

THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

*EVERYTHING YOU NEED TO
KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

Pro Git

Скот Чакон, Бен Штрауб

Верзија 2.1.2-4-ge5142e9, 2022-01-31

Садржај

Лиценца	1
Предговор Скота Чакона	2
Предговор Бена Штрауба	4
Посвете	5
Сарадници	6
Увод	7
Почетак	9
О контроли верзије	9
Кратка историја програма Гит	13
Шта је Гит?	13
Командна линија	17
Инсталирање програма Гит	17
Подешавања за први пут	21
Тражење помоћи	24
Резиме	25
Основе програма Гит	26
Прављење Гит репозиторијума	26
Снимање промена над репозиторијумом	28
Преглед историје комитова	40
Опозив	47
Рад са удаљеним репозиторијумима	52
Означавање	57
Гит алијаси	63
Резиме	65
Гранање у програму Гит	66
Укратко о гранању	66
Основе гранања и спајања	73
Управљање гранама	82
Процеси рада са гранањем	86
Удаљене гране	89
Ребазирање	99
Резиме	109
Гит на серверу	110
Протоколи	110
Постављање програма Гит на сервер	115
Генерисање јавног SSH кључа	118
Подешавање сервера	119
Гит демон	122

Паметан HTTP	124
GitWeb	125
GitLab	127
Опције за хостовање које нуде трећа лица	132
Резиме	132
Дистрибуирани Гит	133
Дистрибуирани процеси рада	133
Како се даје допринос пројекту	136
Одржавање пројекта	160
Резиме	175
GitHub	176
Отварање налога и подешавања	176
Како се даје допринос пројекту	181
Одржавање пројекта	201
Управљање организацијом	216
Писање скрипти за GitHub	219
Резиме	230
Гит алати	231
Избор ревизија	231
Интерактивно стејџовање	240
Скривање и чишћење	244
Потписивање вашег рада	251
Претрага	255
Поновно исписивање историје	259
Демистификовани ресет	267
Напредно спајање	287
Rerere	306
Отклањање грешака са програмом Git	313
Подмодули	316
Паковање	339
Замена	343
Складиште акредитива	351
Резиме	356
Прилагођавање програма Гит	357
Конфигурисање програма Гит	357
Гит атрибути	368
Гит куке	378
Пример полисе коју спроводи програм Гит	381
Резиме	391
Гит и остали системи	392
Гит као клијент	392

Миграирање на Гит	431
Резиме	450
Гит изнутра	451
Водовод и порцелан	451
Гит објекти	452
Гит референце	463
Pack фајлови	467
Рефспек	471
Протоколи за пренос	474
Одржавање и опоравак податак	479
Променљиве окружења	487
Резиме	492
Додатак А: Програм Гит у другим окружењима	494
Графички интерфејси	494
Гит у Visual Studio	499
Гит у Visual Studio Code	500
Гит у IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine	500
Гит у Sublime Text	501
Гит унутар Bash	501
Гит у Zsh	502
Гит у Powershell	505
Резиме	507
Додатак В: Утравђивање програма Гит у ваше апликације	508
Гит из командне линије	508
Libgit2	508
JGit	513
go-git	517
Dulwich	519
Додатак С: Гит команде	521
Подешавање и конфигурација	521
Набављање и креирање пројектата	523
Основно снимање	524
Гранање и спајање	527
Дељење и ажурирање пројектата	529
Инспекција и поређење	531
Отклањање грешака	532
Крпљење	532
Имејл	533
Спољни системи	534
Администрација	535
Водоводне команде	536

Лиценца

Овај рад је лиценциран под Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. Ако желите да погледате копију ове лиценце, посетите <http://creativecommons.org/licenses/by-nc-sa/3.0/> или пошаљите писмо на адресу Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Предговор Скота Чакона

Добродошли у друго издање књиге Про Гит. Прво издање је објављено сада већ пре четири године. Од тада се много тога променило, а ипак многе важне ствари се нису промениле. Док већина основних команда и концепата важи и данас пошто је основни Гит тим заиста добар у одржавању компатибилности са старијим верзијама, појавили су се значајни додаци и измене у заједници која окружује Гит. Друго издање ове књиге би требало да обради те измене и ажурира књигу тако да буде од веће користи новим корисницима.

Када сам написао прво издање, програм Гит је још увек био релативно сложен за употребу и једва прихваћен алат у круговима околних хакера. Почекео је да узима замах у одређеним заједницама, али ни близу није достигао свеприсутност коју има данас. Од тада, прихватила га је скоро свака заједница отвореног кода. Програм Гит је направио невероватан напредак на Виндууз платформи, у експлозији графичких корисничких интерфејса за њега на свим платформама, и подршки за IDE и пословној примени. Про Гит од пре четири године није знао ништа о све му овоме. Један од главних циљева овог новог издања је да се дотакне свих тих нових граница у Гит заједници.

Такође је експлодирала и заједница отвореног кода која користи Гит. Када сам пре скоро пет година сео да напишем књигу (требало ми је мало више времена да објавим прву верзију), тек сам почео да радим у мало познатој компанији која развија Гит хостинг веб сајт под називом *GitHub*. У време објављивања било је можда неколико хиљада људи који користе сајт и само четворо нас који раде на њему. Док пишем овај увод, *GitHub* објављује наш 10 милионити хостован пројекат, са скоро 5 милиона регистрованих налога програмера и преко 230 запослених. Без обзира на то да ли га волите или не, *GitHub* је из корена променио велике делове заједнице отвореног кода на начин који је у време када сам сео да напишем прво издање једва могао и да се замисли.

Написао сам мали одељак у оригиналној верзији Про Гит књиге у вези *GitHub* као примеру хостованог Гит са којим се никада нисам осећао комотно. Није ми се свиђало да пишем о нечemu што сам осећао да је у суштини ресурс заједнице, а да ту говорим и о компанији за коју радим. Мада још увек не волим тај конфликт интереса, важност *GitHub* сервиса у Гит заједници је неизбежна. Уместо примера Гит хостинга, одлучио сам да тај део књиге претворим у нешто што дубље описује шта је *GitHub* и како да се он ефективно користи. Ако ћете учити како да користите програм Гит, онда ће вам познавање употребе сервиса *GitHub* помоћи да постанете део огромне заједнице, што је само по себи вредност без обзира на то који Гит хост одлучите да користите за свој сопствени код.

Још једна велика измена у односу на време последњег издања је развој и успон HTTP протокола за мрежне Гит трансакције. Већина примера у књизи ду измењени са SSH на HTTP јер је он много једноставнији.

Било је невероватно пратити како програм Гит расте током последњих неколико година од релативно непознатог система за контролу верзије у суштински доминантни систем за контролу верзија у комерцијалним применама и отвореном коду. Срећан сам што је Про Гит тако успела и што је могла да буде једна од ретких техничких књига на тржишту која је уједно и успешна и потпуно отворена.

Надам се да ћете уживати у овом ажурираном издању књиге Про Гит.

Предговор Бена Штрауба

Прво издање ове књиге је оно што ме је навукло на програм Гит. То је био увод у стил прављења софтвера који се чинио природнијим од било чега што сам раније видео. Тада сам већ неколико година био програмер, али ово је било право скретање које ме је довело на много интересантнији пут од онога на којем сам био до тада.

Сада, након више година, дајем допринос главној Гит имплементацији, радио сам за највећу Гит хостинг компанију и путовао сам по свету учећи људе о програму Гит. Када ме је Скот упитао да ли бих био заинтересован да радим на другом издању, уопште није било потребе да размислим о томе.

Било је велико задовољство и привилегија радити на овој књизи. Надам се да вам помаже онолико колико је помогла мени.

Посвете

Мојој супрузи, Беки, без које ова авантура никада не би ни почела. — Бен

Ово издање је посвећено мојим девојкама. Мојој супрузи Џесики која ме је подржавала све ове године и мојој ћерки Џозефини, која ће ме подржавати када постанем превише стар да знам шта се догађа. — Ском

Сарадници

Пошто је ово књига отвореног извора, током година смо примили неколико исправки и измена садржаја. Ово су сви људи који су дали допринос енглеској верзији књиге Pro Git као пројекту отвореног кода. Хвала свима који помажу да ово постане боља књига за све.

Akrom K	Ivan Pešić	Mike Thibodeau	Steven Roddis
Aleh Suprunovich	Jan Groenewald	Niels Widger	Thanix
Alexandre Garnier	Jean-Noel Avila	Nikola Zeravcic	Thomas
Ackermann			
Andrei Dascalu	Jean-Noël Avila	Nils Reuß	Tom Schady
Andrew MacFie	Jim Hill	Pablo Schläpfer	Yue Lin Ho
AnneTheAgile	Johannes Schindelin	Pascal Borreli	Yusuke SATO
Anthony Loiseau	Jon Forrest	Pavel Janík	atalakam
Anton Trunov	Jon Freed	Peter Kokot	axmbo
Antonino Ingargiola	Justin Clift	Philippe Miossec	dualsky
Ben Sima	Krzysztof Szumny	Ray Chen	eevan78
Carlos Martín Nieto	Lazar Ljubenović	Rintze M. Zelle	iprok
Changwoo Park	Lazar95	Rob Blanco	mosdalsvsocld
Christoph Prokop	Logan Hasson	Roman Kosenko	paveljanik
Christopher Wilson	Louise Corrigan	Ronald Wampler	petsuter
Cory Donnelly	Luc Morin	Rüdiger Herrmann	rahrah
Damien Tournoud	Marti Bolivar	Sam Ford	rmzelle
Dan Schmidt	Masood Fallahpoor	Sanders Kleinfeld	
sanders@oreilly.com			
Daniele Tricoli	Matthew Miner	Sanja	
schacon@gmail.com			
Danny Lin	Michael MacAskill	Sarah Schneider	sindyoke
Denis Savitskiy	Michael Welch	Sean Head	spacewander
Dmitri Tikhonov	Michiel van der Wulp	Siarhei Krukau	xJom
Haruo Nakayama	Mihajlo Ilijic	Skyper	zwPapEr
Helmut K. C. Tessarek	Mike Charles	Stephan van Maris	

УВОД

Управо ћете провести неколико сати свог живота читајући о програму Гит. Хајде да утрошимо минут времена како би вам показали шта смо вам припремили. Ево кратког сажетка свих десет поглавља и три додатка ове књиге.

У **Поглављу 1** ћемо покрити Системе за Контролу Верзија (VCS) и основе програма Гит — без техничких ствари, једноставно шта је програм Гит, зашто се појавио у свету пуном VCS, шта га издаваја од осталих и зашто га толико људи користи. Затим ћемо објаснити како да преузмете програм програм Гит и како да га подесите за прво коришћење у случају да га већ немате на свом систему.

У **Поглављу 2** ћемо прећи преко основне употребе програма Гит — како да користите програм Гит у 80% случајева на које ћете најчешће наилазити. Када прочитате ово поглавље, требало би да будете у стању да клонирате репозиторијум, сазнате шта се дешавало у историји пројекта, измените фајлове и дате допринос својим променама. Ако се тада књига спонтано запали, требало би да будете у стању да се носите са програмом Гит док не набавите други примерак.

Поглавље 3 говори о моделу гранања у програму Гит, који се често описује као врхунска особина програма Гит. Овде ћете научити шта заиста издаваја програм Гит из крда. Када завршите, можда ћете осетити потребу да проведете мало времена у тишини, размишљајући о томе како сте живели пре него што је Гит гранање постало део вашег живота.

Поглавље 4 ће се посветити програму Гит на серверу. Ово поглавље је за оне који желе да у својој организацији поставе Гит, или на свој лични сервер како би омогућили сарадњу. Такође ћемо истражити разне могућности за хостовање ако вам више одговара да неко други то ради уместо вас.

Поглавље 5 ће се детаљно бавити разним дистрибуираним процесима рада и начинима на које се они постижу у програму Гит. Када пређете ово поглавље, требало да можете стручно да радите са више удаљених репозиторијума, користите програм Гит преко имејла и вешто жонглирате бројним удаљеним гранама и приложеним закрпама.

Поглавље 6 детаљно обрађује *GitHub* хостинг сервис и алате. Бавимо се пријављивањем и управљањем налогом, креирањем и коришћењем Гит репозиторијума, уобичајеним процесима рада за давање доприноса пројектима и за прихватавање доприноса вашем пројекту, програмским интерфејсом сервиса *GitHub* и многим малим саветима који вам у општем случају олакшавају живот.

Поглавље 7 говори о напредним Гит командама. Овде ћете научити о темама као што је овладавање страшном 'reset' командом, употребом бинарне претраге за откривање багова, уређивање историје, детаље око избора ревизије и још много тога. Ово поглавље заокружује ваше познавање програма Гит тако да постајете прави мастер.

Поглавље 8 се бави подешавањем вашег личног Гит окружења. То укључује постављање скрипти кука за спровођење или охрабривање прилагођених полиса и употребу конфигурационих подешавања тако да можете радити на начин који вама одговара.

Обрадићемо и изградњу сопственог скупа скрипти које спроводе прилагођену полису за комитовање.

Поглавље 9 говори о програму Гит и осталим VCS системима. Ово укључује употребу програма Гит у *Subversion* (SVN) свету и конвертовање пројеката из осталих VCS система у Гит. Доста организација још увек употребљава SVN и немају намеру да то промене, али до овог тренутка сте већ научили невероватну снагу програма Гит — и ово поглавље вам показује шта да радите ако још увек морате да користите SVN сервер. Такође ћемо покрити начин за увоз пројекта из неколико различитих система у случају када не можете свакога убедити да направи скок.

Поглавље 10 урања у мутне а ипак прелепе дубине унутрашњости програма Гит. Сада када знате све о програму Гит и знате њиме да рукујете снажно и грациозно, можете наставити са дискусијом о начину на који програм Гит чува своје објекте, о томе какав је модел објекта, детаљима *pack* фајлова, серверским протоколима, и другим стварима. Кроз књигу ћемо упућивати на одељке овог поглавља онда када вам буде дошло да уроните дубље на том месту; али ако сте као ми и желите да уђете у техничке детаље, можда би требало прво да прочитате поглавље 10. То остављамо вама да одлучите.

У **Додатку А** ћемо представити већи број примера употребе програма Гит у различитим окружењима. Обрађујемо више различитих ГКИ и IDE окружења за програмирање у којима можете пожелети да користите програм Гит, као шта вам је доступно. Ако вас интересује преглед употребе програма Гит у вашој љусци, у програмима *Visual Studio* или *Eclipse*, погледајте овде.

У **Додатку Б** ћемо истражити скриптованање и проширивање програма Гит кроз алате као што су *libgit2* и *JGit*. Ако вас интересује писање сложених и брзих прилагођених алата и потребан вам је приступ програму Гит на ниском нивоу, овде можете погледати како изгледа предео.

Конечно, и **Додатку В** ћемо проћи кроз све главне Гит команде, једну по једну и указати на место у књизи на којем смо их објаснили, као и шта смо урадили помоћу њих. Ако желите да знате место у књизи на којем смо користили било коју одређену Гит команду, потражите га овде.

Хајде да почнемо.

Почетак

Ово поглавље ће се бавити неким основним стварима о програму Гит. Почекнемо уз кратка објашњења о томе како раде алати за контролу верзије, а онда ћемо прећи на то како подесити програм Гит да ради на систему који користите и на крају ћемо све то уклопити тако да можете почети са радом. На крају поглавља требало би да разумете зашто постоји програм Гит, зашто би требало да га користите и бићете спремни да све то спроведете у дело.

О контроли верзије

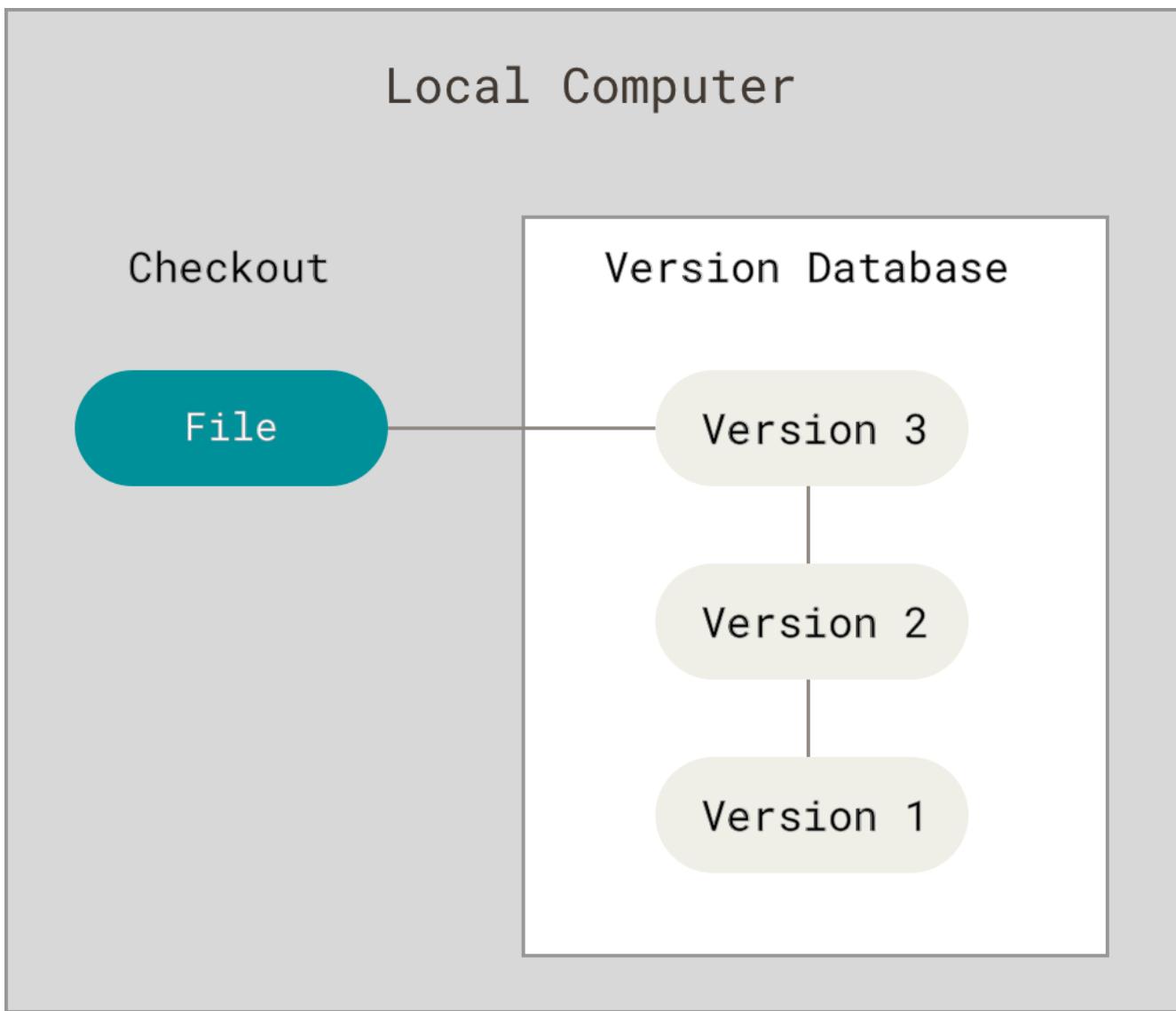
Шта је то „контрола верзије”, и зашто би вас то занимало? Контрола верзије је систем који памти промене фајла или скупа фајлова настале током времена и који вам омогућава да се касније вратите на одређене верзије. Као примере у овој књизи користићете изворни код софтвера као фајлове над којима се примењује контрола верзије, али у стварности овакав приступ би радио са скоро сваком врстом фајлова на рачунару.

Ако сте графички или веб дизајнер и желите да сачувате сваку верзију слике или макете (што је добра пракса), врло је мудра идеја користити систем за контролу верзије (*Version Control System*, VCS). Дозвољава вам да вратите фајлове на пређашње стање, да вратите читав пројекат на пређашње стање, да поредите измене током времена, да видите ко је последњи изменио нешто што би могло да буде узрок проблема који је настао, ко је објавио да постоји неки проблем и када, и још много тога. У општем случају, употреба VCS система подразумева и то да ако нешто забрљате или изгубите фајлове, лако можете да их повратите. Штавише, све ово добијате уз веома мало муке.

Локални системи за контролу верзије

Метода за контролу верзија коју већина људи бира јесте копирање фајлова у други директоријум (они мудрији би могли да га рецимо обележе датумом). Овај приступ је веома чест јер је доста једноставан, али је истовремено јако подложен грешкама. Лако је заборавити у ком директоријуму се налазите, па да случајно упишете нешто у погрешан фајл или копирате преко фајлова које сте намеравали да сачувате.

Да би се изборили са овим проблемом, програмери су одавно развили локалне VCS системе који су имали једноставну базу података у којој су се чувале све промене одређених фајлова.

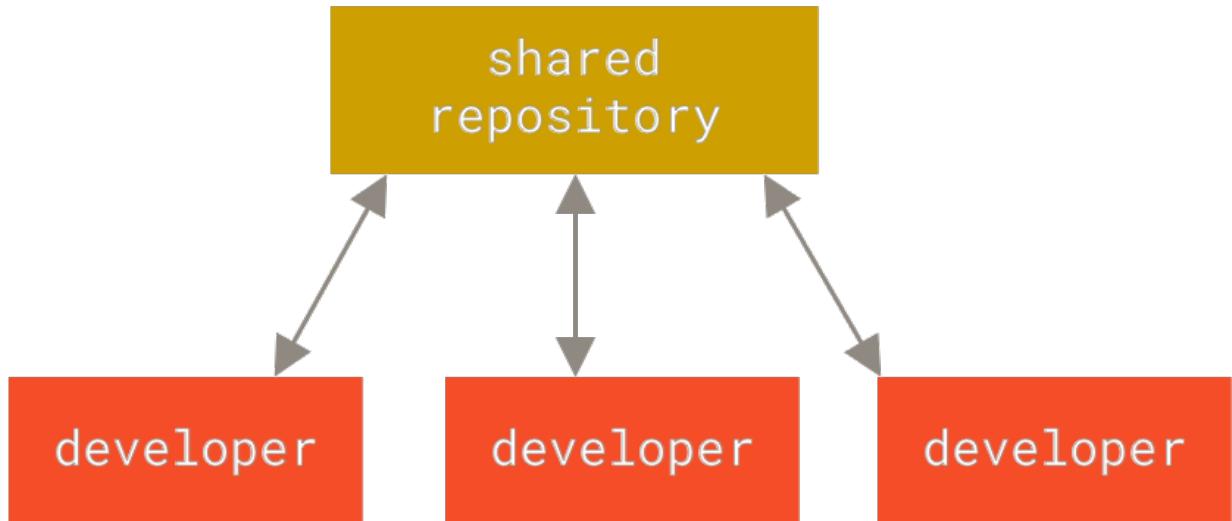


Слика 1. Локална контрола верзије

Један од популарнијих алата за VCS био је систем под именом *RCS*, који се и данас испоручује уз многе рачунаре. *RCS* ради тако што у посебном формату на диску чува скуп закрпи (*patch set*, односно разлике између фајлова); онда се проласком кроз све закрпе може поново добити изглед фајла у било ком временском тренутку.

