Git, които правят мрежови операции, почти всички
работят с локалната база данни.
Когато сте готови да споделите работата си обаче, ето какви са възможностите:

==== git fetch

`git fetch` комуникира с отдалечено хранилище и издърпва всичката информация
от него, която липсва в локалната база данни.

Представихме я във <<ch02-git-basics-chapter#_fetching_and_pulling>> и
продължихме с примери за нея в <<ch03-git-branching#_remote_branches>>.

Използвахме я и в няколко примера в <<ch05-distributed-
git#_contributing_project>>.

Използваме я за издърпваме на конкретна референция, която е извън областта
по подразбиране в <<ch06-github#_pr_refs>> и видяхме как да издърпваме от bundle
в <<ch07-git-tools#_bundling>>.

Направихме строго потребителски refspecs с цел да накараме `git fetch` да
направи нещо малко по-различно от подразбиращото се в <<ch10-git-
internals#_refspec>>.

==== git pull

Командата `git pull` по същество е комбинация от `git fetch` и `git merge`
командите като Git ще се опита да изтегли и слее промените от отдалеченото
хранилище в една стъпка.

Представихме я първоначално във <<ch02-git-basics-chapter#_fetching_and_pulling>>
и демонстрирахме как да видим какво ще бъде слято, ако бъде изпълнена в
<<ch02-git-basics-chapter#_inspecting_remote>>.

Видяхме как да я използваме в помощ при rebasing затруднения в <<ch03-git-
branching#_rebase_rebase>>.

Показваме как да я използваме с URL за да интегрираме промени в one-off
маниер в <<ch05-distributed-git#_checking_out_remotes>>.
```

```
Набързо отбелязваме, че може да използвате опцията `--verify-signatures` за да
проверите дали къмитите, които интегрирате са били GPG подписани в <<ch07-git-
tools#_signing_commits>>.

==== git push

Командата `git push` използваме за комуникация с отдалечено хранилище, за
изчисляване и изпращане към него на информацията, която е налична локално,
но не присъства на отдалечения сървър.
Тя изисква права за запис в отдалеченото хранилище и нормално се използва с
някакъв вид автентикация.

За пръв път се срещнахме с `git push` в <<ch02-git-basics-
chapter#_pushing_remotes>>.
Там видяхме основите на публикуването на клонове в отдалечени хранилища.
В <<ch03-git-branching#_pushing_branches>> навлязохме малко по-дълбоко в
публикуването на специфични клонове, а в <<ch03-git-
branching#_tracking_branches>> видяхме как да направим tracking клонове, към
които да публикуваме автоматично.
В <<ch03-git-branching#_delete_branches>> използвахме флага `--delete` за да
изтрием клон от сървъра с `git push`.

В <<ch05-distributed-git#_contributing_project>> видяхме множество примери на
използване на командата за споделяне на работа с множество отдалечени
хранилища.

Научихме как да я използваме за споделяне на тагове с опцията `--tags` в
<<ch02-git-basics-chapter#_sharing_tags>>.

В <<ch07-git-tools#_publishing_submodules>> използвахме опцията `--recurse-
submodules` за да проверим дали цялата ни работа по подмодулите е била
публикувана преди да публикуваме основния проект. Това може да е много
полезно, когато използваме подмодули.

В <<ch08-customizing-git#_other_client_hooks>> говорихме за `pre-push` hook скрипта,
който можем да настроим да се стартира преди публикуването да завърши и да
проверява дали съдържанието е валидно.

Последно, в <<ch10-git-internals#_pushing_refspecs>> видяхме как се публикува с
пълни refspec, вместо с общоприетите shortcuts, които се използват нормално.
Това може да ви помогне да сте много прецизни в това каква точно работа
искате да публикувате.

==== git remote

`git remote` е управляващ инструмент за данните за отдалечени хранилища.
Позволява да изписвате дългите URL-и под формата на съкратени описателни
имена като например ``origin''.
Може да имате много такива и `git remote` се използва за добавяне, редактиране
и изтриването им.
```

```
Командата разгледахме в подробности в <<ch02-git-basics-chapter#_remote_repos>>,
включително показване, добавяне, изтриване и преименуване на отдалечени
референции.

Тя също се използва почти във всяка глава в книгата, но винаги в стандартния
формат `git remote add <name> <url>`.

==== git archive

Командата `git archive` използваме за създаване на архивен файл от специфичен
snapshot на проекта.

Използвахме `git archive` за да създадем tarball на проект, подходящ за
споделяне в <<ch05-distributed-git#_preparing_release>>.

==== git submodule

Командата `git submodule` се използва за управление на външни хранилища в
рамките на основния проект.
Може да се използва за библиотеки или други споделени ресурси.
`submodule` има няколко подкоманди като (`add`, `update`, `sync`) за
управлението им.

Присъства и се обяснява изцяло само в <<ch07-git-tools#_git_submodules>>.

=== Инспекция и сравнение

==== git show

Командата `git show` може да покаже Git обект в опростен и по-интуитивен вид.
Нормално се използва за показване на информация за таг или къмит.