Централизовани системи за контролу верзије

Следећи велики проблем на који људи наилазе је потреба за сарадњом са програмерима на другим системима. Да би се изборили са овим проблемом, развијени су централизовани системи за контролу верзије (CVCS). Ови системи, као што су *CVS*, *Subversion* и *Perforce* имају један сервер који садржи све верзионисане фајлове, и одређени број клијената који преузимају фајлове са тог централног места. Током много година, ово је био стандардни начин за реализацију контроле верзије.



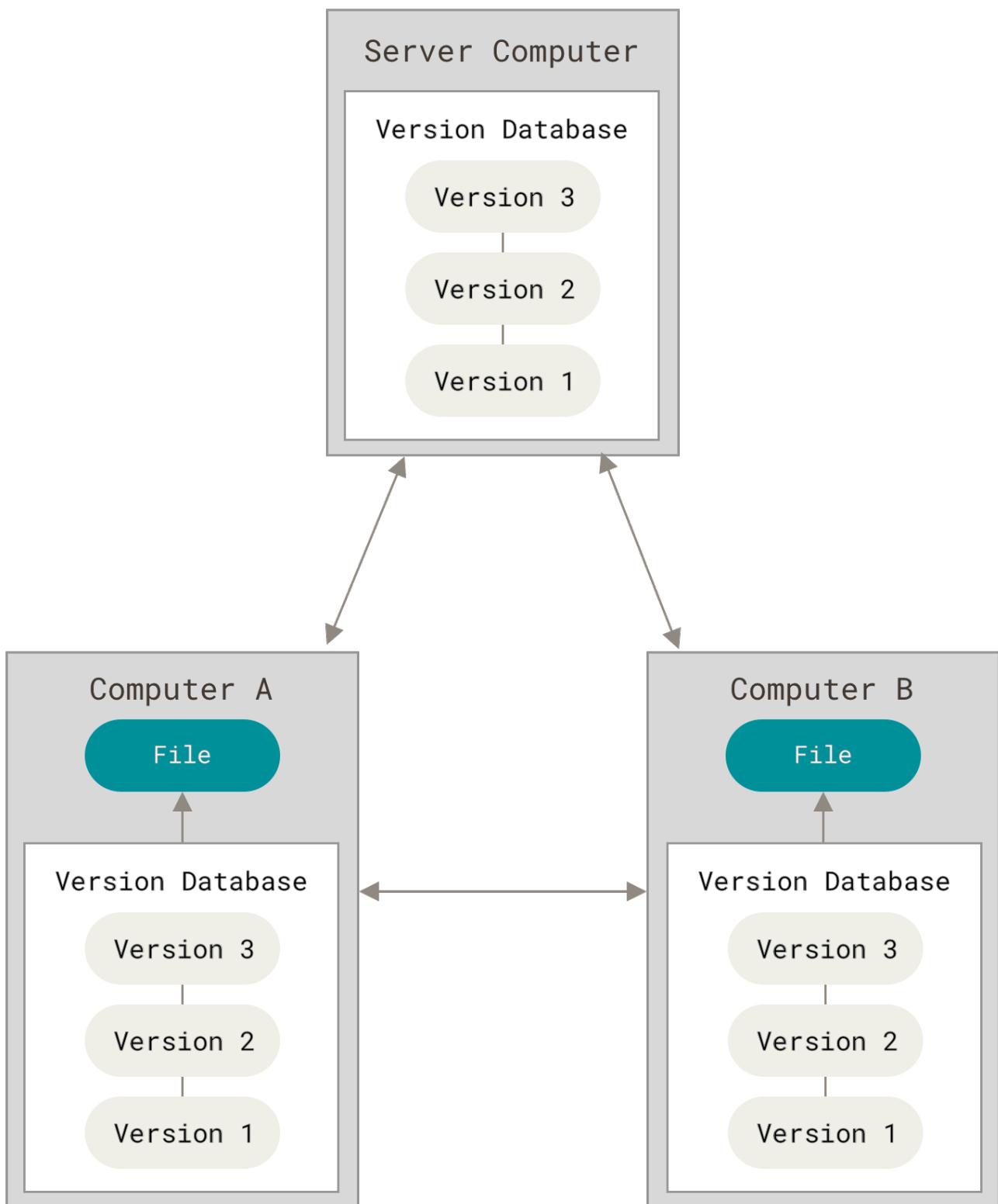
Слика 2. Централизована контрола верзије

Оваква поставка нуди многе предности, поготово над локалним VCS системима. На пример, сви до неке границе знају шта остали раде на пројекту. Администратори имају детаљну контролу над тиме ко шта може да уради и много је лакше да се администрира CVCS систем него борити се са локалним базама података сваког клијента.

Међутим, оваква поставка има и неке озбиљне недостатке. Најочигледнији је јединствена тачка квара коју представља овако централизовани сервер. Ако сервер буде у квиру на сат времена, ниједна особа не може да ради на пројекту, нити да чува верзионисане промене онога што тренутно ради. Ако се хард диск на коме се налази централизована база података оштети, губи се апсолутно све — читава историја пројекта осим оних тренутних верзија које људи имају на локалним машинама. Локални VCS системи имају исти овај проблем — кад год се читава историја пројекта налази на једном месту, постоји ризик да се изгуби све.

Дистрибуирани системи за контролу верзије

Овде долазе на ред дистрибуирани системи за контролу верзије (DVCS). Код DVCS система (као што су *Git*, *Mercurial*, *Bazaar* или *Darcs*), клијенти не преузимају само тренутан изглед фајлова, већ потпуно пресликају цео репозиторијум. Тако, ако неки од сервера престане са радом, а ови системи су се повезали помоћу њега, сваки од клијентових репозиторијума може да се ископира назад на сервер да би се он обновио. Сваки клон је буквально потпуна резервна копија свих података.



Слика 3. Дистрибуирана контрола верзија

Штавише, многи од ових система се прилично добро носе са више репозиторијума на даљину са којима могу да раде, тако да можете сарађивати са различитим групама људи на различите начине истовремено у истом пројекту. Ово вам омогућава да подесите неколико типова токова рада који нису могући у централизованом систему, као што су хијерархијски модели.

Кратка историја програма Гит

Како и многи добре ствари у животу, Гит је почeo са мало креативног уништења и плаховите полемике.

Линукс језгро је прилично широк софтверски пројекат отвореног кода. Током већине времена одржавања Линуксовог језгра (1991-2002), промене у софтверу слате су унаоколо као закрпе и архивирани фајлови. Године 2002, пројекат Линукс језгра почeo је да користи власнички DVCS који се звао *BitKeeper*.

Године 2005, однос између заједнице која је радила на Линукс језгру и комерцијалне компаније која је развијала *BitKeeper* се распао, и бесплатан статус алата био је укинут. Ово је приморало Линуксову заједницу програмера (и посебно Линуса Торвалдса, оснивача Линукса) да осмисле свој сопствени алат ослањајући се на неке лекције које су научили док су користили *BitKeeper*. Неки од циљева које је имао нови систем били су следећи:

- брзина,
- једноставан дизајн,
- снажна подршка за нелинеарни развој (на хиљаде паралелних грана),
- потпуно дистрибуиран концепт,
- могућност да се ефикасно рукује великим пројектима као што је Линукс језгро (брзина и величина података).

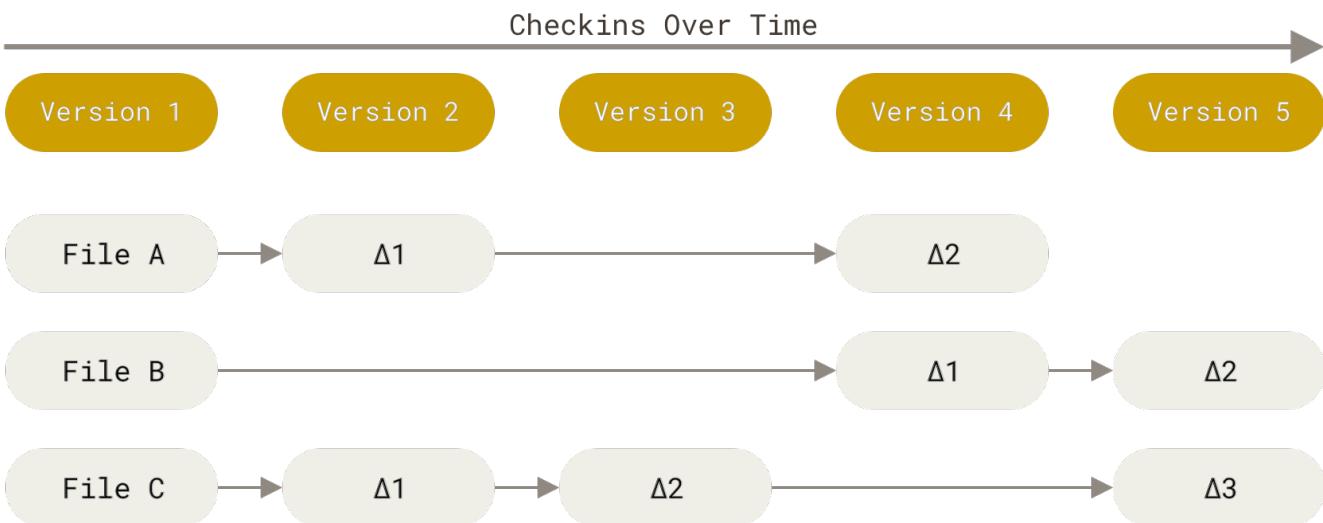
Од његовог настанка 2005, Гит је еволуирао и сазрео у алат који је био једноставан за употребу, али је задржао ове квалитете. Невероватно је брз, веома је ефикасан са великим пројектима, и има одличан систем гранања за нелинеарно развијање програма (погледајте [Гранање у програму Гит](#)).

Шта је Гит?

Па, шта је у суштини Гит? Веома је важно схватити овај одељак, јер ако разумете шта је Гит и основе његовог функционисања, онда ће вам бити много једноставније да ефикасно користите програм Гит. Док учите програм Гит, пробајте да заборавите на ствари које знате о многим другим VCS системима као што су *Subversion* и *Perforce*; на тај начин ћете избећи неке суптилне недоумице док користите овај алат. Програм Гит чува и посматра податке на много другачији начин од осталих система, иако је његов кориснички интерфејс веома сличан њиховом, а разумевање тих разлика ће помоћи да не дође до забуне током коришћења.

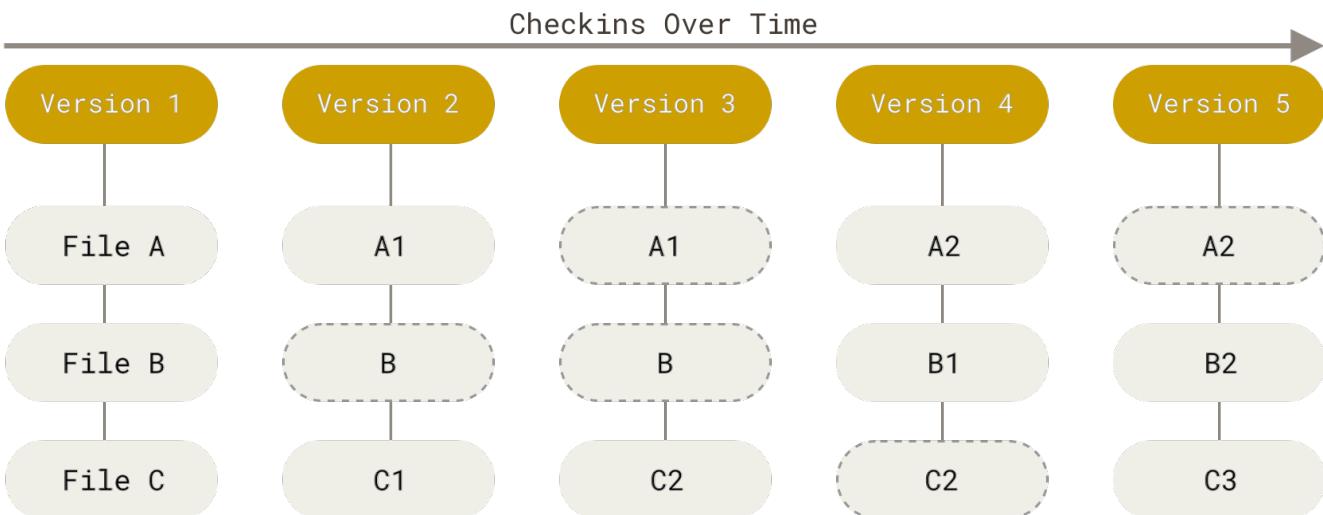
Снимци, не разлике

Главна разлика између програма Гит и других VCS система (укључујући *Subversion* и екипу) је начин на који Гит посматра податке. Концептуално, већина других система чува податке као листу промена фајлова. Ови системи (*CVS*, *Subversion*, *Perforce*, *Bazaar* и тако даље) посматрају податке које чувају као скуп фајлова и промена које су направљене над њима током времена (ово се обично описујекао контрола верзије базирана на *дeлтама*).



Слика 4. Чување података као промене у односу на основну верзију сваког фајла

Програм Гит не посматра податке на овај начин, нити их тако памти. Уместо тога, програм Гит посматра податке као да су скуп снимака (*snapshots*) минијатурног фајл система. Сваки пут када извршите комит (*commit*), или сачувате стање свог пројекта у Гит репозиторијум, он у суштини узима слику стања ваших фајлова у том тренутку и памти референцу на тај снимак. Да би одржао ефикасност, ако се фајл није променио, програм Гит не чува фајл поново, већ само везу ка претходном идентичном фајлу који је већ раније сачувао. Програм Гит податке посматра као **ток снимака**.



Слика 5. Чување података као снимака пројекта током времена

Ово је важна разлика између програма Гит и скоро свих других VCS система. Због овога Гит скоро сваки аспект контроле верзије који су већина осталих система само копирали из претходне генерације на другачији начин. То чини Гит минијатурним фајл системом са утврђеним изузетно моћним алатима, а не само обичним VCS системом. Истражићемо неке предности које добијате чистим посматрањем података на овај начин док будемо говорили о гранању у програму Гит у [Гранање у програму Гит](#).

Скоро свака операција је локална

Већини операција у програму Гит су потребни само локалне фајлове и ресурси да би се извршиле - у општем случају, није потребна никаква информација са другог рачунара на мрежи. Ако сте навикнути на CVCS системе где већина операција има застој због латенције

мреже, овај аспект Гита ће вас уверити у то да су богови брезине благосиљали Гит ванземаљским моћима. Пошто имате читаву историју пројекта одмах ту на локалном диску, већина операција ће се извршити скоро тренутно.

На пример, да бисте прегледали историју пројекта, Гит не мора да оде на сервер да је преузме да би вам је приказао—само треба да је прочита директно из локалне базе података. Ово значи да ћете видети историју пројекта истог тренутка. Ако желите да видите измене које су унете у фајл између тренутне верзије и оне од пре месец дана, Гит може да погледа како је фајл изгледао пре месец дана и да уради локално израчунавање разлике, уместо да пита удаљени сервер да уради то или да повуче старију верзију фајла са удаљеног сервера и да израчуна разлику локално.

Ово такође значи да нема много ствари које не можете да урадите док нисте прикачени на мрежу или на VPN. Ако се укрцате на авион или воз и желите да урадите нешто, можете да комитујете без проблема (у своју локалну копију, сећате се?) све док не дођете до мрежне конекције да гурнете податке на сервер. Ако одете кући и не можете да подесите VPN клијент да ради како треба, још увек можете да радите на пројекту. Код многих других система, овакве ствари су немогуће или захтевају превише муке. На пример, у програму *Perforce* не можете да урадите много тога ако нисте повезани са сервером; а у програмима *Subversion* и *CVS* можете да мењате фајлове, али не можете да комитујете промене у базу података (јер она није на мрежи). Ово можда не изгледа као велика ствар, али изненадили бисте се када видите колико овакве ствари значе.

Гит има интегритет

У програму Гит се за све рачуна контролна сума (*checksum*) пре него што се сачува, а онда се стварима приступа користећи ту контролну суму. То значи да је немогуће променити садржај било ког фајла или директоријума а да Гит не зна за то. Ова функционалност је уградјена у Гит у најнижим слојевима и својствена је његовој филозофији. Не можете да изгубите податке током транзита, или да дође до оштећења фајлова а да Гит то не примети.

Механизам који програм Гит користи за контролну суму се зове SHA-1 хеш (*SHA-1 hash*). То је стринг од 40 карактера који се састоји од хексадецималних цифара (0-9 и a-f) и рачуна се на основу садржаја фајла или структуре директоријума у програму Гит. SHA-1 хеш изгледа отприлике овако:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Ове хеш вредности ћете сретати посвуда у програму Гит јер их он доста користи. Заправо, све што програм Гит чува у базу података чува се на основу хеш вредности садржаја, а не на основу имена фајла.

Гит углавном само додаје податке

Када обављате акције у програму Гит, скоро све само додају податке у Гит базу података. Тешко је натерати систем да уради нешто што не може да се опозове, или да обрише податке на било који начин. Као и код било ког другог VCS система, можете да направите грешку код промена које још нису комитоване; али након што комитујете снимак у Гит,

веома је тешко изгубити га, поготово ако редовно шаљете базу података другом репозиторијуму.

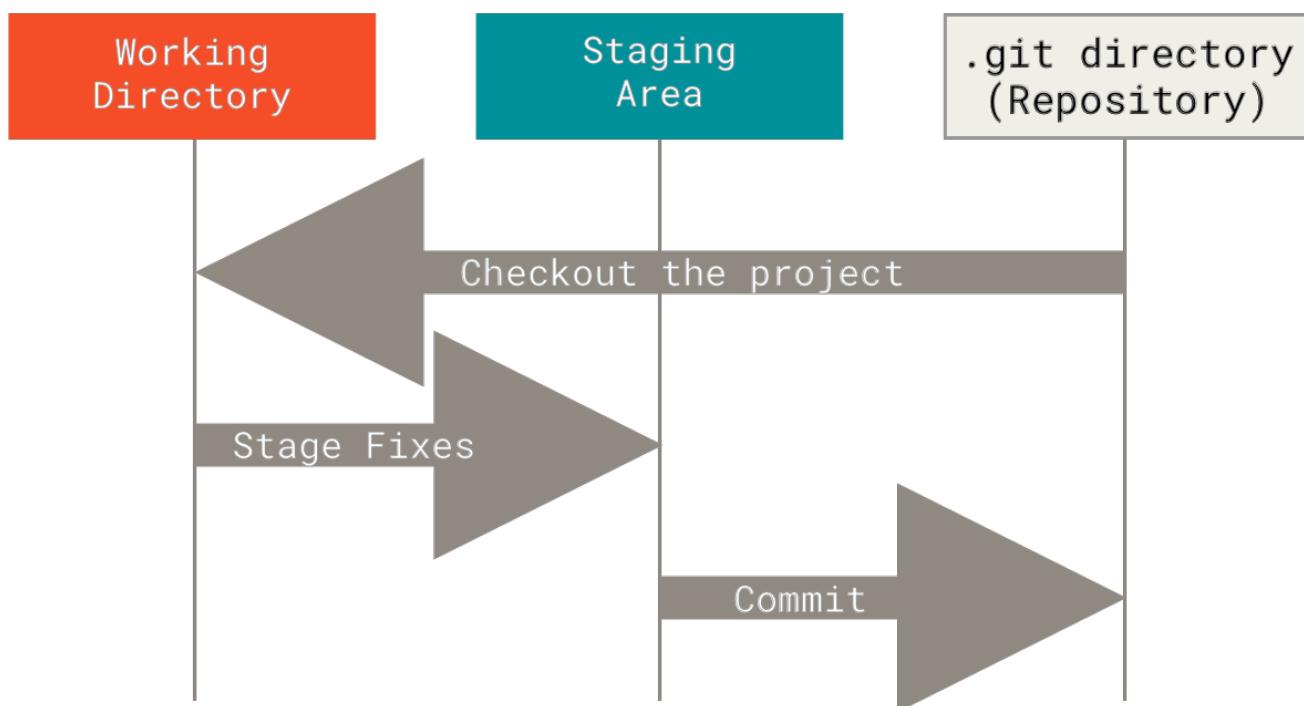
Због овога је коришћење програма Гит право уживање, јер зnamо да можемо да експериментишемо без бриге да ћemo нешто упропастити. За мало детаљнији поглед на то како Гит чува податке и како вратити податке које сте наизглед изгубили, погледајте [Опозив](#).

Три стања

Сада обратите пажњу — ово је главна ствар коју треба запамтити о програму Гит ако желите да остатак процеса учења тече како ваља. Гит има три главна стања у којима се могу наћи ваши фајлови: *измењено*, *стејцовано* и *комитовано*:

- Измењено значи да сте променили фајл али да га још увек нисте комитовали у базу података.
- Стејцовано значи да сте изменjeni фајл означили да се у свом тренутном стању укључи у следећи комит снимка који будете направили.
- Комитовано значи да су подаци смештени у локалну базу података на сигурном.

Ово нас води до три главне секције Гит пројекта: Гит директоријум, радни директоријум и стејџ.



Слика 6. Радни директоријум, стејџ и Гит директоријум

Радни директоријум је једно одјављивање (*checkout*) неке верзије пројекта. Ови фајлови се довлаче из компресоване базе података из Гит директоријума и смештају се на диск да бисте их користили и мењали.

Стејџ је фајл који се обично налази у Гит директоријуму и чува информације о томе шта ће се укључити у следећи комит. Његово техничко име у Гит терминологији је „индекс”, али се

често користи и назив „стејџ”.

Гит директоријум је место где програм Гит чува све метаподатке и базу података објекта вашег пројекта. Ово је најважнији део програма Гит и то је оно што се копира када клонирате репозиторијум са другог рачунара.

Основни ток рада са програмом Гит иде отприлике овако:

1. Вршите измене фајлова у радном директоријуму.
2. Селективно стејџујете само оне измене које желите да буду део наредног комита, што додаје *само* те измене на стејџ.
3. Комитујете, што значи да узмете фајлове у оној верзији у којој су били када сте их додали на стејџ и тај снимак трајно смештате у Гит директоријум.

Ако се нека верзија фајла налази у Гит директоријуму, сматра се *комитованом*. Ако је изменјена и онда дodata на стејџ, она је *стејџована*. А ако се променила од одјављивања, али није стејџована, онда је *измењена*. У [Основе програма Гит](#), научићете више о овим стањима и о томе како се најбоље користе, као и начин да потпуно прескочите стејџ.

Командна линија

Постоји много начина на које можете да користите програм Гит. Ту је оригинална командна линија и многи графички кориснички интерфејси (ГКИ) са разноразним способностима. У овој књизи ћемо користити програм Гит из командне линије. За почетак, командна линија је једино место где можете да покренете *све* Гит команде - већина ГКИ ради једноставности имплементира само неки подскуп функционалности Гита. Ако умете да урадите нешто из командне линије, вероватно ћете се снаћи и са ГКИ верзијом, док обрнуто не мора да важи. Такође, избор графичког клијента је питање личног укуса, а *сви* корисници већ имају инсталiranу командну линију и могу да јој приступе.

То значи да се од Вас очекује да покренете *Terminal* на мекОСу или *Command Prompt* или *Powershell* на Виндузу. Ако не знаете о чему причамо, добра идеја је да овде станете и да на брзину то истражите, како бисте без проблема могли да наставите са праћењем примера и описа у овој књизи.

Инсталирање програма Гит

Пре него што почнете да користите програм Гит, он мора бити доступан на рачунару. Ако је већ инсталiran, вероватно је добра идеја да га ажурирате на последњу верзију. Можете да га инсталирате као пакет или преко неког другог инсталера, или да преузмете изворни код па да га сами компајлирате.



Ова књига је писана користећи програм Гит верзије **2.8.0**. Премда већина команди треба да ради чак и на прастарим верзијама програма Гит, неке од њих можда неће, или ће можда имати мало другачије понашање ако користите старију верзију. Пошто је програм Гит одличан у очувању компатибилности са старијим верзијама, свака верзија после 2.8 ће радити како ваља.

Инсталирање на Линуксу

Ако на Линуксу желите да инсталирате основне Гит алате помоћу бинарног инсталера, у општем случају то можете да урадите помоћу алата за управљање пакетима који долази уз вашу дистрибуцију. Ако сте на Федори (или било којој дистрибуцији која је у близку вези са њом и заснована је на RPM, као што је RHEL или CentOS), можете да употребите *dnf*:

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

Ако сте на дистрибуцији која је базирана на Дебијану као што је Убунту, пробајте *apt*:

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

За више опција, на Гит веб сајту (<http://git-scm.com/download/linux>) се налазе инструкције за инсталирање на неколико различитих Јуникс дистрибуција.

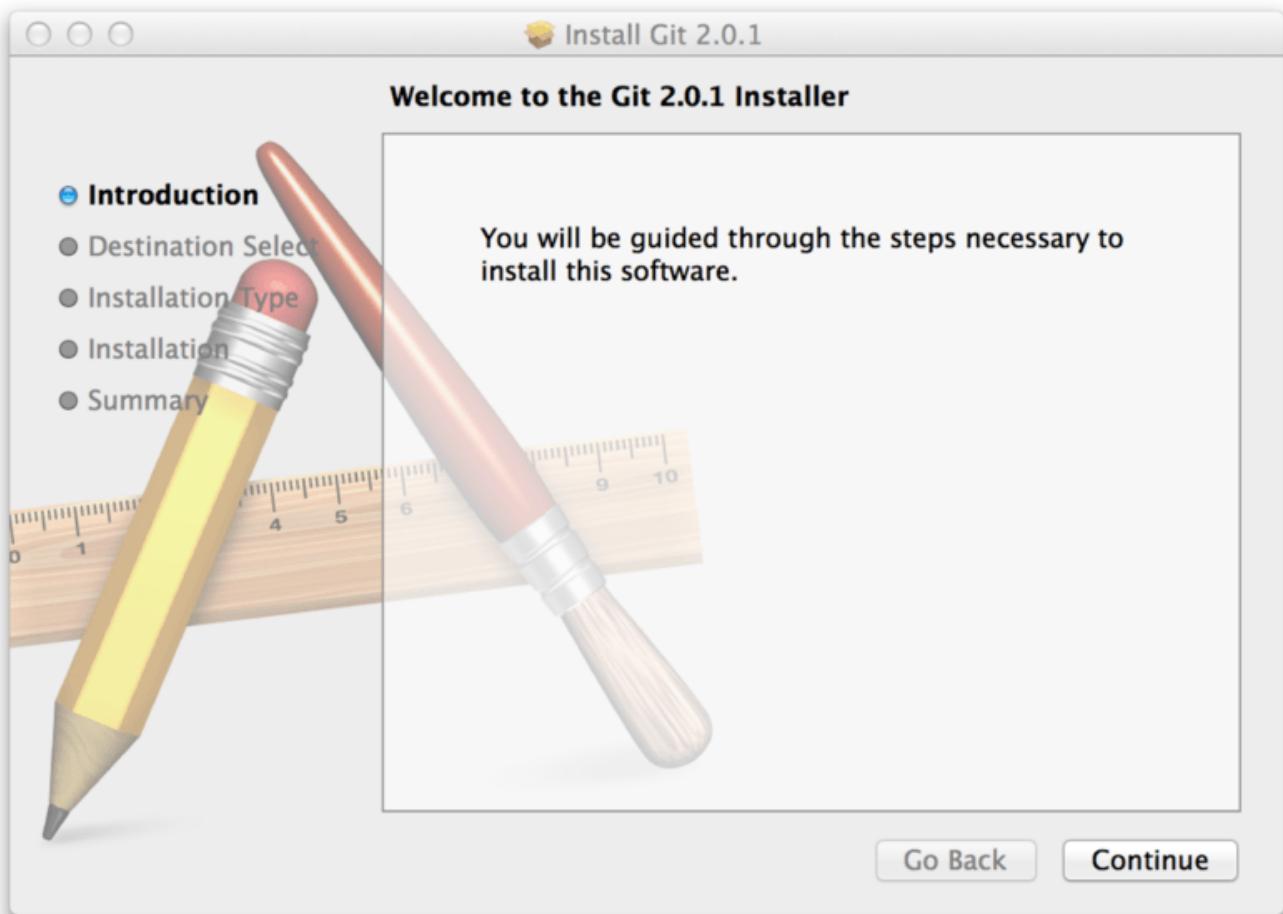
Инсталирање на мекОС

Постоји неколико начина да се програм Гит инсталира на Меку. Најлакше је вероватно инсталирати *Xcode Command Line Tools*. На *Mavericks* (10.9 или новијем) ово можете да урадите тако што ћете једноставно покренути *git* из Терминала први пут.

```
$ git --version
```

Ако га већ немате инсталiran, питаће вас да ли желите да га инсталирате.

Ако желите новију верзију, можете да је инсталирате и преко бинарног инсталера. мекОС Гит инсталер се одржава и доступан је за преузимање на Гит веб сајту, са адресе <https://git-scm.com/download/mac>.



Слика 7. Git мекОС инсталер

Инсталирање на Виндоузу

Такође постоји неколико начина на које можете инсталирати програм Гит на Виндоузу. Званична изградња је доступна за преузимање на Гит веб сајту. Једноставно идите на <https://git-scm.com/download/win> и преузимање ће аутоматски почети. Имајте на уму да се овај пројекат зове *Git for Windows* и независан је од самог Гита; за више информација о њему, идите на <https://gitforwindows.org/>.

Ако желите аутоматизовану инсталацију можете употребити [Гит Chocolatey пакет](#). Имајте на уму да Chocolatey пакет одржава заједница корисника.

Инсталирање из извornог кода

Неки људи ће пожелети да инсталирају програм Гит из извornог кода, јер тако добијају најновију верзију. Бинарни инсталери обично помало каскају, мада с обзиром на то да је Гит сазрео током претходних година, разлика је много мања.

Ако желите да инсталирате програм Гит из извора, морате да имате следеће библиотеке од којих Гит зависи: *autotools*, *curl*, *zlib*, *openssl*, *expat* и *libiconv*. На пример, ако сте на систему који има *dnf* (као што је Федора) или *apt-get* (као систем базиран на Дебијану), можете да употребите неку од следећих команда да инсталирате минималне зависности које су неопходне за компајлирање и инсталирање бинарних фајлова програма Гит:

```
$ 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 libbz-dev libssl-dev
```

Ако желите документацију у разним форматима (*doc*, *html*, *info*), неопходне су и следеће додатне зависности:

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



Корисници RHEL система изведених из RHEL као што је CentOS и Scientific Linux ће морати да [укључе EPEL репозиторијум](#) како би могли да преузму [docbook2X](#) пакет.

Ако користите дистрибуцију засновану на Дебијану (Дебијан/Убунту/Убунту-изедене), такође вам је потребан и [install-info](#) пакет:

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

Ако користите дистрибуцију засновану на RPM (Федора/RHEL/RHEL-изведене), такође вам је потребан [getopt](#) пакет (који је на дистрибуцијама заснован на Дебијану већ инсталiran):

```
$ sudo dnf install getopt
```

Уз то, ако користите Федору/RHEL/RHEL-изведене, морате да урадите и следеће:

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

услед разлика у имену бинарног фајла.

Када имате све неопходне зависности, можете да преузмете последње објављени *tarball* са неколико места. Можете да га добијете са сајта [kernel.org](#), на адреси <https://www.kernel.org/pub/software/scm/git>, или са мироре на GitHub веб сајту <https://github.com/git/git/releases>. Мало је јасније која је верзија последња на GitHub страници, али [kernel.org](#) такође има *помнисце издања* ако желите да верификујете шта преузимате.

Онда, компајлирајте и инсталирајте:

```
$ 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
```

Када ово обавите, можете да преузмете Гит и преко сâмог Гит репозиторијума за ажурирања:

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

Подешавања за први пут

Сада када је програм Гит инсталiran на вашем систему, треба да урадите неколико ствари којима своје Гит окружење прилагођавате себи. Ове ствари треба урадити само једном на сваком рачунару; задржавају се између ажурирања. Можете их променити било када поновним покретањем команди.

Програм Гит долази са алатом који се зове `git config` и он вам омогућава да приказујете и постављате конфигурационе променљиве које контролишу све аспекте изгледа и рада програма Гит. Ове променљиве се могу налазити на три различита места:

1. Фајл `[путања]/etc/gitconfig`: садржи вредности за сваког корисника на систему и све њихове репозиторијуме. Ако команди `git config` проследите опцију `--system`, врши се читање и упис у овај фајл. Пошто је ово системски конфигурациони фајл, потребне су вам администраторске или привилегије супер корисника ако желите да га измените.
2. Фајл `~/.gitconfig` или `~.config/git/config`: специфично за корисника. Програму Гит можете наложити да пише и чита из овог фајла тако што проследите `--global` опцију и то утиче на све репозиторијуме са којима радите на вашем систему.
3. `config` фајл у Гит директоријуму (односно `.git/config`) који год репозиторијум да тренутно користите: специфично за тај репозиторијум. Можете форсирати да програм Гит чита и пише из овог фајла опцијом `--local`, али је то и онако подразумевано понашање. Није изненађење да се морате налазити негде унутар Гит репозиторијума ако желите да ова опција функционише како треба.

Сваки ниво преклапа вредности из претходног нивоа, на пример вредности у `.git/config` одређују оне у `/etc/gitconfig`.

На Виндоуз системима, програм Гит тражи `.gitconfig` фајл у `$HOME`` директоријуму (за већину људи је то `C:\Users\$USER`). Такође тражи `/etc/config`, мада је то релативно у односу на MSys корен, што је гдегод одлучите да инсталirate Гит на Виндоуз систему када покренете инсталацију. Међутим, ако користите 2.x или каснију верзију програма Гит за Виндоуз, путања је `C:\Documents and Settings\All Users\Application Data\Git\config` на Виндоуз XP и `C:\ProgramData\Git\config` на Виндоуз Висти и новијим верзијама. Овај конфигурациони фајл се може променити само покретањем `git config -f <фајл>` уз администраторске

привилегије.

Сва своја подешавања, као и то одакле долазе, можете видети ако употребите:

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

Ваш идентитет

Прва ствар коју треба да урадите када инсталирате програм Гит је да подесите корисничко име и мејл адресу. Ово је важно јер сваки Гит комит користи ове податке, и непроменљиво се урезује у комитове које креirate:

```
$ git config --global user.name "Petar Petrovic"  
$ git config --global user.email petarpetrovic@primer.com
```

Опет, ово треба да урадите само једном ако проследите **--global** опцију, јер ће онда Гит увек користити те податке за све што ради на том систему. Ако желите да преклопите ово другим именом или мејл адресом за специфичне пројекте, можете да покренете команду без **--global** опције када се налазите у том пројекту.

Многи ГКИ алати ће вам помоћи да урадите ово када их први пут покренете.

Ваш едитор

Сада када је подешен ваш идентитет, можете да конфигуришете подразумевани едитор текста који ће се користити када програм Гит од вас захтева да укуцате поруку. Ако није конфигурисан, програм Гит користи подразумевани едитор вашег система, што је најчешће *Vim*.

Ако желите да користите други едитор текста, као што је *Emacs*, можете урадити следеће:

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

На Виндоуз систему, уколико желите да користите други едитор, морате да наведете комплетну путању до његовог извршног фајла. Начин да то урадите може бити различит и зависи од начина на који је ваш едитор спакован.

У случају програма *Notepad++*, популарног едитора за програмере, највероватније ћете користити 32-битну верзију, јер у време писања 64-битна верзија не подржава све додатке. Ако имате 32-битни Виндоуз систем, или користите 64-битни едитор на 64-битном систему, откуцаћете нешто слично следећем:

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

 *Vim*, *Emacs* и *Notepad++* су популарни едитори текста које програмери често користе на системима заснованим на Јуниксу као што су Линукс и мекОС или Виндоуз систем. Ако користите неки други едитор, или 32-битну верзију, молимо вас да пронађете одговарајуће инструкције за подешавање вашег омиљеног едитора у програму Гит у [git config core.editor команде](#).

 Ако не подесите едитор на овај начин, вероватно ћете се веома збунити када га програм Гит покрене. Један од примера на Виндоуз систему може бити прерано прекинута Гит операција током уређивања које је започео програм Гит.

Име ваше подразумеване гране

Када са `git init` креирате нови репозиторијум, програм Гит ће подразумевано да креира грану под именом *master*. Почеквши од верзије 2.28 програма Гит па надаље, имате могућност да поставите неко друго име за почетну грану.

Ако желите да име подразумеване гране буде *main*, извршите следеће:

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

Приказ подешавања

Ако желите да погледате текућа подешавања, можете да користите `git config --list` команду да излистате сва подешавања која програм Гит може да нађе у том тренутку:

```
$ git config --list
user.name=Petar Petrovic
user.email=petarpetrovic@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
...
```

Неке кључеве можете видети и више од једног пута, јер програм Гит чита исти кључ из различитих фајлова (на пример, [\[путања\]/etc/gitconfig](#) и [~/.gitconfig](#)). У овом случају, програм Гит користи последњу вредност за сваки јединствен кључ који види.

Можете да проверите и шта програм Гит мисли да је вредност тачно одређеног кључа помоћу `git config <кључ>` на следећи начин:

```
$ git config user.name
Petar Petrovic
```

Пошто би програм Гит исту конфигурациону променљиву могао да прочита из више од једног фајла, може се догодити да имате неочекивану вредност неког од ових кључева и да не знате разлог за то. У случајевима као што је овај, можете питати програм Гит за *порекло* те вредности, па ће вам он одговорити који конфигурациони фајл је дао последњу реч код постављања те вредности:

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

Тражење помоћи

Ако вам икад буде била потребна помоћ док користите програм Гит, постоје три еквивалентна начина да дођете до свеобухватне странице са упутством (*manual page* или скраћено *manpage*) за било коју од команда програма Гит:

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

На пример, можете да добијете *manpage* за `git config` команду покретањем:

```
$ git help config
```

Ове команде су добре јер им можете приступити одакле год пожелите, чак и ако сте ван мреже. Ако странице помоћи које тако добијете и ова књига нису довољне и потребна вам је помоћ неке особе, можете да пробате `#git`, `#github`, или `#githhub` канале на *Libera Chat IRC* серверу који се налази на адреси <https://libera.chat/>. На овим каналима се обично налази на стотине људи који знају пуно тога о програму Гит и често су вољни да помогну.

Уз то, ако вам није потребна детаљна *manpage* помоћ, већ кратко подсећање о доступним опцијама неке Git команде, можете опцијом `-h` да потражите сажети излаз команде „*help*”, као на пример:

```

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

-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
--pathspec-from-file <file> read pathspec from file
--pathspec-file-nul       with --pathspec-from-file, pathspec elements are
separated with NUL character

```

Резиме

Сада треба да имате основно разумевање о томе шта је програм Гит и како се разликује од централизованих система за контролу верзије које сте можда раније користили. Такође сада на свом систему треба да имате радну верзију програма Гит која је подешена Вашим личним идентитетом. Време је да научите неке основе програма Гит.

Основе програма Гит

Ако постоји једно поглавље које је довољно прочитати да бисте могли да се снађете у програму Гит, онда је то ово. Ово поглавље покрива сваку основну команду која ће вам бити потребна да бисте обавили велику већину ствари које ћете најчешће радити док будете користили Гит. Кад завршите с овим поглављем, моћи ћете да конфигуришете и иницијализујете репозиторијум, да почнете и престанете да пратите фајлове, и да стејџујете и комитујете промене. Показаћемо вам и како да брзо подесите програм Гит тако да игнорише неке фајлове и типове фајлова, како да поништите грешке брзо и лако, како да претражите историју пројекта и погледате промене између комитова, и како да гурнете и повучете (*push* и *pull*) фајлове са удаљених репозиторијума.

Прављење Гит репозиторијума

Гит пројекат можете да преузмете на један од два основна начина:

1. Можете да узмете неки постојећи локални директоријум који тренутно није под контролом верзије, па да га претворите у Гит репозиторијум, или
2. Можете да *клонирате* постојећи Гит репозиторијум са неког другог места.

У сваком случају, добићете Гит репозиторијум на локалној машини који је спреман за рад.

Иницијализација репозиторијума у постојећем директоријуму

Ако имате директоријум пројекта који тренутно није под контролом верзије и желите почети да га контролишујете помоћу програма Гит, најпре морате да одете до директоријума пројекта. Ако ово никада раније нисте радили, процедура се разликује у односу на систем који користите:

за Линукс:

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

за мекОС:

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

за Виндоуз:

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

и откуцајте:

```
$ git init
```

Ово прави нови поддиректоријум под именом `.git` који садржи све неопходне фајлове - скелет Гит репозиторијума. У овом тренутку се још увек ништа не прати у вашем пројекту. Погледајте [Гит изнутра](#) за више информација о томе шта су тачно фајлови који се налазе у `.git` директоријуму који сте управо направили.

Ако желите да почнете са контролом верзије постојећих фајлова (за разлику од празног директоријума), вероватно би требало да почнете праћење тих фајлова и да учините иницијални комит. То можете постићи са неколико `git add` команда које наводе фајлове које желите да пратите, а затим укуцајте команду `git commit`:

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

Ускоро ћемо објаснити шта раде ове команде. Сада имате Гит репозиторијум са праћеним фајловима и иницијалним комитом.

Клонирање постојећег репозиторијума

Ако желите да преузмете копију постојећег Гит репозиторијума — на пример, пројекта којем бисте желели дате свој допринос — команда која вам је потребна је `git clone`. Ако сте упознати са другим VCS системима као што је *Subversion*, приметићете да је команда `clone` а не `checkout`. Ово је битна разлика - уместо да само прави радну копију, програм Гит прима целу копију скоро свих података које има сервер. Свака верзија сваког фајла целе историје пројекта се повлачи када се покрене `git clone`. Заправо, ако дође до грешке на серверовом диску, најчешће можете да искористите скоро сваки од клонова који клијенти имају да бисте вратили сервер у стање у коме је био када је клониран (можете да изгубите неке куке на серверској страни, али сви верзионисани подаци би били ту — погледајте [Постављање програма Гит на сервер](#) за више детаља).

Репозиторијум се клонира са `git clone <url>`. На пример, ако хоћете да клонирате Гит везивну библиотеку која се зове `libgit2`, можете то да урадите овако:

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

Ово прави директоријум са именом `libgit2`, инцијализује `.git` директоријум у њему, повлачи све податке са тог репозиторијума и одјављује радну копију последње верзије. Ако одете у нови `libgit2` директоријум, видећете све фајлове из пројекта тамо, спремне за рад над њима или коришћење.

Ако желите да клонирате репозиторијум у нешто под различитим именом од `libgit2`, то можете навести следећом командом:

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

Ова команда ради исто што и претходна, с тим што се одредишни директоријум зове **mylibgit**.

Програм Гит може да користи велики број различитих трансфер протокола. Претходни пример користи <https://> протокол, али можете да користите и <git://> или корисник@сервер:патања/до/репозиторијума.git, који користи SSH протокол за пренос. [Постављање програма Гит на сервер](#) ће представити све доступне опције које сервер може подесити за приступање Гит репозиторијуму, као и предности и мане сваког од њих.

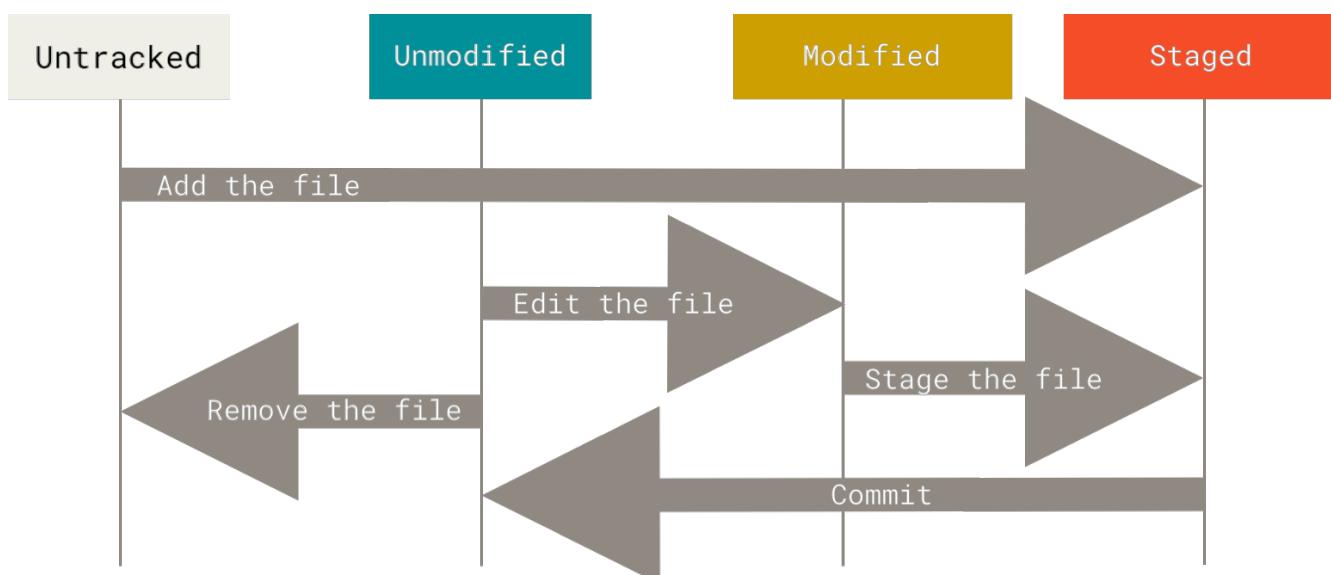
Снимање промена над репозиторијумом

Сада имате *пристојан* Гит репозиторијум и одјављене (тј. *радне копије*) све фајлове тог пројекта. Треба да направите неке измене и комитујете снимке тих измена у ваш репозиторијум сваки пут када пројекат досегне стање које желите да забележите.

Упамтите да сваки фајл у вашем радном директоријуму може бити у једном од два стања: *праћен* или *непраћен* (*tracked* или *untracked*). Праћени фајлови су фајлови који су били у последњем снимку; они могу да буду неизмењени, изменjeni или стејџовани. Укратко, праћени фајлови су сви фајлови о којима програм Гит води рачуна.

Непраћени фајлови су све остало—било који фајлови у радном директоријуму који нису били у последњем снимку и нису на стејџу. Када први пут клонирате репозиторијум, сви фајлови ће бити праћени и неизмењени јер сте их је програм Гит управо одјавио и ви још увек нисте било шта изменили.

Како будете уређивали фајлове, програм Гит ће приметити да су изменjeni, јер сте их променили у односу на стање од последњег комита. Док радите, ове изменене фајлове ћете селективно стејџовати и онда ћете комитовати све стејџоване промене, и циклус се понавља.