Използвахме я първо в <<ch02-git-basics-chapter#_annotated_tags>> за да покажем
информация за анотирани тагове.

По-късно я използвахме често в <<ch07-git-tools#_revision_selection>> за да
изведем къмитите, към които се отнасят конкретни ревизии на проекта.

Едно от по-интересните неща, които направихме с `git show` беше в <<ch07-git-
tools#_manual_remerge>>, когато я използвахме за да извлечем специфично
съдържание от файлове от различни етапи по време на merge конфликт.

==== git shortlog

`git shortlog` се използва за съкратено обобщение на изхода от `git log`.
Тя приема много от аргументите на `git log`, но вместо да извежда всички
къмити, ще отпечата обобщение на къмитите групирани по автор.

Показахме как да я използваме за да създадем прегледен changelog в <<ch05-
distributed-git#_the_shortlog>>.
```

```
==== git describe

Командата `git describe` приема Git обект и извежда стринг, който е четим за
хората и не се променя.
Това е начин да получим описание на къмит, което да е уникално като SHA-1
хеша, но малко по-разбираемо.

Използвахме `git describe` в <<ch05-distributed-git#_build_number>> и <<ch05-
distributed-git#_preparing_release>> за да получим стринг, с който да именуваме
нашия release файл.


=== Дебъгване

Git има няколко команди, които могат да бъдат от помощ при дебъгване на
проблеми в кода.
Те помагат да определите къде нещо е било въведено и кой го е въвел.

==== git bisect

`git bisect` е много полезен инструмент за дебъгване, използва се за да
определим кой специфичен къмит първи е въвел определена грешка в кода чрез
автоматично двоично търсене.

Среща се в <<ch07-git-tools#_binary_search>>, където е обяснен напълно.

==== git blame

Командата `git blame` анотира редовете на произволен файл с информация за
това кой къмит последно е направил промяна по тях и кой е разработчика
направил промените.
Това е удобно, когато трябва да намерите човека, когото да питате за повече
информация по специфична секция от кода.

Среща се единствено в <<ch07-git-tools#_file_annotation>>.

==== git grep

Командата `git grep` може да използвате за да търсите за произволен стринг
или регулярен израз в произволен файл от кода, дори и в по-стари версии на
проекта.

Използваме я единствено в <<ch07-git-tools#_git_grep>>.

=== Patching

Има няколко команди в Git обединени около концепцията да възприемаме
къмитите от гледна точка на това какви промени въвеждат, тоест като серия от
пачове.
Тези команди помагат да управлявате клоновете си в подобен маниер.
```

```
==== git cherry-pick

`git cherry-pick` взема промяната въведена в единичен къмит и се опитва да я
въведе наново като нов къмит в текущия клон.
Това може да е полезно в случай, че искате да вземете един или два къмита
индивидуално от даден клон вместо да го слеете целия, което взема всичките
му промени.

Cherry picking процесът е обяснен и демонстриран в <<ch05-distributed-
git#_rebase_cherry_pick>>.

==== git rebase

Командата `git rebase` в общи линии е автоматична версия на `cherry-pick`.
Тя определя серия от къмити и след това ги cherry-pick-ва един по един в същия
ред на друго място.

Пребазирането е разгледано в детайли в <<ch03-git-branching#_rebasing>>, където
освен това обръщаме внимание на възможните проблеми, които възникват в
резултат на пребазиране на вече публикувани клонове.

Използваме я също в примера за разделяне на историята в две отделни
хранилища в <<ch07-git-tools#_replace>> с помощта на флага `--onto`.

Разглеждаме и ситуации, при които може да се окажем в merge конфликт по
време на пребазиране в <<ch07-git-tools#ref_rerere>>.

Използваме я в интерактивен скриптов режим с опцията `-i` в <<ch07-git-
tools#_changing_multiple>>.

==== git revert

Командата `git revert` по същество е обратна на `git cherry-pick`.
Тя създава нов къмит, който прилага точно обратното на промените въведени в
посочения къмит и реално ги отменя.

Използваме я във <<ch07-git-tools#_reverse_commit>> за да отменим merge къмит.

=== Email команди

Много Git проекти, включително и самия Git, се поддържат през мейлинг листи.
Git разполага с вградени инструменти за улеснение на процеса -- от такива за
генериране на пачове, които можете да изпратите по пощата, до такива за
прилагане на пачовете от имейл съобщение.

==== git apply

`git apply` прилага пач създаден с `git diff` или дори с GNU diff командите.
Тя е подобна на възможностите на `patch` командата с някои малки разлики.
```

```
Демонстрирахме я и показахме при какви обстоятелства е подходяща в <<ch05-
distributed-git#_patches_from_email>>.

==== git am

Командата `git am` се използва за прилагане на пачове от email inbox с mbox
формат.
Полезна е за получаване и лесно прилагане на пачове по имейл.