Слика 8. Животни циклус статуса фајлова

Провера статуса фајлова

Главни алат који користите да бисте сазнали који фајлови су у ком стању је команда `git status`. Ако покренете ову команду директно после клонирања, видећете нешто овако:

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

Ово значи да имате чист радни директоријум - другим речима, ниједан од ваших праћених фајлова није изменјен. Програм Гит такође не види никакве непраћене фајлове, иначе би овде били наведени. Коначно, команда вам каже на којој се грани налазите и информише вас да није одвојила од исте гране на серверу. Засад, грана ће увек бити `master`, што је подразумевано; овде не треба да се бринете о томе. [Гранање у програму Гит](#) ће детаљније размотрити гране и референце.

Рецимо да у пројекат додате нови фајл, обичан `README` фајл. Ако фајл није постојао раније, а ви покренете `git status`, видећете свој непраћени фајл на следећи начин:

```
$ 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)
```

Овде видите да је ваш нови `README` фајл непраћен, јер је у извештају под насловом „`Untracked files`”. Непраћено у суштини значи да програм Гит види фајл који нисте имали у претходном снимку (комиту); програм Гит га неће укључити у комитоване снимке док му ви експлицитно не наредите тако. Ради овако да не бисте случајно почели да додајете генериране бинарне фајлове или друге фајлове које нисте намеравали да додате. Пошто желите да почнете праћење фајла `README`, хајде да то и урадимо.

Праћење нових фајлова

Да бисте почели да пратите нов фајл, можете да употребите команду `git add`. Праћење `README` фајла почиње након покретања ове команде:

```
$ git add README
```

Ако поново покренете команду `status`, видећете да је ваш `README` фајл сада праћен и

стејџован за комит:

```
$ 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
```

Види се да је фајл стејџован јер је под насловом „Changes to be committed”. Ако сада комитујете, верзија фајла у тренутку када сте покренули `git add` команду је оно што ће се наћи у историјском снимку. Можда се сећате да када сте раније покренули `git init`, затим покренули и команду `git add <фајлови>`—то је било потребно да бисте почели да пратите фајлове у вашем директоријуму. Команда `git add` као аргумент узима име путање до фајла или директоријума; ако је директоријум, онда команда рекурзивно додаје све фајлове у том директоријуму.

Стејџовање изменених фајлова

Хајде да променимо фајл који је већ праћен. Ако промените фајл `CONTRIBUTING.md` који се од раније прати, па онда поново покренете команду `git status`, добићете нешто овако:

```
$ 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
```

Фајл `CONTRIBUTING.md` се појављује под насловом „Changes not staged for commit”—што значи да је фајл који је праћен сада изменјен у радном директоријуму, али још увек није на стејџу. Да бисте га стејџовали, покрените команду `git add`. `git add` је команда за више намена — можете да је користите за праћење нових фајлова, за стејџовање фајлова, као и за друге ствари као што је обележавање да су конфликти код фајлова до којих је дошло приликом спајања разрешени. Корисно је да о наредби размишљате као „додај тачно овај садржај у следећи комит”, а не као „додај овај фајл у пројекат”. Покренимо сада `git add` да стејџујемо фајл `CONTRIBUTING.md`, а онда поново покренимо `git status`:

```
$ 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
```

Оба фајла су сада стејцована и спремна за следећи комит. У овом тренутку, претпоставимо да сте се сетили још једне мале измене у **CONTRIBUTING.md** пре него што сте га комитовали. Отварате фајл и правите ту измену, и сада сте спремни за комит. Ипак, хајде да покренемо **git status** још једном:

```
$ 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
```

Шта је сад ово ког ћавола? **CONTRIBUTING.md** је наведен у стејцованим *и* у нестејцованим фајловима. Како је то могуће? Испоставља се да Гит стејцује фајл баш у тренутку када покренете команду **git add**. Ако комитујете сада, верзија **CONTRIBUTING.md** која је била када сте покренули команду **git add** ће ући у комит, а не верзија која се налази у радном директоријуму када се покрене **git commit**. Ако фајл измените након покретања команде **git add**, морате поново да покренете **git add** како бисте стејцовали последњу верзију фајла:

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

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

Кратки статус

Мада је излаз команде `git status` прилично свеобухватан, такође је доста речит. Програм Гит има и заставицу за кратки статус тако да промене можете прегледати у компактнијем облику. Ако покренете команду `git status -s` или `git status --short` добијате много простији излаз:

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

Уз нове фајлови који се не прате стоји `??`, уз нове фајлове који су додати на стејџ стоји `A` (*added*), уз изменењене фајлове стоји `M` (*modified*) и тако даље. Постоје две колоне у испису - лева колона наводи статус стејџовања, а десна колона статус радног стабла. У горњем примеру, фајл `README` је изменењен у радном директоријуму, али још увек није стејџован, док је фајл `lib/simplegit.rb` изменењен и стејџован. Фајл `Rakefile` је изменењен, стејџован, па затим поново изменењен, тако да постоје промене које су стејџоване, али и оне које нису.

Игнорисање фајлова

Често ћете имати неку групу фајлова коју не желите да програм Гит аутоматски додаје, па чак ни да вам их приказује као непраћене. То су обично аутоматски генерисани фајлови као што су логови или фајлови које генерише ваш систем за изградњу. У тим случајевима, можете да направите обрасце за испис листе фајлова који ће се поредити и ставити их у фајл под именом `.gitignore`. Ево примера `.gitignore` фајла:

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

Прва линија налаже програму Гит да игнорише све фајлове који се завршавају на `.o` или `.a` - објектне и архивне фајлове који могу бити производ изградње вашег кода. Друга линија налаже програму Гит да игнорише све фајлове који се завршавају тилдом (`~`), коју користе многи едитори текста као што је Emacs за обележавање привремених фајлова. Можете да укључите и `log`, `tmp` или `pid` директоријум; аутоматски генерисану документацију; и тако даље. Постављање `.gitignore` фајла пре него што кренете са радом је генерално добра идеја јер тако нећете случајно да комитујете фајлове које не желите у свом Гит репозиторијуму.

Правила за обрасце (шаблоне) које можете да ставите у `.gitignore` фајл су следећа:

- игноришу се празне линије и линије које почињу са `#`,
- функционишу стандардни *glob* обрасци и примењиваће се рекурзивно по целом радном стаблу,

- обрасце можете да почнете косом цртом (/) ако желите да избегнете рекурзију,
- обрасце можете да завршите косом цртом (/) ако наводите директоријум,
- образац можете да негирате тако што ћете га почети знаком узвика (!).

Glob обрасци су као поједностављени регуларни изрази које користе командна окружења. Звездица (*) хвата један или више карактера; [abc] хвата сваки карактер у великим заградама (у овом случају a, b или c); знак питања (?) хвата један карактер; а велике заграде у којима се налазе карактери раздвојени цртицом ([0-9]) хватају било који карактер између њих (у овом случају од 0 до 9). Можете да користите и две звездице за хватање угњеждених директоријума; a/**/z би хватало a/z, a/b/z, a/b/c/z, и тако даље.

Ево још једног примера `.gitignore` фајла:

```
# игнориши све .a фајлове
*.a

# или прати lib.a, мада се изнад игноришу сви .a фајлови
!lib.a

# игнориши само TODO фајлове у текућем директоријуму, а не и у поддир/TODO
/TODO

# игнориши све фајлове у build/ директоријуму
build/

# игнориши doc/notes.txt, али не и doc/server/arch.txt
doc/*.txt

# игнориши све .pdf фајлове у doc/ директоријуму (и његовим
# поддиректоријумима)
doc/**/*.pdf
```



Ако вам је потребна добра почетна тачка за ваш пројекат, *GitHub* одржава прилично свеобухватну листу добрих `.gitignore` примера фајлова за гомилу пројекта и језика на <https://github.com/github/gitignore>.



У простом случају, репозиторијум би могао да има један `.gitignore` фајл у свом кореном директоријуму који се рекурзивно примењује на цео репозиторијум. Међутим, могуће је да у поддиректоријумима постоје и додатни `.gitignore` фајлови. Правила у овим угњежденим `.gitignore` фајловима се примењују само на директоријум у коме се налазе. Репозиторијум извornog кода Линукс језгра има 206 `.gitignore` фајлова.

Детаљи у вези вишеструких `.gitignore` фајлова излазе ван оквира ове књиге; ако сте заинтересовани, погледајте `man gitignore`.

Преглед стејдованих и нестејдованих промена

Ако вам је команда `git status` превише нејасна — желите прецизно да знате шта сте променили, а не само фајлове које сте променили — можете да употребите команду `git diff`. Касније ћемо покрити `git diff` мало детаљније, али вероватно ћете је најчешће користити да бисте одговорили на следећа два питања: Шта сте променили или још нисте стејдовали? И шта сте стејдовали што ћете ускоро комитовати? Док `git status` одговара на ова питања веома опште тако што вам даје имена фајлова, `git diff` показује тачне линије које су додате и уклоњене — као да је закрпа.

Рецимо да поново уредите и стејџујете `README` фајл, па онда промените фајл `CONTRIBUTING.md` али га не стејџујете. Ако покренете команду `git status`, поново ћете видети нешто овако:

```
$ 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
```

Да бисте видели шта се променили или још нисте стејдовали, укуцајте `git diff` без осталих аргумента:

```
$ 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
```

Ова команда пореди шта се налази у вашем радном директоријуму са оним што је на стејџу. Резултат су направљене промене које још увек нисте стејдовали.

Ако желите да видите шта сте стејџовали, тј. шта ће ући у следећи комит, можете употребити `git diff --staged`. Ова команда пореди стејџоване промене са последњим комитом:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Битно је да обратите пажњу на то да `git diff` сам по себи не приказује све промене које сте направили од последњег комита — само промене које још увек нису стејџоване. Ако сте стејџовали све промене, `git diff` вам неће вратити ништа.

Као други пример, рецимо да сте стејџовали фајл `CONTRIBUTING.md` па га онда уређивали; сада можете искористити `git diff` да погледате промене у фајлу које су стејџоване и промене које нису стејџоване. Ако наше окружење изгледа овако:

```
$ 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
```

Сада можете искористити `git diff` да видите шта још увек није стејџовано:

```
$ 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` да видите шта сте досад стејцовали (`--staged` и `--cached` су 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 у спољном алату

Наставићемо да користимо команду `git diff` на разне начине кроз остатак ове књиге. Постоји још један начин да се погледају ове разлике ако вам више одговара графички или спољни програм за преглед разлика. Ако уместо `git diff` покренете `git difftool`, можићете да видите ове разлике у софтверу као што је *emerge*, *vimdiff* и многим другим (укључујући и комерцијалне производе). Покрените `git difftool -tool-help` да видите шта је доступно на вашем систему.



Комитовање промена

Сада кад је стејц постављен онако како желите, можете да комитујете своје промене. Упамтите да све што још увек није стејцовано—сви фајлови које сте креирали или изменили, а нисте покренули `git add` над њима од тренутка када сте их уредили—неће бити укључени у овај комит. Они ће остати као изменјени фајлови на диску. У овом случају, рецимо да сте последњи пут када сте покренули `git status` видели да је све стејцовано, што значи да сте спремни да комитујете промене. Најједноставнији начин да комитујете је да

укуцате `git commit`:

```
$ git commit
```

Када урадите то, покренуће се едитор који сте изабрали.



Ово је подешено на основу `EDITOR` променљиве ваше љуске - обично `vim` или `emacs`, мада можете да га конфигуришете на шта год пожелите командом `git config --global core.editor` као што сте видели у [Почетак](#)).

Едитор приказује следећи текст (у овом примеру је искоришћен едитор `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 видети да подразумевана комит порука садржи искоментарисани последњи излаз команде `git status` и једну празну линију на врху. Можете да обришете ове коментаре и упишете своју поруку, или можете да их оставите како би вам помогли да се присетите шта комитујете.



За још експлицитнији подсетник онога што сте изменили, команди `git commit` можете да проследите опцију `-v`. То ће убацити у едитор и разлику ваших промена тако да прецизно можете видети шта комитујете.

Када изађете из едитора, програм Гит прави ваш комит са том комит поруком (из које су избачени коментари и разлика).

Други начин је да укуцате комит поруку командом `commit` у *inline* режиму тако што ћете је навести након заставице `-m`, на следећи начин:

```
$ 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
```

Управо сте направили свој први комит! Можете видети да вам је комит приказао нешто о себи: у коју грану сте комитовали (`master`), која је SHA-1 контролна сума комита (`463dc4f`), колико фајлова је изменењено и статистику о линијама које су додате и обрисане у комиту.

Упамтите да комит чува снимак који сте поставили на стејџ. Све што нисте стејџовали и даље стоји тамо изменењено; можете да урадите још један комит па да и то додате у историју. Сваки пут када урадите комит, правите снимак пројекта у том стању на који касније можете да се вратите, или да вршите поређење са њим.

Прескацање стејџа

Премда може бити веома корисно да комитујете ствари тачно онако како желите, стављање на стејџ понекад уме да буде мало комплексније од онога што је потребно за ваш процес рада. Ако желите да прескочите стејџ, програм Гит нуди једноставну пречицу. Додавањем опције `-a` команди `git commit`, програму Гит налажете да аутоматски стејџује сваки фајл који је већ праћен пре него што је урађен комит, што вам омогућава да прескочите `git add` део:

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

    modified:   CONTRIBUTING.md

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

Обратите пажњу на то да на овај начин нисте морали да покренете `git add` за `CONTRIBUTING.md` фајл пре него што сте комитовали. То је из разлога што заставица `-a` укључује све изменење фајлове. Ово је згодно, али будите опрезни; понекад због ове заставице можете укључити и промене које не желите да се укључе.

Уклањање фајлова

Да бисте уклонили фајл из програма Гит, морате да га уклоните из праћених фајлова (тачније, да га склоните са стејџа), па да онда урадите комит. Команда `git rm` ради управо то, а такође и уклања фајл из радног директоријума тако да га након тога више не видите као фајл који се не прати.

Ако једноставно уклоните фајл из радног директоријума, он се појављује под „Changes not staged for commit“ (односно, *измене који нису стејџоване*) у излазу команде `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")
```

Ако онда покренете `git rm`, ова команда стејџује брисање фајла:

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

    deleted:   PROJECTS.md
```

Следећи пут када комитујете, фајл ће нестати и више неће бити праћен. Ако сте фајл изменили, или сте га већ додали у индекс, уклањање морате да форсирате опцијом `-f`. Ово је сигурносна мера која обезбеђује да случајно не обришете податке који још нису сачувани у снимку, јер у том случају програм Гит не може да вам их врати натраг.

Још једна корисна ствар коју ћете можда желети да урадите је да задржите фајл у радном стаблу али да га уклоните са стејџа. Другим речима, можда желите да задржите фајл на хард диску али више не желите да га програм Гит прати. Ово је посебно корисно ако сте нешто заборавили да додате у фајл `.gitignore` и случајно га стејџовали, као што је велики лог фајл или гомила компајлираних `.a` фајлова. Да бисте урадили ово, употребите опцију `--cached`:

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

Команди `git rm` можете да прослеђујете фајлове, директоријуме и *file-glob* обрасце. То значи да можете да радите ствари као што је:

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

Приметите обрнуту косу црту (`\`) испред `*`. Ово је неопходно јер програм Гит има своје развијање имена фајлова које се ради уз развијање које обавља љуска. Ова команда из `log/` директоријума уклања све фајлове који имају `.log` екstenзију. Или можете да урадите нешто слично овоме:

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

чиме уклањате све фајлове чије се име завршава на `~`.

Премештање фајлова

За разлику од многих других VCS система, програм Гит не прати померање фајлова експлицитно. Ако у програму Гит промените име фајлу, нема никаквих метаподатака сачуваних у програму Гит који говоре да сте фајлу променили име. Међутим, програм Гит је у стању да то прилично паметно закључи - мало ћемо се касније позабавити откривањем премештених фајлова.

Имајући ово у виду, мало је конфузно то што програм Гит има `mv` команду. Ако у програму Гит желите да промените име фајлу, можете извршити нешто слично овоме:

```
$ git mv файл_од файл_на
```

и радиће како треба. Заправо, ако покренете нешто тако па онда погледате статус, видећете да програм Гит то сматра фајлом чије је име промењено:

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

    renamed:   README.md -> README
```

Ипак, ово је еквивалентно са извршавањем следећих команда:

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

Програм Гит имплицитно схвата да се ради о промени имена фајла, тако да није важно да ли ћете име мењати на овај начин или командом `mv`. Једина права разлика је то што је `git mv` једна команда уместо три — удобније је користити њу. Штавише, можете да користите било који алат да фајлу промените име, а да касније примените `rm` и `add`, пре него што комитујете.

Преглед историје комитова

Када сте направили неколико комитова, или сте клонирали репозиторијум са постојећом историјом комитова, вероватно ћете хтети да погледате уназад да видите шта се дешавало. Најосновнији и најмоћнији алат за ово је команда `git log`.

Примери који следе користе веома једноставан пројекат који се зове „simplegit”. Да бисте преузели пројекат, извршите:

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

Када покренете `git log` команду у овом пројекту, треба да добијете излаз који личи овом:

```
$ 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
```

Са подразумеваним подешавањима, без аргумента, `git log` ће излисти комитове који су направљени у том репозиторијуму у обрнутом хронолошком редоследу—односно, најскорији комит ће се појавити први. Као што видите, ова команда исписује листу свих комитова са њиховом SHA-1 контролном сумом, ауторовим именом и мејлом, датумом када је написан и комит поруком.

Постоји велики број опција за команду `git log` које ће вам помоћи да се прикаже тачно оно што тражите. Овде ћемо представити неке најпопуларније.

Једна од опција које су од највеће помоћи је `-p` или `--patch`, која приказује разлику (*patch* излаз) уведену у сваком комиту. Можете и да ограничите број приказаних лог уноса, па тако `-2` приказује само два последња комита.

```
$ 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
```

Ова опција приказује исту информацију с тим што разлика директно следи након сваке ставке. Ово је веома корисно када се врши преглед кода или да се брзо сазна шта се дододило са кодом након низа комитова које је урадила особа која даје допринос пројекту. Са `git log` можете да употребите и низ опција за скраћење. На пример, ако желите да видите неке скраћене статистике за сваки комит, можете искористити опцију `--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(+)
```

Као што видите, опција `--stat` испод сваког комита исписује листу изменјених фајлова, колико је фајлова промењено и колико је линија у тим фајловима додато и обрисано. На крају исписује и резиме информација.

Још једна веома корисна опција је `--pretty`. Ова опција мења излаз лога у неки формат који није подразумевани. Припремљено је неколико опција које можете да користите. Вредност `oneline` за ову опцију исписује сваки комит у једну линију, што је веома корисно ако гледате пуно комитова. Ту су и вредности `short`, `full` и `fuller` које исписују у сличном формату али са мање или више информација, респективно:

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

Најзанимљивија вредност опције је `format`, која допушта да наведете сопствени формат за излаз лога. Ово је посебно корисно када генеришете излаз који треба да парсира машина — пошто експлицитно наводите формат, знате да се неће променити када се

програм Гит ажурира.

```
$ 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
```

Корисне опције за `git log --pretty=format` представља листу неких од кориснијих спецификатора које узима `format`.

Табела 1. Корисне опције за `git log --pretty=format`

Спецификатор	Опис излаза
<code>%H</code>	Хеш комита
<code>%h</code>	Скраћени хеш комита
<code>%T</code>	Хеш стабла
<code>%t</code>	Скраћени хеш стабла
<code>%P</code>	Хешеви родитеља
<code>%p</code>	Скраћени хешеви родитеља
<code>%an</code>	Име аутора
<code>%ae</code>	Мејл аутора
<code>%ad</code>	Датум аутора (формат поштује <code>--date=option</code>)
<code>%ar</code>	Датум аутора, релативан
<code>%cn</code>	Име комитера
<code>%ce</code>	Мејл комитера
<code>%cd</code>	Датум комитера
<code>%cr</code>	Датум комитера, релативан
<code>%s</code>	Наслов

Можда се питате која је разлика између *автора* и *комитера*. Аутор је особа која је првобитно направила фајл, а комитер је особа која је последња допринела промене. Значи ако пошаљете закрпу пројекту и један од главних чланова је примени, обојица добијате признање — ви као аутор, а главни члан пројекта као комитер. Ову разлику ћемо детаљније размотрити у [Дистрибуирани Гит](#).

Вредности опције `online` и `format` су посебно корисне са још једном `log` опцијом која се зове `--graph`. Ова опција додаје мали ASCII график који приказује вашу грану и историју спајања.

```
$ 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
```

Ова врста излаза ће постати занимљивија када у следећем поглављу прођемо кроз гранање и спајање.

То су само неке једноставне опције за форматирање излаза команде `git log` - има их још пуно. У [Честе опције команде git log](#) су наведене опције које смо досад прошли, као и неке друге које би могле да буду корисне, уз кратко објашњење о начину на који мењају излаз.

Табела 2. Честе опције команде `git log`

Опција	Опис
<code>-p</code>	Показује закрпу која је уведена сваким комитом.
<code>--stat</code>	Приказује статистику фајлове измене у сваком комиту.
<code>--shortstat</code>	Приказује само линију промене/додавања/брисања из <code>--stat</code> команде.
<code>--name-only</code>	Приказује листу изменених фајлова након информације о комиту.
<code>--name-status</code>	Приказује листу променених фајлова из информацију о линијама које су промењене/додате/обрисане.
<code>--abbrev-commit</code>	Приказује само првих неколико слова из SHA-1 контролне суме уместо свих 40.
<code>--relative-date</code>	Приказује датум у релативном формату (на пример, „2 weeks ago“) уместо комплетног формата датума.
<code>--graph</code>	Приказује ASCII графикон гране и историје спајања поред лога.
<code>--pretty</code>	Приказује комитове у алтернативном формату. Међу вредностима опције су <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , и <code>format</code> (где би требало да наведете сопствени формат).

Ограничавање исписа лога

Поред опција за форматирање излаза, `git log` прима и велики број корисних опција за ограничавање — другим речима, опција којима можете задати приказ само једног подскупа комитова. Већ сте видели једну од њих—опцију `-2`, која приказује само последња два комита. Заправо, можете да укуцате `-<n>`, где је `n` било који природан број и на тај начин прикажете последњих `n` комитова. У стварности је нећете често користити јер програм Гит подразумевано пајпјује сви излаз кроз пејџер тако да ћете излаз лога увек гледати страну по страну.

Међутим, опције које ограничавају време, као што су `--since` и `--until`, су веома корисне. На пример, следећа команда враћа листу комитова који су урађени у последње две недеље:

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

Ова команда ради са пуно формата — можете да наведете тачан датум као „2008-01-15” или релативни датум као „2 years 1 day 3 minutes ago”.

Листу комитова можете филтрирати и тако да задовољи неки критеријум претраге. Опција `--author` вам омогућава да прикажете комитове само од наведеног аутора, а са `--grep` можете тражити кључне речи у комит порукама.



Можете да задате и више од једне инстанце `--author` и `--grep` критеријума претраге који ће онда да ограниче излаз комитова тако да се подударају са *било којим* од `--author` шаблона и *било којим* од `--grep` шаблона; међутим, додавање `--all-match` опције додатно ограничава излаз на само оне комитове који се подударају са *свим* `--grep` шаблонима.

Још један веома користан филтер је `-S` опција (која се у жаргону назива Гитова „ріскаже” тј. пијук опција) која узима стринг и показује само комитове који су променили број појављивања тог стринга у коду. На пример, ако желите да нађете последњи комит који је додао или обрисао референцу на неку одређену функцију, можете да позовете

```
$ git log -S име_функције
```

Последња веома корисна опција коју као филтер можете проследити команди `git log` је путања. Ако наведете директоријум или име фајла, излаз лога можете да ограничите само на комитове који су увели измене над тим фајловима. Ово је увек последња опција и генерално се испред ње стављају две цртице (`--`) да би се путање одвојиле од осталих опција:

```
$ git log -- путања/до/фајла
```

Опције за ограничавање команде `git log` као подсетник приказује неколико ових и неколико других честих опција.

Табела 3. Опције за ограничавање команде `git log`

Опција	Опис
<code>-<n></code>	Приказује само последњих <i>n</i> комитова.
<code>--since, --after</code>	Приказује комитове који су направљени после наведеног датума.
<code>--until, --before</code>	Приказује комитове који су направљени пре наведеног датума.
<code>--author</code>	Приказује само комитове код којих се наведени стринг подудара са именом аутора.

Опција	Опис
--committer	Приказује само комитове код којих наведени стринг подудара са именом комитера.
--ггр	Приказује само комитове код којих комит порука садржи наведени стринг.
-S	Приказује само комитове који додају или бришу код који се подудара са стрингом.

На пример, ако желите да видите комитове који су изменили тест фајлове у Гит историји извornог кода, а урадио их је корисник Junio Hamano у октобру 2008 и нису комити спајања, можете да извршите овакву команду:

```
$ 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
```

Од скоро 40.000 комитова у Гит историји извornог кода, ова команда је приказује оних 6 који задовољавају наведене услове.



Спречавање приказа комита спајања

Зависно од процеса рада који се користи у вашем репозиторијуму, може бити да поприличан проценат комита у лог историји представљају комити спајања, који обично не пружају корисне информације. Ако желите да спречите затрпавање лог историје комитима спајања, једноставно додајте лог опцију **--no-merges**.

Опозив

У било ком тренутку можете пожелети да нешто опозовете (*undo*). Овде ћемо обрадити неколико основних алата за опозив радњи. Будите опрезни, јер не можете увек да опозовете неке од ових опозива. Ово је једна од ретких области програма Гит где можете изгубити неки део вашег рада у случају ако поступите на погрешан начин.

Чест случај када је потребан опозив се јавља онда комитујете прерано и можда заборавите да додате неке фајлове, или погрешно напишете своју комит поруку. Ако желите да поново урадите тај комит, направите додатне измене које сте заборавили, стејџујте их и поново комитујте опцијом **--amend**:

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

Ова команда узима ваш стејџ и користи га за комит. Ако од последњег комита нисте направили никакве измене (на пример, покренете ову команду одмах након последњег комита), онда ће снимак изгледати потпуно исто и све што ћете променити је комит порука.

Појављује се исти едитор за комит поруку, али се у њему већ налази укуцана порука из претходног комита. Поруку можете уредити као и увек, али тиме преписујете претходни комит.

Као пример, ако комитујете и онда схватите да сте заборавили да стејџујете промене у фајлу који сте желели да додате у овај комит, можете да урадите нешто овако:

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

На крају остаје само један комит — други комит замењује резултате првог.

Важно је да разумете да када преправљате свој последњи комит, ви га уствари не поправљате већ га комплетно *замењујете* потпуно новим, побољшаним комитом који склања стари с пута и на његово места поставља нови комит. У суштини, исто је као да се претходни комит није ни дододио, па се неће ни приказивати у историји вашег репозиторијума.



Очигледна вредност преправљања комитова је у томе што мала побољшања свог последњег комита можете да урадите без затрпавања историје вашег репозиторијума порукама као што су „Уупс, заборавио сам да додам фајл“ или „Axxx, исправка грешке у куцању у последњем комиту“.



Преправљајте само комите који су још увек у локалу и нису гурнути негде. Преправљање комита који су раније гурнути и форсирено гурање гране ће правити проблеме вашим сарадницима. За више детаља о томе шта се дешава ако ово урадите, као и начин за опоравак ако се налазите на пријемном крају, прочитајте [Опасности ребазирања](#).

Уклањање фајла са стејџа

Следећа два одељка показују како да радите са променама на стејџу и радном директоријуму. Добро је то што вас команда коју користите да би одредили стање ове две области такође подсећа и на начин за опозив промена које над њима направите. На пример, рецимо да сте променили два фајла и да желите да их комитујете као две посебне измене, али сте случајно укуцали `git add *` и тако их оба додали на стејџ. Како да један од њих склоните са стејџа? Команда `git status` вас подсећа на то:

```
$ 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
```

Одмах испод текста „Changes to be committed” пише да треба да употребите `git reset HEAD <file>...` ако фајл желите да уклоните са стејџа. Тада ћемо искористити да фајл `CONTRIBUTING.md` уклонимо са стејџа:

```
$ 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
```

Ова команда је помало чудна, али обавља посао. Фајл `CONTRIBUTING.md` је изменењен или поново није на стејџу.



Тачно је да `git reset` може бити опасна команда, посебно ако је позовете са заставицом `--hard`. Међутим, у случају који је описан изнад, фајл у радном директоријуму се не дира, тако да је команда релативно сигурна.

Засад је овај чаробни позив све што је потребно да знате о `git reset` команди. Много детаљније ћемо испитати шта тачно ради `reset` у [Демистификовани ресет](#) када ћемо показати и како да овладате њоме и постигнете неке веома занимљиве ствари.

Уклањање измена са изменењеног фајла

Шта ако одлучите да не желите да задржите измене у фајлу `CONTRIBUTING.md`? Како да једноставно уклоните измене—да га вратите на стање у каквом је био када сте последњи пут начинили комит (или урадили почетно клонирање, или како год га поставили у радни директоријум)? Срећом, `git status` вам говори како и то да урадите. У прошлом примеру излаза, фајлови који нису били на стејџу били су представљени овако:

```
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
```

Каже вам како да експлицитно одбаците промене које сте направили. Хајде да урадимо то што нам саветује:

```
$ 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 видети да су се промене вратиле на старо.



Важно је да разумете да је `git checkout -- <файл>` опасна команда. Све локалне промене које сте направили над тим фајлом су нестале — програм Гит га је управо преписао последњом стејџованом или комитованом верзијом. Никада немојте да користите ову команду осим ако нисте потпуно сигурни да вам промене које сте направили више не требају.

Ако ипак жељите да задржите промене које сте направили у фајлу, али за сада не жељите да вам стоји на путу, у [Гранање у програму Гит](#) ћете сазнати како да употребите скривање (*stashing*) и гранање (*branching*); обично су то бољи начини за рад.

Упамтите, у програму Гит се све што је *комитовано* скоро увек може опоравити. Чак и комитови који су били на гранама које су обрисане, или комитови који су преписани `--amend` комитом могу да се опораве (погледајте [Опоравак података](#) у вези опоравка података). Ипак, све што изгубите, а никад није било комитовано, вероватно више никад нећете видети.

Опозив ствари са `git restore`

Верзија 2.23.0 програма Гит је увела нову команду: `git restore`. У суштини је то алтернатива команди `git reset` коју смо управо описали. Од верзије 2.23.0 програма Гит надаље, програм Гит ће за многе операције опозива уместо `git reset` користити `git restore`.

Хајде да поново прођемо кроз наше кораке, опозовемо ствари командом `git restore` уместо са `git reset`.

Уклањање фајла са стејџа помоћу `git restore`

Следећа два одељка показују како да радите са изменама стејџа и радног директоријума директно помоћу команде `git restore`. Згодно је што вас команда коју користите да одредите

статус те две области уједно и подсећа како да опозовете промене које начините над њима. На пример, претпоставимо да сте изменили два фајла и желите то да комитујете као две одвојене измене, али сте случајно откуцали `git add *` и стејцовали оба. Како да један од њих склоните са стејџа? Команда `git status` вас подсећа:

```
$ 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
```

Непосредно испод текста „Changes to be committed” пише „use `git restore --staged <file>...` to unstage” (употребите `git restore --staged <фајл>...` да уклоните са стејџа). Дакле, хајде да искористимо овај савет и уклонимо са стејџа фајл `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
```

Фајл `CONTRIBUTING.md` је изменењен, али је поново ван стејџа.

Уклањање промена из изменењеног фајла помоћу `git restore`

Шта ако схватите да не желите задржати измене над фајлом `CONTRIBUTING.md`? Како једноставно можете да уклоните измене — тј. да га вратите у стање у коме је био када сте направили последњи комит (или урадили почетно клонирање, или шта год урадили да га поставите у радни директоријум)? Срећом, `git status` вам такође говори како и то да урадите. У излазу последњег примера, област ван стејџа изгледа овако:

```
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
```

Ово вам говори прилично директно како да одбаците измене које сте направили. Хајде да урадимо то што каже:

```
$ 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 је да разумете да је `git restore <фајл>` опасна команда. Све локалне промене које сте направили над тим фајлом су нестале — програм Гит га је управо преписао последњом стејџованом или комитованом верзијом. Никада немојте да користите ову команду осим ако нисте потпуно сигурни да вам промене које сте направили више не требају.

Рад са удаљеним репозиторијумима

Да бисте могли да сарађујете на било ком Гит пројекту, морате да научите како да организујете удаљене репозиторијуме (*remote repositories*, често само *remotes*). Удаљени репозиторијуми су верзије пројекта које су хостоване на Интернету или негде на мрежи. Можете их имати неколико, од којих вам је сваки у општем случају или доступан само за читање (*read-only*) или и за читање и за упис (*read/write*). Сарадња са другима подразумева управљање тим удаљеним репозиторијумима и гурање (*push*) и повлачење (*pull*) података када је потребно поделити рад. Управљање удаљеним репозиторијумима подразумева да знате како се додају удаљени репозиторијуми, уклањају они који више нису важећи, управљање разним удаљеним гранама и њихово дефинисање као оне које се прате или оне које се не прате и још тога. У овом одељку ћемо прећи неке од тих вештина управљања удаљеним репозиторијумима.

Удаљени репозиторијуми могу да се налазе и на вашој локалној машини.



Сасвим је могуће да радите са „remote” (удаљеним) репозиторијумом који се уствари налази на истом хосту на којем тренутно радите. Реч „remote” не повлачи обавезно да се репозиторијум налази негде на Интернету или на мрежи, већ само да је на неком другом месту. За рад са таквим удаљеним репозиторијумом су још увек потребне стандардне операције гурање, повлачење и добављање (*fetching*) као и са било којим другим удаљеним репозиторијумом.

Приказ удаљених репозиторијума

Да бисте видели које удаљени сервери су конфигурисани, можете да извршијете команду `git remote`. Она приказује скраћена имена свих удаљених репозиторијума које сте задали. Ако сте клонирали репозиторијум, требало би да барем видите `origin` - то је подразумевано име које програм Гит даје серверу са кога сте извршили клонирање:

```
$ 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
```

Такође можете да наведете и `-v`, што ће приказати URL адресе које је програм Гит ускладиштио уз кратко име које се користе када се чита и пише у тај удаљени репозиторијум:

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

Ако имате више од једног удаљеног репозиторијума, ова команда ће их све излистати. На пример, репозиторијум са више удаљених репозиторијума за рад са неколико сарадника би могао да изгледа овако:

```
$ 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)
```

Ово значи да прилично једноставно можемо да повучемо доприносе од било ког од ових корисника. Можда додатно имамо и дозволу да шаљемо једном или више њих, мада то не можемо да видимо одавде.

Обратите пажњу на то како удаљени репозиторијуми користе разне протоколе; о овоме ћемо детаљније причати у [Гранање у програму Гит](#).

Додавање удаљених репозиторијума

Поменули смо и показали на неколико примера како команда `git clone` имплицитно за вас додаје `origin` удаљени репозиторијум. Ево како се експлицитно додаје нови удаљени

репозиторијум. Ако желите да додате нови удаљени Гит репозиторијум као кратко име на које ћете касније лако моћи да се позовете, извршите команду `git remote add <кратко_име> <url>`:

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

Сада у командној линији можете да користите `pb` уместо комплетне URL адресе. На пример, ако хоћете да добавите све информације које има Пол, а које ви још увек немате у свом репозиторијуму, можете да извршите `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
```

Полова `master` грана је сада доступна локално као `pb/master` — можете да је спојите са неком од ваших грана, или у том тренутку можете да одјавите локалну грану ако желите да је прегледате. У [Гранање у програму Гит](#) ћемо се много детаљније позабавити тиме шта су тачно гране и како се користе.

Добављање и повлачење из удаљених репозиторијума

Као што сте управо видели, податке из својих удаљених пројекта добављате тако што извршите:

```
$ git fetch <име-удаљеног-репозиторијума>
```

Команда одлази до тог удаљеног пројекта и повлачи све податке са њега које још увек немате. Након што урадите ово, требало би да имате референце на све гране из тог удаљеног репозиторијума, које можете да спојите са својим или да их у било ком тренутку истражите.

Ако клонирате репозиторијум, команда аутоматски додаје тај удаљени репозиторијум под именом „`origin`”. Дакле, `git fetch origin` добавља сав нови рад који је гурнут на тај сервер након тренутка када сте га клонирали (или добавили с њега). Битно је напоменути да команда `git fetch` само преузима податке у ваш локални репозиторијум - не спаја их

аутоматски са вашим радом нити мења оно на чиму тренутно радите. Морате ручно да их спојите када будете спремни.

Ако је ваша текућа грана подешена тако да прати удаљену грану (за више информација погледајте следећи одељак и [Гранање у програму Гит](#)), можете да искористите команду `git pull` која аутоматски добавља, па затим спаја ту удаљену грану са вашом тренутном граном. За вас је ово можда једноставнији или удобнији процес рада; и са подразумеваним подешавањима, команда `git clone` аутоматски поставља локалну `master` грану тако да прати удаљену `master` грану (или како год се зове подразумевана грана) на серверу са којег сте извршили клонирање. Извршавање `git pull` у општем случају добавља податке са сервера са ког сте првобитно клонирали репозиторијум и аутоматски покушава да их споји са кодом на коме тренутно радите.

Почевши од верзије 2.27 програма Гит па надаље, команда `git pull` ће издати упозорење у случају да променљива `pull.rebase` није постављена. Програм Гит ће наставити да издаје упозорење све док не поставите ову променљиву.



Ако желите подразумевано понашање програма гит (премотавање унапред (*fast-forward*) ако је то могуће, а у случају да није, креирање комита спајања):
`git config --global pull.rebase "false"`

Ако желите да се одради ребазирање (*rebase*) када се повлачи: `git config --global pull.rebase "true"`

Гурање ка удаљеним репозиторијумима

Када ваш пројекат дође у стање у којем ваш рад пожелите да поделите са другима, морате да га погурате узводно. Команда за ово је проста: `git push <име-удаљеног> <име-гране>`. Ако своју `master` желите да погурате вашем `origin` серверу (још једном, клонирање у општем случају аутоматски поставља оба ова имена уместо вас), онда можете извршити следећу команду којом гурате на сервер све комитове које сте урадили:

```
$ git push origin master
```

Ова команда функционише само ако сте извршили клонирање са сервера на којем имате дозволу за упис и ако у међувремену нико није слao податке. Ако заједно са неким истовремено клонирате, па онда он или она пошаље измене узводно, а затим ви покушате да пошаљете своје, ваше измене ће с правом бити одбијене. Мораћете најпре да добавите његов или њен рад и да га уградите у свој пре него што вам буде дозвољено да пошаљете било шта. Погледајте [Гранање у програму Гит](#) за детаљније објашњење о томе како се шаље на удаљене сервере.

Истраживање удаљеног репозиторијума

Ако желите да видите више информација о одређеном удаљеном репозиторијуму, можете да искористите команду `git remote show <име-удаљеног>`. Ако ову команду покренете са одређеним кратким именом, као што је `origin`, добићете нешто слично овоме:

```
$ 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)
```

То ће приказати URL адресе за удаљени репозиторијум као и информацију о грани која се прати. Команда вам чини услугу тиме што вам каже да ако сте на `master` грани и ако извршите `git pull`, аутоматски ће спојити `master` грану удаљеног репозиторијума са локалном након што је добави. Такође приказује и удаљене референце које су повучене.

Ово је прост пример са којим ћете се вероватно сусретати. Међутим, када мало озбиљније почнете да користите програм Гит, видећете много више информација ако извршите `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)
```

Ова команда показује у коју грана се аутоматски туре када извршите `git push` док сте на одређеним гранама. Такође вам показује и које удаљене гране на серверу још увек немате,

које удаљене гране имате које су уклоњене са севера, и више локалних грана које аутоматски могу да се споје са својим одговарајућим удаљеним гранама које се прате онда кад извршите `git pull`.

Уклањање и промена имена удаљеним репозиторијумима

Ако желите да промените кратко име удаљеног репозиторијума, извршите команду `git remote rename`. На пример, ако `pb` желите да промените у `paul`, то можете урадити помоћу наредбе `git remote rename`:

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

Вреди напоменути да ово мења и имена свим вашим удаљеним гранама које се прате. Оно на шта је раније указивало `pb/master`, сада је `paul/master`.

Ако из неког разлога желите да уклоните удаљени репозиторијум — померили сте сервер или више не користите одређени мирор, или сарадник можда више не даје допринос пројекту — можете да искористите или `git remote remove` или `git remote rm`:

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

Једном када на овај начин уклоните референцу на удаљени репозиторијум, бришу се и све гране које прате удаљене, као и сва подешавања придружена том удаљеном репозиторијуму.

Означавање

Као и већина VCS система, програм Гит има могућност означавања (таговања, обележавања) одређених тачака у историји као важних. Ова функционалност се обично користи да би се обележиле тачке издања нових верзија (`v1.0`, `v2.0` и тако даље). У овом одељку ћете научити како да прикажете постојеће ознаке, како да направите и бришете ознаке, као и које различите врсте ознака постоје.

Испис ознака

Исписивање постојећих ознака у програму Гит је прилично једноставно. Само треба да откуцате `git tag` (уз необавезно `-l` или `--list`):

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

Ова команда ће приказати ознаке сортиране по абецедном редоследу; редослед у којем се приказују нема посебну важност.

Ознаке можете и да претражујете по неком одређеном обрасцу. На пример, репозиторијум изворног кода програма Гит садржи преко 500 ознака. Ако вас интересује само да погледате серију 1.8.5, можете да извршите следеће:

```
$ 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
```

За приказ ознака помоћу цокера морате да користите опцију `-l` или `--list`

Ако вам је потребна само комплетна листа ознака, извршавање команде `git tag` имплицитно подразумева да то желите листу, па вам то и приказује; у овом случају није обавезна употреба `-l` или `--list`.

Међутим, ако наводите шаблон са цокером којим желите да гађате више имена ознака, обавезна је употреба `-l` или `--list`.

Креирање ознака

Програм Гит користи две основне врсте ознака: *лаке (lightweight)* и *прибележене (annotated)*.

Лаке ознаке доста личе на грану која се не мења — то је само показивач на одређени комит.

С друге стране, прибележене ознаке се у бази података програма Гит чувају као пуни објекти. Они добијају своју контролну суму; садрже име и мејл особе која је поставила ознаку као и датум када је то учињено; имају поруку означавања; и могу да се потпишу и верификују помоћу *GNU Privacy Guard (GPG)*. Углавном се препоручује да правите прибележене ознаке како бисте имали све ове информације; али ако желите привремену ознаку, или из неког разлога не желите да чувате остале податке о ознаки, доступне су вам и лаке ознаке.

Прибележене ознаке

Креирање прибележене ознаке у програму Гит је једноставно. Најлакши начин је да наведете **-a** када извршавате команду **tag**:

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

Са **-m** наводите поруку означавања која се чува заједно са ознаком. Ако не наведете поруку за прибележену ознаку, програм Гит покреће едитор да бисте могли да је унесете.

Податке о ознаци можете видети заједно са комитом који је њоме обележен командом **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
```

Ово приказује информације о особи која је додала ознаку, датум када је комит означен и прибележену поруку пре него што прикаже податке о самом комиту.

Лаке ознаке

Други начин за означавање комитова је помоћу лаких ознака. Ово је у суштини контролна suma комита смештена у фајл — не чувају се никакве друге информације. Ако желите да направите лаку ознаку, једноставно немојте да додате ниједну од опција **-a**, **-s** и **-m**, само наведите име ознаке:

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

Ако сада извршите `git show` над ознаком, нећете видети додатне информације. Команда приказује само комит:

```
$ 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

Накнадно означавање

Комитове можете да означавате и након што их прођете. Претпоставимо да историја ваших комитова изгледа овако:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbe Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fce02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaef7d07798508422908a Update readme
```

Претпоставимо сада да сте заборавили да означите пројекат код верзије v1.2, која треба да је уз комит „Update rakefile“. Ознаку можете да додате иако сте већ прошли тај комит. Да бисте га означили, потребно је да наведете контролну суму комита (или њен део) на крај команде:

```
$ git tag -a v1.2 9fce02
```

Сада можете видети да сте комит означили:

```
$ 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 9fce802d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

  Update rakefile
...
```

Дељење ознака

Са подразумеваним подешавањима, команда `git push` не преноси ознаке удаљеним серверима. Мораћете експлицитно да гурнете ознаке дејеном серверу након што их креирате. Овај процес је исти као дељење удаљених грана — можете да га покренете са `git push origin <име-ознаке>`.

```
$ 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 пуно ознака које желите да гурнете одједном, можете употребити и опцију `--tags` команде `git push`. Ово ће пренети све ознаке на удаљени сервер који тренутно нису тамо.

```
$ 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
```

Сада, када неко клонира или повуче са вашег репозиторијума, добиће и све ваше ознаке.



git push гура обе врсте ознака

git push <име_удаљеног> --tags ће гурнути и лаке и прибележене ознаке. Тренутно не постоји опција да се турну само лаке ознаке, али ако употребите **git push <име_удаљеног> --follow-tags** на удаљени репозиторијум се гурају само прибележене ознаке.

Брисање ознака

Ако желите да обришете ознаку у свом локалном репозиторијуму, можете употребити **git tag -d <име_ознаке>**. На пример, могли бисмо да уклонимо лаку ознаку из горњег примера на следећи начин:

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

Имајте на уму да ово не уклања ознаку са удаљених сервера. Постоје две уобичајене варијације брисања ознаке са удаљеног сервера.

Прва је **git push <име_удаљеног> :refs/tags/<име_ознаке>**:

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

Начин на који се интерпретира ова команда је да се чита као нулта вредност испред двотачке која се гура на име удаљене ознаке, што у суштини брише ту ознаку.

Други (и интуитивнији) начин за брисање удаљене ознаке је:

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

Одјављивање ознака

Ако желите да погледате верзије фајлова на које показује ознака, можете да извршите **git checkout** те ознаке, мада ово поставља ваш репозиторијум у „detached HEAD” стање, што има

неке лоше споредне ефекте:

```
$ 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
```

Ако у „detached HEAD” стању направите измене па креирате комит, ознака остаје иста, али ваш нови комит не припада ниједној грани и неће моћи да му се приступи, осим ако се не употреби хеш комита. Дакле, ако морате направити измене—на пример, рецимо да исправљате баг у старијој верзији—у општем случају ћете хтети да направите грану:

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

Ако урадите ово и направите комит, грана `version2` ће се мало разликовати од ваше `v2.0.0` ознаке, јер ће се вашим новим изменама померити унапред, па будите опрезни.