Видяхме използването ѝ и възможна работна последователност в <<ch05-
distributed-git#_git_am>> включително с опциите ѝ `--resolved`, `-i` и `-3`.

Съществуват също няколко hooks, които може да използвате в помощ на работния
процес с `git am`, разгледани са в <<ch08-customizing-git#_email_hooks>>.

Също така я използваме за прилагане на форматирани като пач GitHub Pull
Request промени в <<ch06-github#_email_notifications>>.

==== git format-patch

`git format-patch` се използва за генериране на серии от пачове в mbox формат,
които може да използвате за изпращане към мейлинг лист със съответното
коректно форматиране.

Разгледахме пример за сътрудничество в проект с `git format-patch` инструмента
в <<ch05-distributed-git#_project_over_email>>.

==== git imap-send

`git imap-send` командата качва информация генерирана с `git format-patch` в IMAP
drafts папка.

Видяхме пример за сътрудничество в проект чрез изпращане на пачове с `git
imap-send` в <<ch05-distributed-git#_project_over_email>>.

==== git send-email

`git send-email` се използва за изпращане по имейл на пачове генерирани с `git
format-patch`.

Отново, пример за сътрудничество по проект с нейна помощ разгледахме в
<<ch05-distributed-git#_project_over_email>>.

==== git request-pull

Командата `git request-pull` просто генерира примерно тяло на имейл
съобщение, което може да изпратите някому.
Ако имате клон на публичен сървър и искате да покажете на някого как да
интегрира промените ви без изпращане на пачове по имейл, може да изпълните
тази команда и да изпратите изхода ѝ до колегата.
```

```
Демонстрирахме това генерирайки pull message в <<ch05-distributed-
git#_public_project>>.

=== Външни системи

Git има няколко команди подпомагащи интеграцията с други version control
системи.

==== git svn

`git svn` се използва за комуникация със Subversion като клиент.
Това значи, че може да използвате Git за да извличате съдържание и да
изпращате къмити към Subversion сървър.

Командата е подробно разгледана в <<ch09-git-and-other-systems#_git_svn>>.

==== git fast-import

За други видове системи или за импорт от почти всякакъв формат, може да
използвате `git fast-import`, която бързо преработва чуждия формат до нещо,
което Git може лесно да запише.

Разглеждаме я в <<ch09-git-and-other-systems#_custom_importer>>.

=== Административни команди

Ако администрирате Git хранилище или трябва да поправите нещо по драстичен
начин, ето какви команди са налични.

==== git gc

`git gc` изпълнява ``garbage collection'' процедури върху вашето хранилище
премахвайки ненужни файлове от базата данни и пакетирайки останалите
файлове в по-ефективен формат.

Тя нормално работи незабележимо на заден план, въпреки че може да се пуска
и ръчно, ако желаете.
Видяхме някои примери в <<ch10-git-internals#_git_gc>>.

==== git fsck

Командата `git fsck` се използва за проверка на вътрешната база данни за
проблеми или несъответствия.

Използвахме я само във <<ch10-git-internals#_data_recovery>> за търсене на т.нар.
dangling обекти.

==== git reflog

`git reflog` преглежда дневника на това къде са били указателите на клоновете
ви във времето за да намери евентуално загубени къмити -- нещо, което може
```

```
да се случи при пренаписване на историите.

Работихме с тази команда главно в <<ch07-git-tools#_git_reflog>>, където също
видяхме как бихме могли да използваме `git log -g` за да получим същата
информация форматирана като `git log` изход.

Показахме практически пример за възстановяване на подобен загубен клон във
<<ch10-git-internals#_data_recovery>>.

==== git filter-branch

Командата `git filter-branch` се използва за прилагане на специфични действия
към набор от къмити, например премахване на файл от всеки от тях или
филтриране на цялото хранилище назад до определена поддиректория за
извличане на проект.

В <<ch07-git-tools#_removing_file_every_commit>> обясняваме командата и
изследваме няколко нейни опции като `--commit-filter`, `--subdirectory-filter` и
`--tree-filter`.

В <<ch09-git-and-other-systems#_git_p4>> я използваме за коригиране на
импортирани външни хранилища.


=== Plumbing команди

В книгата използвахме и доста на брой команди от по-ниско ниво, plumbing
командите.

Първата такава беше `ls-remote` в <<ch06-github#_pr_refs>>, за разглеждане на raw
референциите на сървъра.

Използвахме `ls-files` в <<ch07-git-tools#_manual_remerge>>, <<ch07-git-
tools#ref_rerere>> и <<ch07-git-tools#_the_index>> за да видим как изглежда
индексната област в по-суров вид.

Споменахме `rev-parse` в <<ch07-git-tools#_branch_references>>. С нея видяхме как
да превърнем всеки стринг в SHA-1 хеш на Git обект.

Повечето low level plumbing команди са в <<ch10-git-internals#ch10-git-internals>>,
цялата глава е малко или много посветена на тях.
В останалите части на книгата умишлено се старахме да не ги използваме.

[#index]
[index]
= Index
```