Гит алијаси

Пре него што завршимо ово поглавље о основама програма Гит, постоји још једна мала ствар која ће учинити да ваше искуство у раду са програмом Гит буде једноставније и лакше: алијаси. Нећемо их користити у овој књизи да би текст био јаснији, али ако наставите да користите програм Гит иоле редовно, алијаси су нешто што би требало да познајете.

Програм Гит не закључује аутоматски о којој команди се ради док је делимично откуцана.

Ако не желите да откуцате комплетан текст сваке команде програма Гит, можете лако да подесите алијас за сваку команду помоћу `git config`. Ево неколико примера које бисте могли да поставите:

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

Ово значи да уместо да куцате, на пример, `git commit`,овољно је да откуцате само `git ci`. Што више будете користили програм Гит, вероватно ћете чешће почети почети да користите и остале команде; не оклевавте да им додате алијасе.

Ова техника је такође веома корисна за креирање команди за које мислите да би требало да постоје. На пример, ако желите да решите проблем употребљивости приликом уклањања фајла са стејџа, јер куцате команду која у себи чак и не садржи реч `stage`, можете да направите свој 'unstage' алијас у програму Гит:

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

То значи да су следеће две команде еквивалентне:

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

На овај начин су ствари много јасније. Људи често додају и `last` команду на следећи начин:

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

Овако једноставно можете да погледате последњи комит:

```
$ git last  
commit 66938dae3329c7aebe598c2246a8ebaf90d04646  
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>
```

Као што видите, програм Гит само замењује нову команду оним на шта указује алијас. Ипак, можда бисте волели да покренете спољну команду, а не подкоманду програма Гит. У том случају, команду треба да почнете карактером `!`. Ово је корисно када пишете сопствене алате који раде са Гит репозиторијумом. Ово можемо показати додавањем алијаса `gitk` за

`git visual`:

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

Резиме

Сада можете да извршите све основне локалне Гит операције—знате како да направите или клонирате репозиторијум, да правите измене, стејџујете и комитујете те измене, и да гледате историју свих промена кроз које је прошао репозиторијум. Сада ћемо представити најбољу ствар у вези програма Гит: модел гранања.

Гранање у програму Гит

Скоро сваки VCS има неку врсту подршке за гранање. Гранање значи да се одвајате од главне линије развоја програма и да настављате рад без утицаја на ту главну линију. У многим VCS алатима, ово је донекле захтеван процес који укључује прављење нове копије директоријума са изворним кодом, што у случају великих пројеката може да потраје.

Неки људи модел гранања у програму Гит зову првокласном особином и заиста, та могућност издаваја програм Гит из мноштва осталих VCS система. Зашто је то тако посебно? Начин на који програм Гит прави гране је невероватно једноставан за обраду, што чини да се операције гранања извршавају скоро тренутно, а скакање с једне на другу грану је углавном подједнако брзо. За разлику од многих других VCS система, програм Гит охрабрује процесе рада који често користе гранање и спајање, чак и неколико пута током једног дана. Разумевањем и овладавањем овом техником добијате моћно и јединствено оруђе које у потпуности може променити начин на који развијате свој производ.

Укратко о гранању

Да бисмо заиста разумели како програм Гит барата гранањем, морамо да се вратимо корак уназад и да истражимо како програм Гит чува податке.

Као што се можда сећате из [Шта је Гит?](#), програм Гит не чува податке као низ скупова промена или разлика, већ као низ *снимака*.

Када направите комит, програм Гит чува комит објекат који садржи показивач на снимак садржаја који сте стејџовали. Овај објекат такође садржи и ауторово име и мејл адресу, поруку која је унесена, као и показиваче на комит или комитове који су директно претходили овом комиту (тј. његовог родитеља или родитеље): нула родитеља за почетни комит, једног родитеља за нормални комит, и више родитеља за комит који је резултат спајања две или више грана.

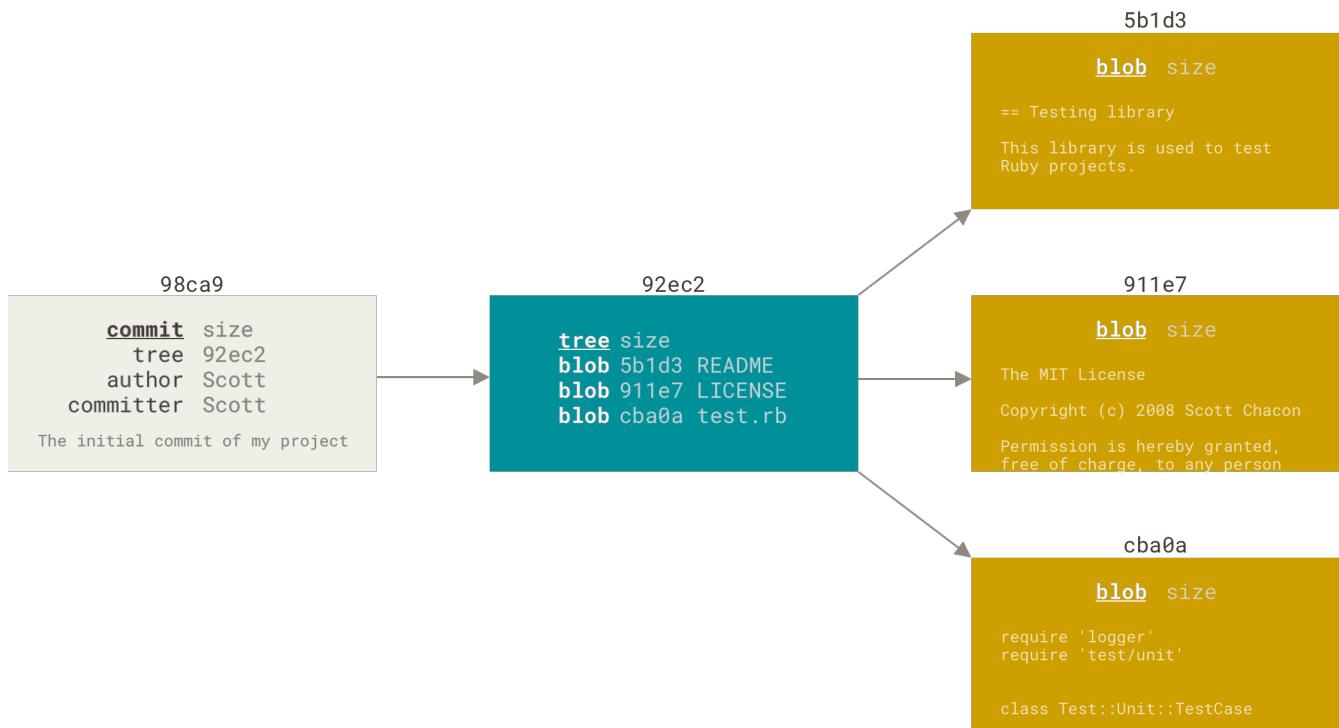
Да бисмо ово сликовито приказали, претпоставимо да имате директоријум који садржи три фајла и да их све стејџујете, а затим комитујете. Стејџовање фајлова рачуна контролну суму сваког од њих (SHA-1 хеш као што смо поменули у [Шта је Гит?](#)), чува ту верзију фајла у Гит репозиторијум (програм Гит то назива *блобовима*) и додаје ту контролну суму на стејџ:

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

Када са `git commit` направите комит, програм Гит прави контролну суму сваког поддиректоријума (у овом случају, само корени директоријум пројекта) и чува их у Гит репозиторијум као стабло. Програм Гит онда креира комит објекат који има метаподатке и показивач на корен стабла пројекта тако да поново може креирати тај снимак онда када буде био потребан.

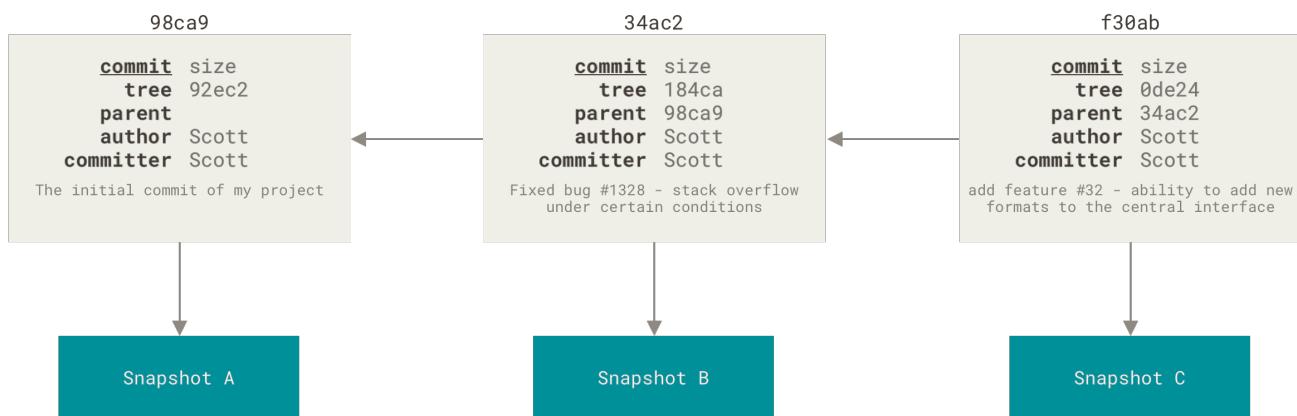
Ваш Гит репозиторијум сада садржи пет објеката: три *блоба* (од којих сваки представља садржај једног од три фајла), једно *стабло* које садржи листу садржаја директоријума и

наводи која имена фајлова се чувају у ком блобу, као и један комит са показивачем на тај корен стабла и све комит метаподатке.



Слика 9. Комит и његово стабло

Ако направите неке измене и комитујете поново, следећи комит чува показивач на комит који је дошао непосредно пре њега.

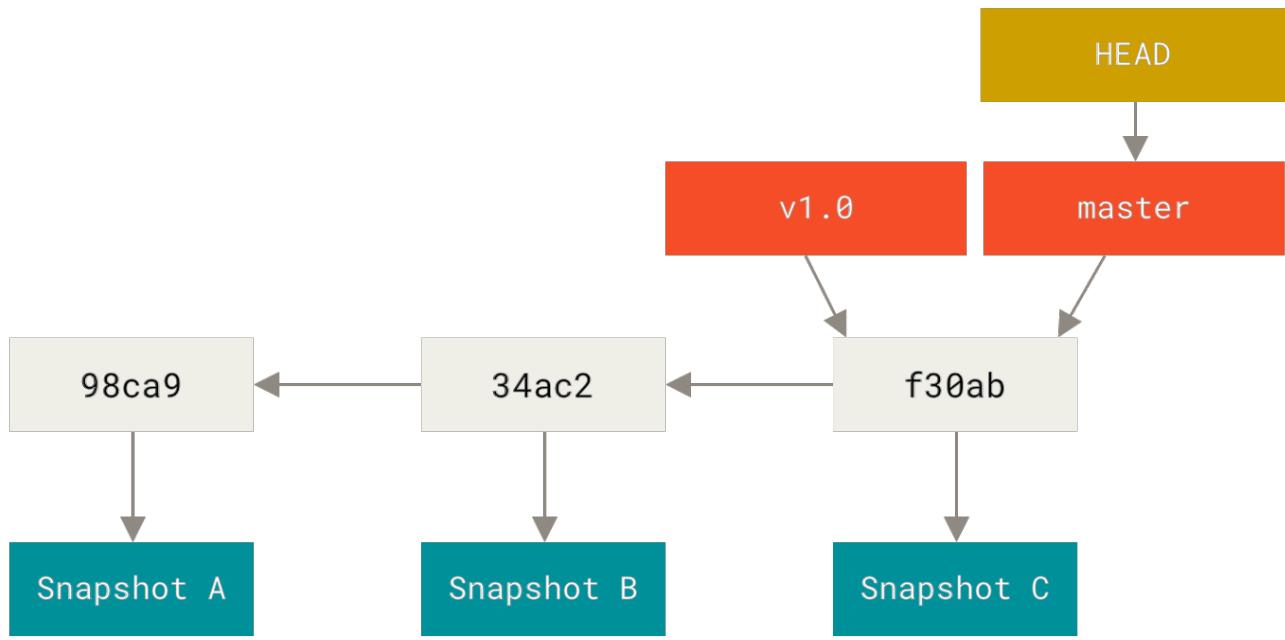


Слика 10. Комитови и њихови родитељи

Грана у програму Гит је просто један мали покретни показивач на неки од ових комитова. Подразумевано име гране у програму Гит је `master`. Када почнете да комитујете, даје вам се `master` грана која показује на последњи комит који сте направили. Сваки пут када комитујете, показивач `master` гране се аутоматски креће унапред.

„master” грана у програму Гит није посебна грана. Она је потпуно иста као и свака друга грана. Једини разлог због којег скоро сваки репозиторијум има такву грану је то што је команда `git init` подразумеваним направи, а већина људи нема потребу да је мења.





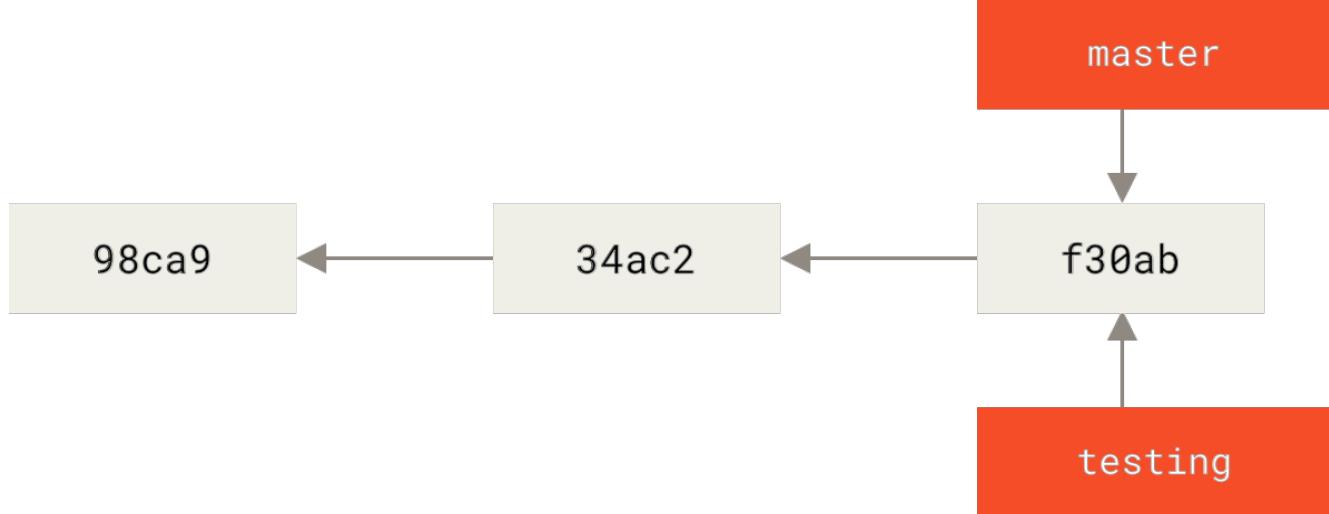
Слика 11. Грана и њена историја комитова

Прављење нове гране

Шта се дешава када направите нову грану? Па, када то урадите, креира се нови показивач којим се крећете унаоколо. Речимо да направите нову грану коју ћете назвати **testing**. То се ради командом **git branch**:

```
$ git branch testing
```

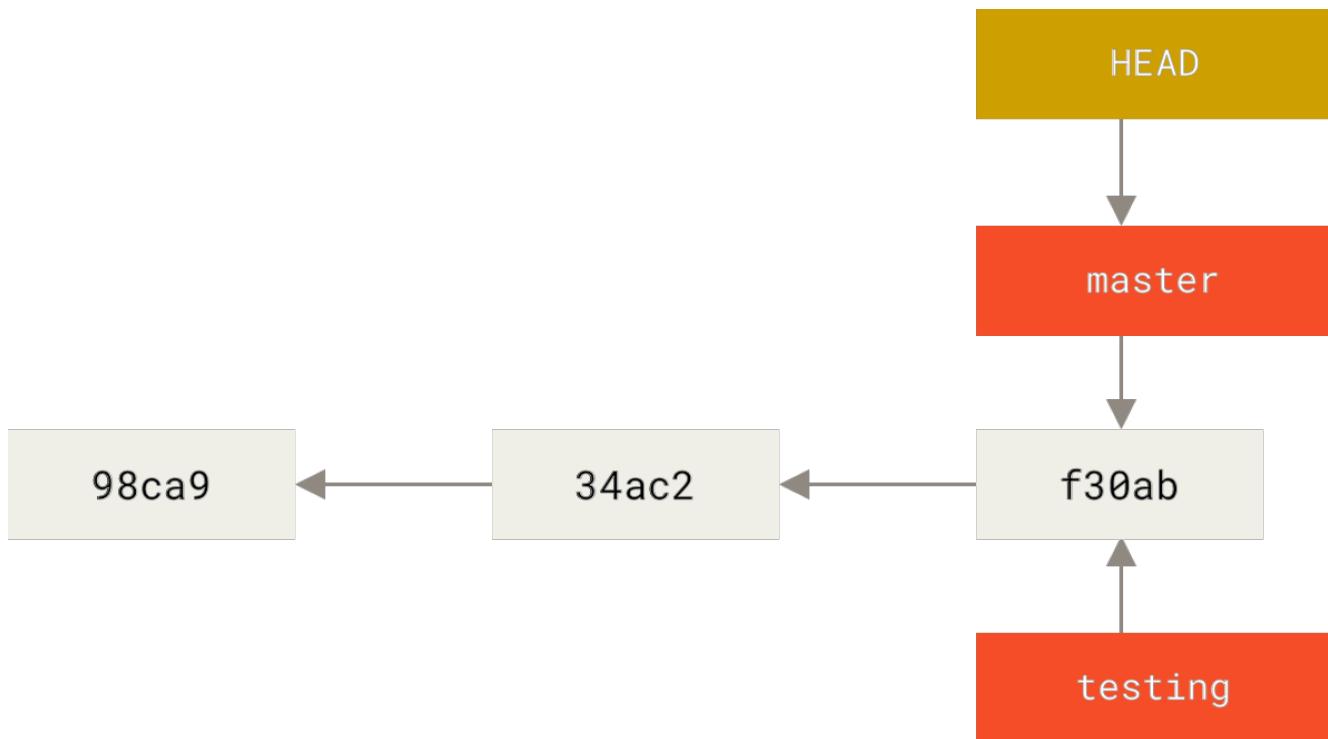
Ово прави нови показивач на исти комит на којем се тренутно налазите.



Слика 12. Две гране које показују на исти низ комитова

Како програм Гит зна на којој грани се тренутно налазите? Он чува посебан показивач који се зове **HEAD**. Обратите пажњу на то да је ово много другачије од **HEAD** концепта у осталим VCS системима на које сте можда навикли, као што су *Subversion* или *CVS*. У програму Гит, ово је показивач на локалну грану на којој се тренутно налазите. У овом случају, још увек сте на

`master` грани. Командом `git branch` сте само направили нову грану — нисте прешли на њу.



Слика 13. `HEAD` показује на грану

Ово лако можете да видите тако што ћете извршити обичну `git log` команду која вам приказује на шта показују показивачи грана. Ова опција се зове `--decorate`.

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

Видите да су `master` и `testing` гране одмах поред комита `f30ab`.

Мењање грана

Када желите да пређете на неку постојећу грану, извршавате команду `git checkout`. Хајде да пређемо на нову грану `testing`:

```
$ git checkout testing
```

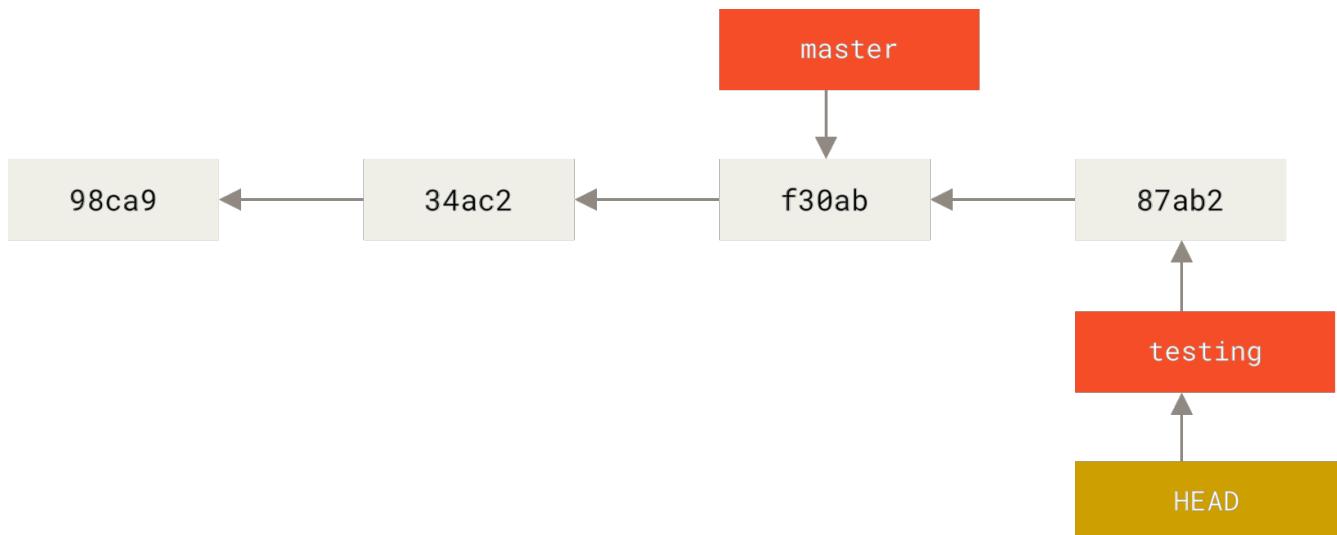
Ово помера показивач `HEAD` тако да показује на грану `testing`.



Слика 14. `HEAD` показује на тренутну грану

Зашто је ово битно? Па, хајде да урадимо још један комит:

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



Слика 15. `HEAD` грана се помера унапред када се направи комит

Ово је занимљиво, јер сада `testing` грана померила унапред, али ваша `master` грана још увек показује на комит на коме сте били када сте извршили `git checkout` да промените гране. Хајде да се вратимо назад на грану `master`:

```
$ git checkout master
```

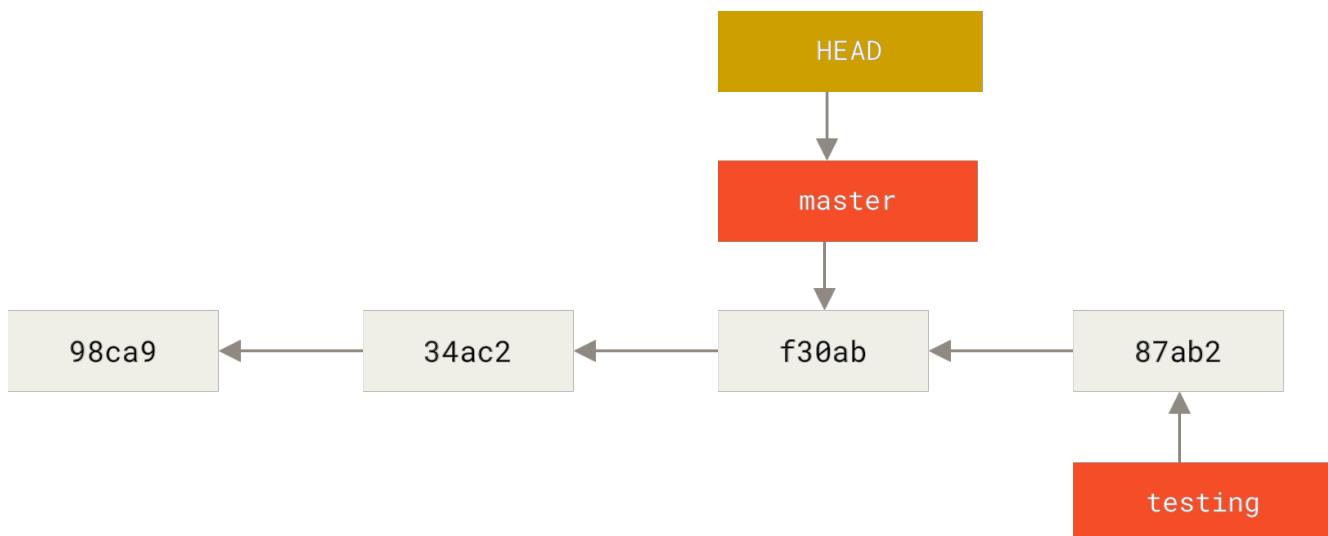
`git log` не приказује све гране све време

Ако бисте сада извршили `git log`, запитали бисте се где нестаде „testing“ грана коју сте управо креирали, јер је нема у излазу команде.



Грана није несталла; програм Git једноставно не зна да вас та грана интересује и покушава да вам прикаже оно што мисли да вас интересује. Другим речима, команда `git log` ће подразумевано да прикаже само историју комитова испод гране коју сте одјавили.

Ако желите да видите историју комитова неке гране, морате експлицитно да је наведете: `git log testing`. Ако желите да видите све гране, додајте `--all` у своју `git log` команду.



Слика 16. `HEAD` се помера када извршиште одјављивање

Ова команда је урадила две ствари. Померила је показивач `HEAD` назад на место у грани `master` и вратила је фајлове у радном директоријуму на снимак на који показује `master`. Ово такође значи и да ће се промене које правите одсад па надаље одвојити од старије верзије пројекта. Команда у суштини премотава уназад, поништавајући рад у `testing` грани, како бисте могли да кренете другим путем.

Мењање грана мења фајлове у радном директоријуму



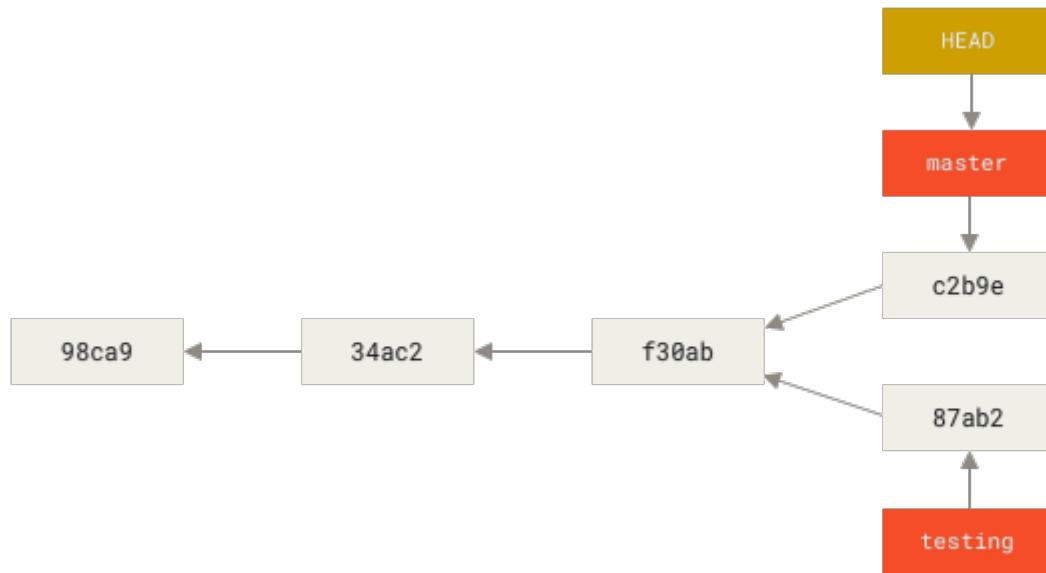
Када у програму Гит прелазите с гране на грану, важно је приметити да ће се фајлови у радном директоријуму променити. Ако се пребаците на старију грану, радни директоријум ће се вратити на изглед који је имао у време када сте комитовали на тој грани. Ако програм Гит није у стању да то уради без проблема, уопште вам неће дозволити се пребаците.

Хајде да направимо неколико промена и поново комитујемо:

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

Сада се историја вашег пројекта развојила (погледајте [Развојена историја](#)). Направили сте

грану, пребацили сте се на њу, урадили нешто у њој, и онда се вратили назад на главну грану и урадили још мало посла. Обе ове промене су изоловане у посебним гранама: можете да скачете с једне на другу напред-назад и да их спојите онда када будете спремни. И све сте то урадили простим командама `branch`, `checkout` и `commit`.



Слика 17. Раздвојена историја

Ово лако можете видети и са `git log` командом. Ако извршите `git log --oneline --decorate --graph --all`, исписаће вам се историја комитова, показујући вам где се сада налазе показивачи на гране и како се историја раздвајала.

```
$ 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
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 Initial commit
```

Пошто је грана у програму Гит заправо једноставан фајл који садржи 40 карактера SHA-1 контролне суме комита на који показује комит, прављење и уништавање грана је јефтино. Креирање нове гране је брзо и једноставно колико и уписивање 41 бајта у фајл (40 карактера и карактер за прелом линије).

Ово је у оштром контрасту са начином на који гранање ради већина старијих VCS алата, који подразумева копирање свих фајлова пројекта у други директоријум. Ово може потрајати неколико секунди или чак минута, у зависности од величине пројекта, док је у програму Гит тај процес увек тренутан. Такође, пошто бележимо родитеље када комитујемо, проналажење одговарајуће базе за спајање се аутоматски одради уместо да се ми бавимо

тиме и у општем случају је тај процес веома једноставан. Ове особине подстичу програмере да често праве и користе гране.

Хајде да погледамо зашто и ви треба то да радите.

Креирање нове гране и истовремени прелаз на њу



Врло је чест случај да желите креирати нову грану и да се одмах пребаците у њу—то можете обавити у једном кораку командом `git checkout -b <именоветране>`.

Почевши од програма Гит верзије 2.23 па надаље уместо `git checkout` можете употребити `git switch` за:



- Пребацивање на постојећу грану: `git switch testing-branch`.
- Креирање нове гране и прелазак на њу: `git switch -c new-branch`. Заставица `-c` стоји уместо `create` (креирај), али можете да употребите и комплетну заставицу: `--create`.
- Повратак на грану коју сте претходно одјавили: `git switch -`.

Основе гранања и спајања

Хајде да прођемо кроз једноставан пример гранања и спајања у процесу рада какав се често јавља у реалном свету. Пратићете ове инструкције:

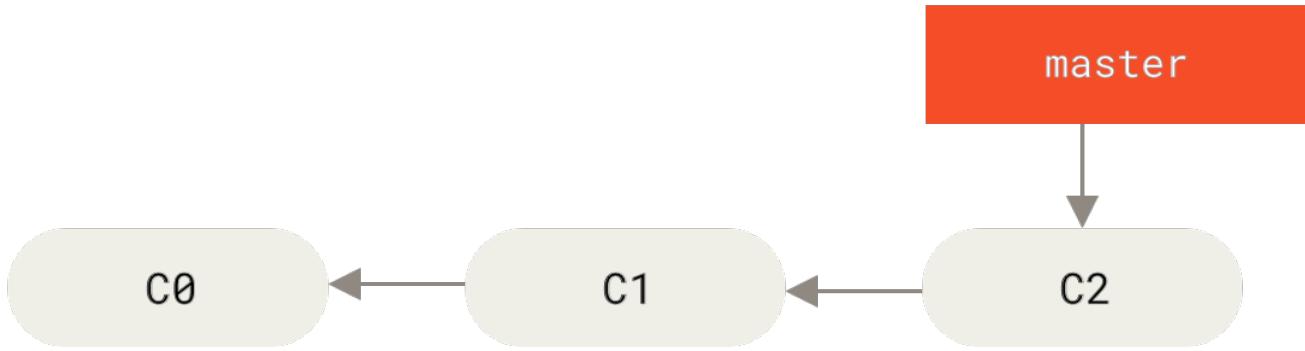
1. Одрадићете неки посао на веб сајту.
2. Направите грану за нову корисничку причу на којој радите.
3. Одрадићете нешто на тој грани.

У овом тренутку, добићете позив да постоји критичан проблем који морате да решите истог тренутка. Урадићете следеће:

1. Пребацићете се на грану за продукцију.
2. Направићете нову грану на којој ћете додати код који решава проблем.
3. Када га тестирате, спојићете грану са решењем проблема, и гурнућете на продукцију.
4. Вратићете се назад на корисничку причу на којој сте радили и наставити посао.

Основе гранања

Најпре, претпоставимо да радите на пројекту у коме на `master` грани већ имате неколико комитова.



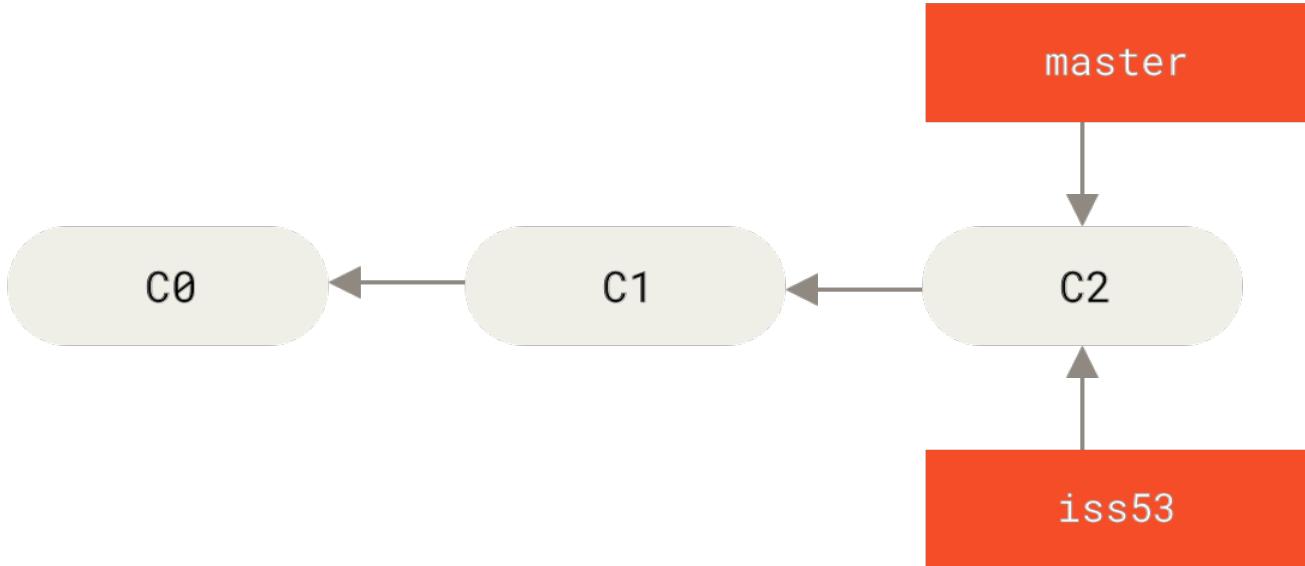
Слика 18. Једноставна историја комитова

Одлучили сте да ћете радити на проблему #53 који се налази на неком систему за праћење проблема који користи ваша компанија. Да бисте направили грану и истовремено скочили на њу, извршите команду `git checkout` са прекидачем `-b`:

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

Ово је скраћеница за:

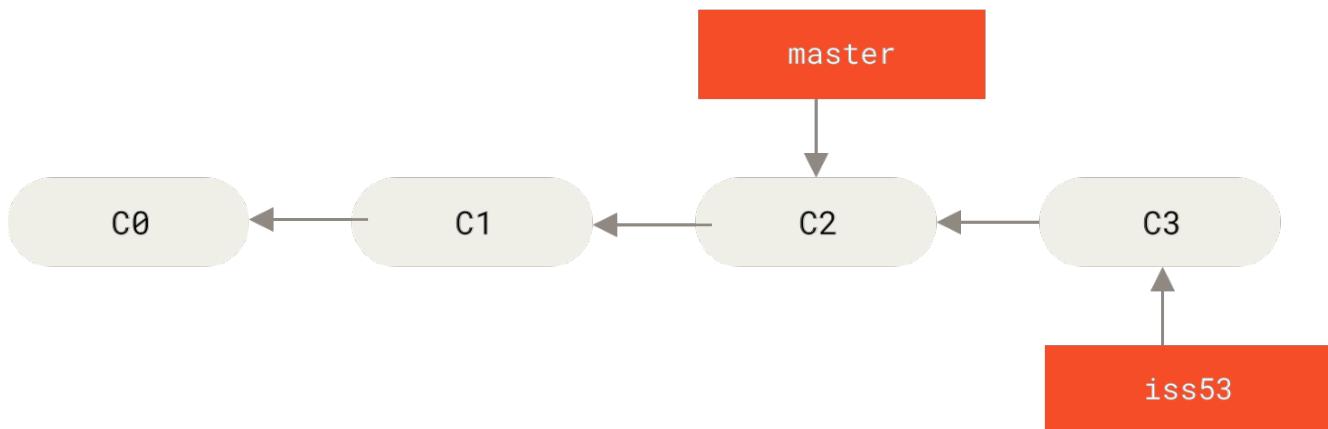
```
$ git branch iss53
$ git checkout iss53
```



Слика 19. Креирање новог показивача на грану

Радите неке ствари на свом веб сајту и вршите неке комитове. Док то радите, грана `iss53` се креће унапред, јер сте је одјавили (односно, `HEAD` показује на њу):

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



Слика 20. Грана `iss53` се померила унапред у складу с послом који сте обавили

Сада добијате позив да постоји проблем са веб сајтом, и морате одмах да га поправите. Са програмом Гит, не морате да решавате проблем заједно са `iss53` променама које сте направили, и не морате да улажете много труда у то да опозовете те промене пре него што почнете рад на примени исправке за оно што је у продукцији. Потребно је само да се пребаците назад на `master` грану.

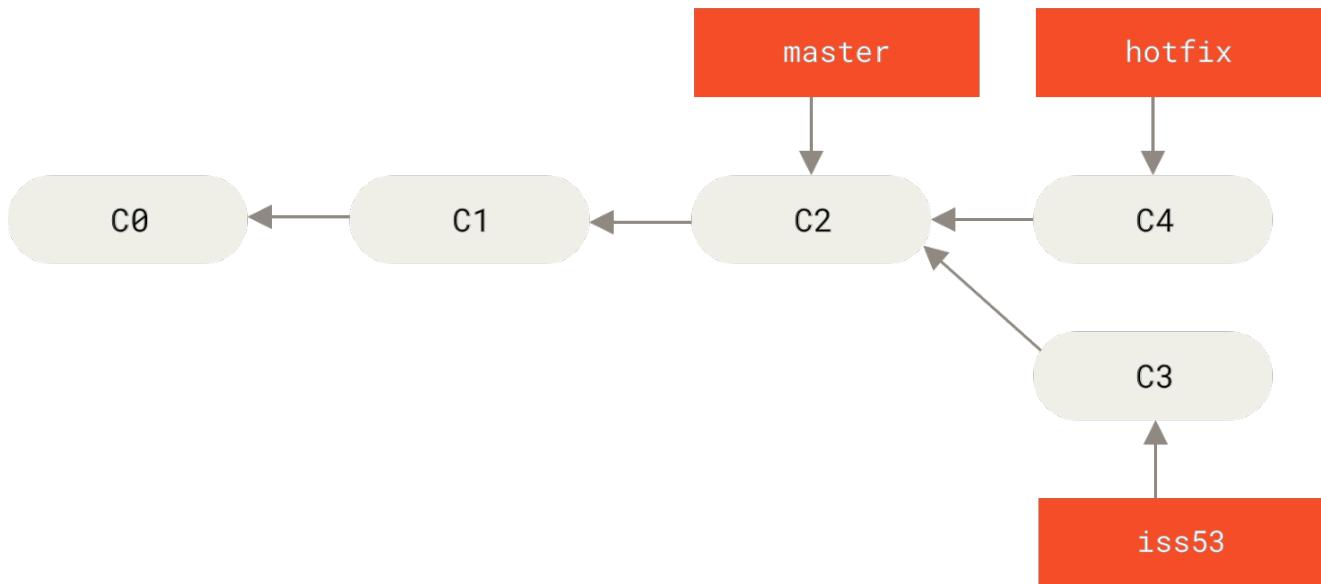
Ипак, пре него што то урадите, обратите пажњу на то да ако ваш радни директоријум или стејџ има некомитоване промене које су у конфликту са граном коју одјављујете, програм Гит вам неће дозволити да промените грану. Најбоље да радно стање буде чисто вршите скок између грана. Постоје начини да се ово заобиђе (наиме, скривање и исправљање комита) које ћемо обрадити касније, у [Скривање и чишћење](#). Засад, претпоставимо да сте комитовали све промене, тако да се безбедно можете вратити на `master` грану:

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

У овом тренутку, радни директоријум вашег пројекта изгледа исто онако како је изгледао пре него што сте почели да радите на проблему #53, па можете да се концентришете на хитни случај. Ово је битна ствар коју треба запамтити: када мењате гране, програм Гит ресетује радни директоријум тако да изгледа онако како је изгледао када сте последњи пут комитовали на тој грана. Аутоматски додаје, брише и мења фајлове тако да ваша радна копија изгледа тачно онако како је изгледала када сте на грани урадили последњи комит.

Следеће, треба да решите хитан проблем. Направићемо `hotfix` грану на којој ћемо радити све док не решимо проблем:

```
$ 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(+)
```



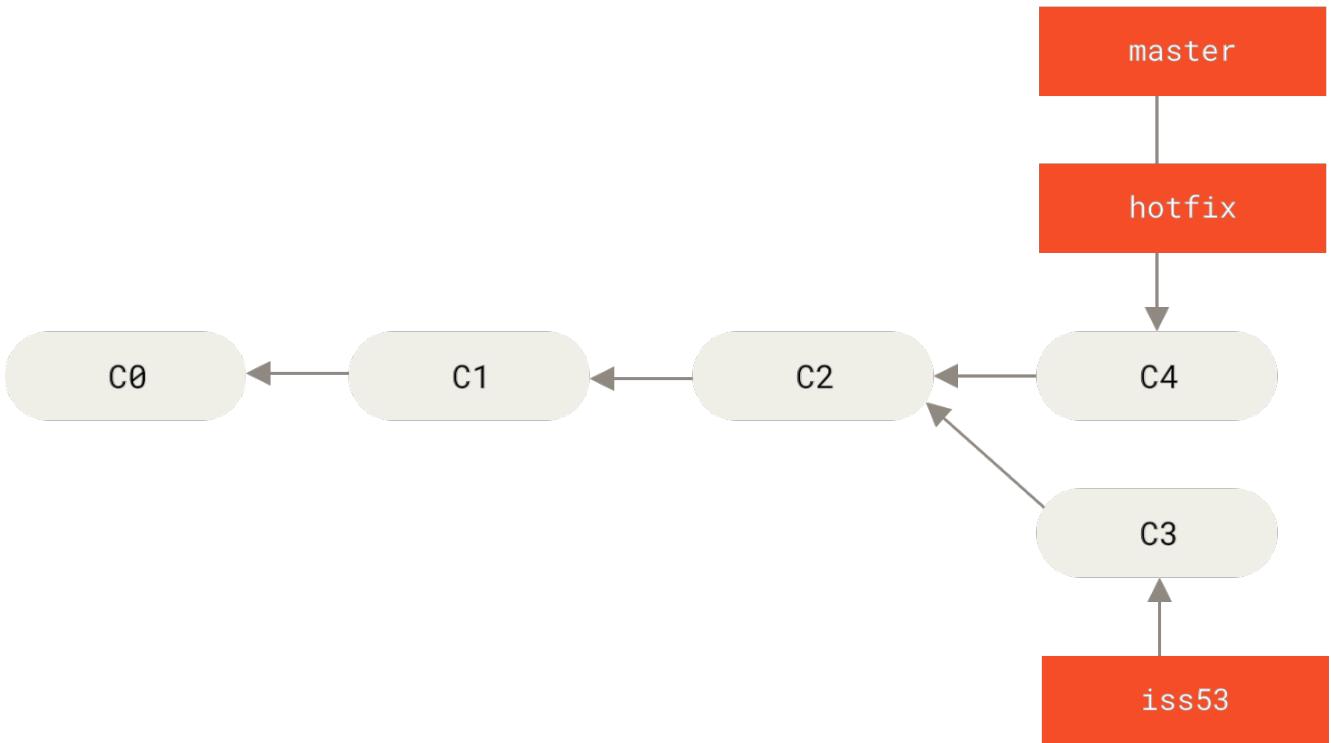
Слика 21. Hotfix грана базирана на master грана

Сада можете да тестирате оно што сте урадили, да будете сигурни да је проблем решен и да грану `hotfix` спојите назад са граном `master`. Ово можете да урадите командом `git merge`.

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

У овом комиту ћете приметити израз „fast-forward” (премотавање унапред). Пошто се комит `C4` на који показује грана `hotfix` коју сте се спојили налазио директно испред комита `C2` на ком се налазите, програм Гит једноставно помера показивач унапред. Да то преформулишемо, када покушате да спојите један комит са комитом до ког се може стићи пратећи историју првог комита, програм Гит поједностављује ствари тако што само помери показивач унапред, јер нема развојеног рада којом би требало се спаја — ово зове премотавање унапред.

Промена се сада налази у снимку комита на који показује `master` грана, па можете да примените исправку.



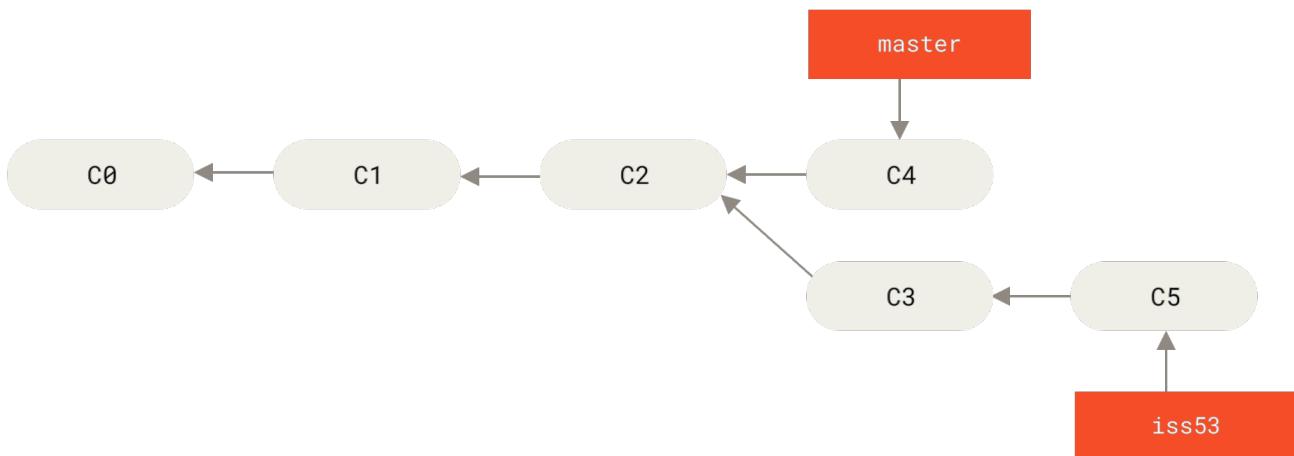
Слика 22. `master` се премотала унапред до `hotfix`

Када се примени ваша супер-важна исправка, време је да се вратите на оно што сте радили пре него што сте били прекинути. Ипак, најпре ћете обрисати `hotfix` грану, јер вам више није потребна—`master` грана показује на исто место. Можете је обрисати помоћу опције `-d` команде `git branch`:

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

Сада можете да се вратите на свој тикет #53 и настављате да радите на њему.

```
$ 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(+)
```



Слика 23. Рад се наставља на iss53

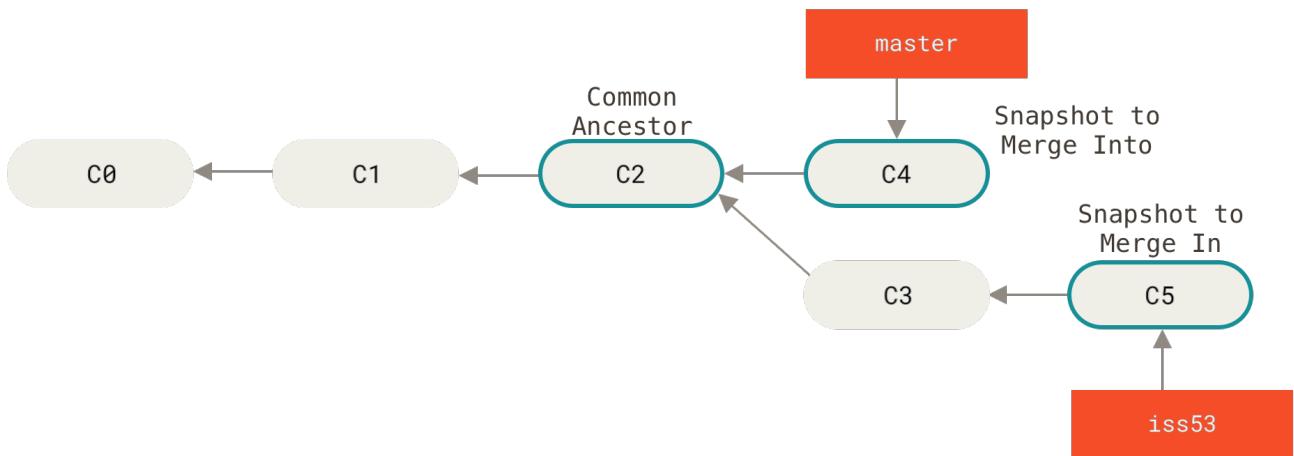
Овде вреди напоменути да се посао који сте одрадили у грани `hotfix` не садржи у фајловима на грани `iss53`. Ако треба да га повучете у њу, можете да спојите грани `master` у грани `iss53` извршавањем команде `git merge master`, или можете да сачекате да интегрисањем тих промена док касније не одлучите да повучете грани `iss53` назад у `master`.

Основе спајања

Претпоставимо да сте одлучили да је ваш рад на проблему #53 готов и да је код спреман за спајање са `master` граном. Да бисте то урадили, спојићете грани `iss53` са граном `master`, као што сте раније спојили `hotfix` грану. Све што треба да урадите јесте да одјавите грани у коју желите да спојите и да онда извршите `git merge` команду:

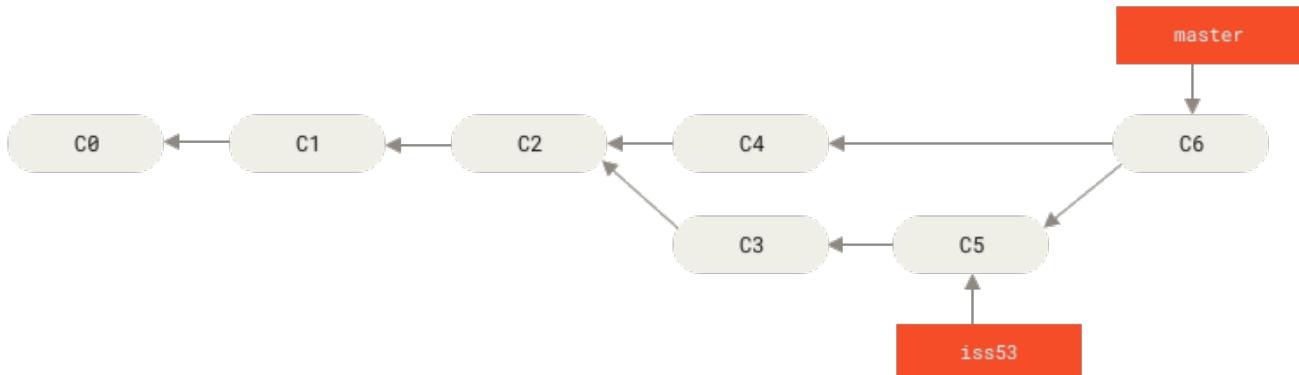
```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Ово изгледа мало другачије од ранијег спајања `hotfix` грани. У овом случају, историја вашег развоја се раздвојила од неке старије тачке. Пошто комит на грани на којој се налазите није директан предак грани у коју спајате, програм Гит мора одради неки посао. У овом случају, програм Гит ради једноставан троструки спој, користећи два снимка на које показују врхови грana и њихов заједнички предак.



Слика 24. Три снимка који се користе у типичном спајању

Уместо да само помери показивач гране унапред, програм Гит прави нови снимак који је резултат овог троструког спајања и аутоматски прави нови комит који показује на њега. Ово се назива комит спајања (*merge commit*) и посебан је једино у том смислу што има више од једног родитеља.



Слика 25. Комит спајања

Сада када је комплетан рад спојен, више нема потребе за `iss53` граном. Можете да затворите тикет у вашем систему за праћење проблема и да обришете грану:

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

Основни конфликти при спајању

Овај процес често неће течи овако глатко. Ако сте у две различите гране које спајате променили исти део истог фајла на различит начин, програм Гит неће моћи лепо да их споји. Ако је ваша исправка за тикет #53 изменила исти део фајла као и `hotfix` грана, добићете конфлікт при спајању који изгледа отприлике овако:

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

Програм Гит није аутоматски направио нови комит спајања. Паузира је процес док ви не решите конфликт. Ако желите да видите који фајлови нису спојени у било ком тренутну након конфликта при спајању, треба да извршите `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")
```

Све што има конфликте при спајању који нису решени наведено је под неспојено (*unmerged*). Програм Гит додаје стандардне маркере за решавање конфликта у фајлове са конфликтима, тако да ручно можете да их отворите и решите конфликте. Фајл садржи секцију која изгледа слично овоме:

```
<<<<< 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
```

Ово значи да је верзија у **HEAD** (ваша **master** грана, јер сте то одјавили када сте покренули команду **merge**) на врху тог блока (све изнад **=====**), док је верзија из гране **iss53** приказана у доњем делу. Да бисте решили конфлікт, морате или да изаберете једну или другу страну, или да ручно спојите садржину фајла. На пример, овај конфлікт се може решити тако што ћете цео горњи блок заменити следећим:

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

Ово решење има помало текста из обе секције, а линије **<<<<<**, **=====** и **>>>>>** су

комплетно уклоњене. Након што решите сваку од оваквих секција у сваком фајлу са конфликтом, извршите `git add` за сваки фајл чиме означавате да је разрешен. У програму Гит, стејџовање фајла означава да је конфликт разрешен.

Ако желите да користите графички алат за решавање оваквих проблема, можете да извршите команду `git mergetool`, која покреће одговарајући визуелни алат и води вас кроз конфликте:

```
$ 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 хоћете да користите неки други алат за спајање који није подразумевани (у овом случају је програм Гит је изабрао `opendiff` јер је команда била покренута на Меку), можете видети све подржане алате наведене при врху након „one of the following tools”. Једноставно укуцајте име алата који бисте радије користили.



Ако су вам потребни напреднији алати за решавање компликованих конфликтова при спајању, погледаћемо неке од њих у [Напредно спајање](#).

Када изађете из алата за решавање конфликтова при спајању, програм Гит ће вас питати да ли је спајање било успешно. Ако скрипти одговорите да јесте, она ће стејџовати фајл и уместо вас га означити као разрешен. Можете да извршите `git status` и тако се уверите да су сви конфликти 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 сте задовољни тиме и ако сте потврдили да је све што је имало конфликте сада на стејџу, можете да откуцате `git commit` чиме завршавате комит спајања. Подразумевана

комит порука изгледа слично овако:

```
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
#
```

Можете да измените ову комит поруку додавањем детаља о томе како сте разрешили спој ако мислите да ће то бити корисно другима који буду гледали ово спајање у будућности—зашто сте урадили то што сте урадили, ако није очигледно.

Управљање гранама

Сада када сте направили, спојили и обрисали неке гране, хајде да погледамо неке алате за управљање гранама који ће вам бити од користи када почнете редовно да користите гране.

Поред тога што може да ствара и брише гране, команда `git branch` има и друге намене. Ако је покренете без опција, добићете једноставну листу ваших тренутних грана:

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

Обратите пажњу на карактер `*` који се јавља испред `master` гране: он означава грану која је тренутно одјављена (тј. грана на коју показује показивач `HEAD`). Ово значи да ако сада комитујете, грана `master` ће бити померена унапред новим стварима које сте направили. Да бисте видели последњи комит на свакој грани, можете извршити `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
```

Корисне `--merged` и `--no-merged` опције могу да филтрирају ову листу на гране које већ јесте или још увек нисте спојили у грану на којој се тренутно налазите. Да бисте видели које гране су већ спојене у грану на којој сте сада, треба да извршите `git branch --merged`:

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

Пошто сте `iss53` већ раније спојили, та грана се налази на листи. Гране које на овој листи немају `*` испред себе су у општем случају спремне да буду обрисане командом `git branch -d`; рад у њима сте већ укључили у неку другу грану, тако да ништа нећете изгубити.

Да бисте видели све гране које садрже рад који још увек нисте спојили, извршите команду `git branch --no-merged`:

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

Ово приказује вашу другу грану. Пошто се у њој налази рад који још увек није спојен, покушај брисања ове гране са `git branch -d` неће успети:

```
$ 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 заиста желите да обришете ту грану и изгубите сав тај део кода, брисање можете да форсирате са `-D`, као што то наводи корисна ` порука.

Опције које су описане изнад, `--merged` и `--no-merged` ће вам, у случају да се као аргумент не наведе комит или име гране, приказати шта је спојено или није спојено у вашу *текућу* грану.

Увек можете да наведете додатни аргумент којим питате за стање спајања у односу на неку другу грану без потребе да је прво одјављујете, као у случају питања шта није спојено у `master` грану?

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

Промена имена гране



Немојте да мењате име гранама које још увек користе остали сарадници. Не мењајте име грани као што је `master/main/mainline` пре него што прочитате одељак „Промена имена `master` гране”.

Претпоставимо да имате грану под именом `bad-branch-name` и желите да јој промените име у `corrected-branch-name`, уз задржавање комплетне историје. Име такође желите да промените и на удаљеном серверу (GitHub, GitLab, неки други сервер). Како ово може да се уради?

Грани локално мењате име командом `git branch --move`:

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

Ово замењује вашу `bad-branch-name` са `corrected-branch-name`, али та промена је за сада само локална. Ако желите да и остали виде исправљено име гране на удаљеном серверу, турните је:

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

Сада ћемо на кратко погледати где се налазимо:

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

Приметите да се налазите на грани `corrected-branch-name` и да је она доступна на удаљеном серверу. Међутим, тамо је још увек присутна и грана са погрешним именом, али је можете

обрисати ако извршите следећу команду:

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

Сада је погрешно име гране потпуно замењено исправним именом.

Промена имена master гране



Променом имена гране као што је master/main/mainline/default престају да раде интеграције, сервиси, помоћни алати и скрипте за изградњу/објављивање које користи ваш репозиторијум. Пре него што то урадите, обавезно се консултујте са својим сарадницима. Такође, будите сигурни да сте обавили детаљну претрагу свог репозиторијума и ажурирали у коду и скриптама евентуалне референце на старо име гране.

Следећом командом мењате име локалној `master` грани у `main`:

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

Више нема `master` локалне гране, јер је њено име промењено у `main`.

Ако желите да остали виде нову `main` грану, морате да је гурнете на удаљени сервер. Тако грана са промењеним именом постаје доступна на удаљеном серверу.

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

Сада завршавамо у следећем стању:

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

Ваше локалне `master` гране више нема, јер је заменила `main` грана. `main` грана је присутна на удаљеном серверу. Међутим, стара `master` грана је још увек присутна на удаљеном серверу. Остали сарадници ће наставити да користе `master` грану као базу за свој рад, све док ви не урадите још неке промене.

Сада се пред вами налази још неколико задатака пре него што довршите транзицију:

- Сваки пројекат који зависи од овог ће морати да ажурира свој кôд и/или конфигурацију.
- Ажурирање свих конфигурационих фајлова за покретач тестова (ако постоје).
- Дотерирање скрипти за изградњу и објављивање.

- Преусмеравање подешавања вашег хоста репозиторијума ствари као што су подразумевана грана репозиторијума, правила спајања и све остало што зависи од имена грана.
- Ажурирање референци на старе гране у документацији.
- Затварање или спајање захтева за повлачење који циљају стару грану (ако постоје).

Када завршите све ове задатке и уверите се да `main` грана функционише исто као и `master` грана, безбедно ћете моћи да обришете `master` грану:

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

Процеси рада са гранањем

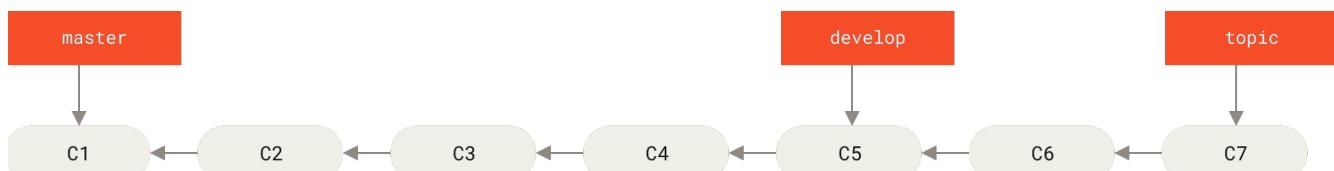
Сада када знате основе о гранама и њиховом спајању, шта можете и шта треба да радите са њима? У овом одељку ћемо се позабавити неким уобичајеним процесима рада које омогућава овакав прост начин гранања у програму Гит, тако да можете одлучити желите ли да их укључите у сопствени развојни циклус.

Дуготрајне гране

Пошто програм Гит користи просто троструко спајање, у општем случају спајање једне гране у другу више пута током дужег временског периода не представља неки проблем. То значи да можете имати неколико грана које су увек отворене и које користите за различите етапе развојног циклуса; можете редовно да спајате неке од њих у остале.

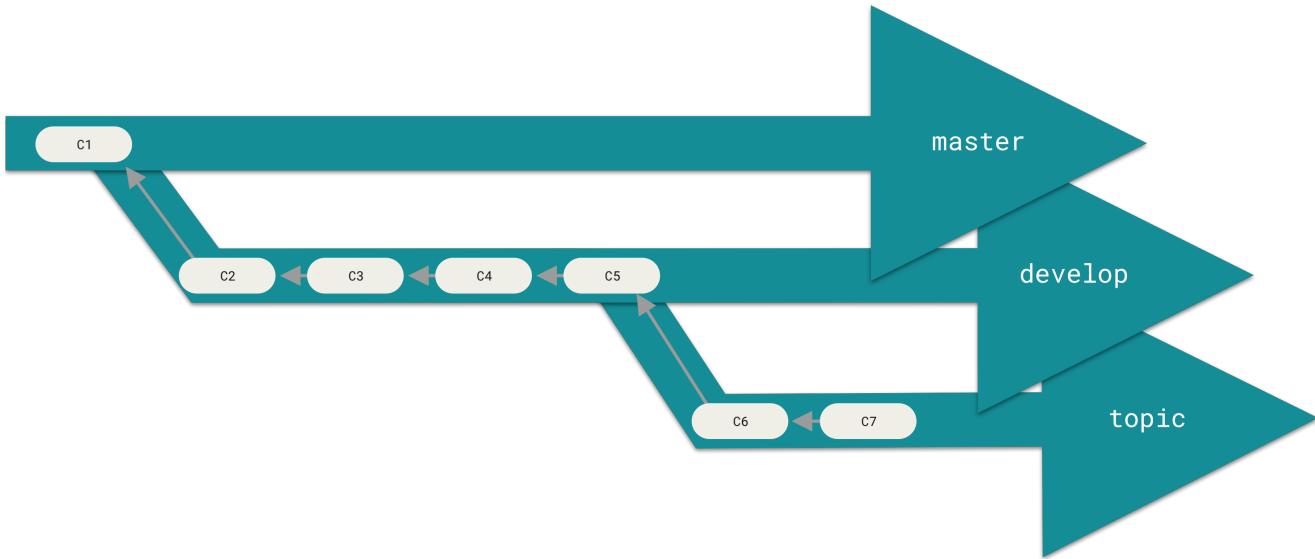
Многи Гит програмери користе овакав начин рада, односно имају код који је у потпуности стабилан у `master` грани — то је вероватно и једини код који је раније био објављен, или који ће се уопште и објављивати у будућности. Имају остале паралелне гране које се зову `develop` или `next` у којима раде или које користе за тестирање стабилности — тако код не мора увек да буде стабилан, али кадгод дође у стабилно стање, може да се споји са `master` граном. Користе се да се у њих повуку тематске гране (које немају дуги животни век попут оне `iss53` коју смо користили раније) када буду спремне, како би се потврдило да пролазе све неопходне тестове и да не уводе нове багове.

У стварности, овде се ради о показивачима који се померају дуж линије комитова које правите. Стабилне гране се налазе даље низ линију ваше историје комитова, гране најновијег развоја су даље при врху историје.



Слика 26. Линеарни поглед на гранање које се ослања на технику прогресивне стабилности

У општем случају је једноставније размишљати о њима као о радним силосима, где скупови комитова сазревају ка стабилнијем силосу онда када се у потпуности тестирају.



Слика 27. „Силос“ поглед на гранање које се ослања на технику прогресивне стабилности

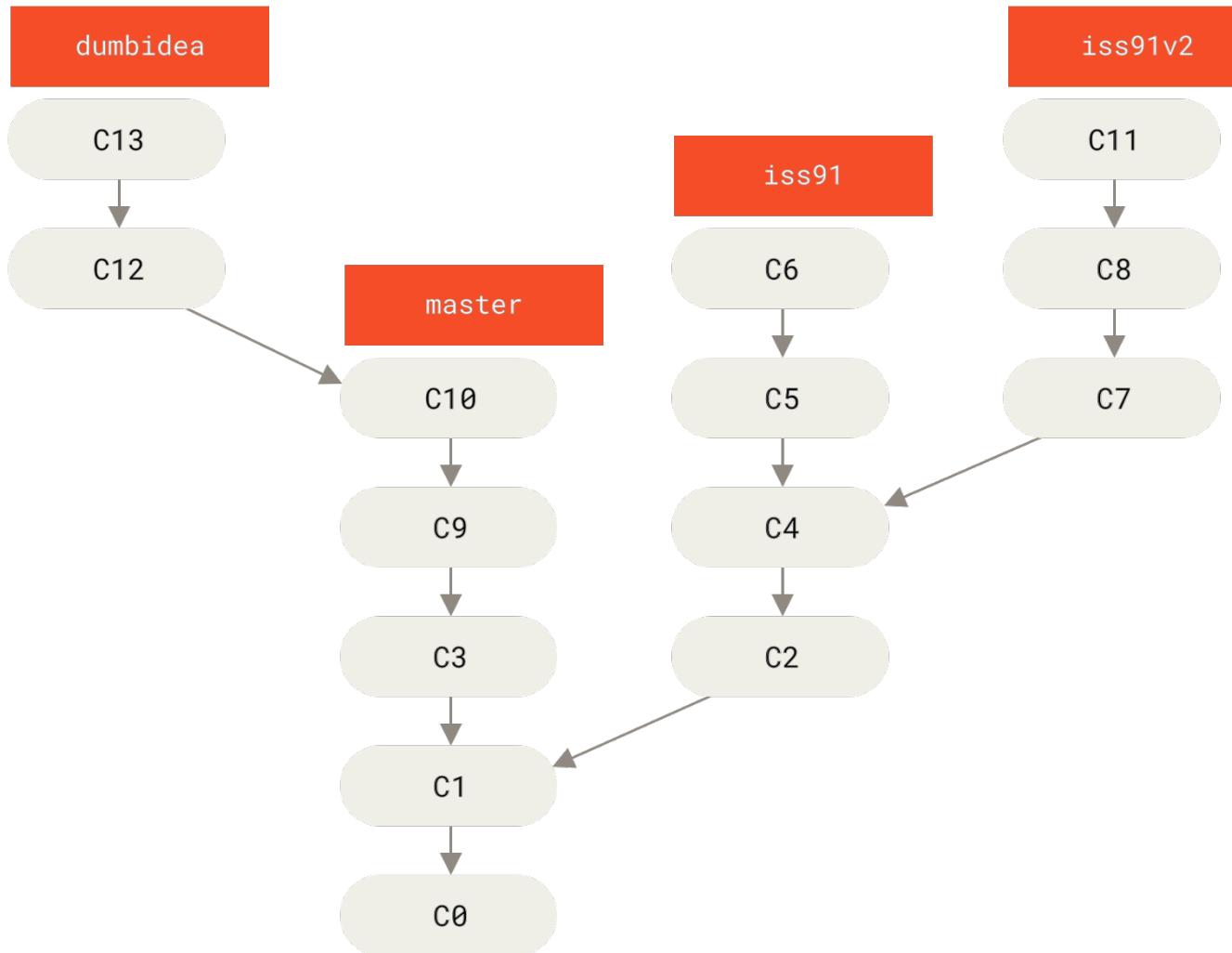
Ово можете наставити да радите за неколико нивоа стабилности. Неки већи пројекти имају и грану **proposed** или **pu** (*proposed updates*) која интегрише гране које још увек нису спремне да се преместе у **next** или **master** грану. Идеја је да се гране налазе на разним нивоима стабилности; када досегну стабилнији ниво, спајају се у грану изнад њих. Ипак, постојање више дуготрајних грана није неопходно, мада је често од користи, поготово када се ради о веома великим или сложеним пројектима.

Тематске гране

Тематске гране су, међутим, корисне за пројекте било које величине. Тематска грана је грана кратког животног века коју креирате и користите само за једну одређену могућност или рад који је у вези са нечим. Ово је нешто што вероватно никад раније нисте радили са VCS системом зато што је креирање и управљање гранама у општем случају превише скupo. Али у програму Гит је уобичајено креирање грана, рад у њима, спајање и брисање грана и неколико пута током дана.

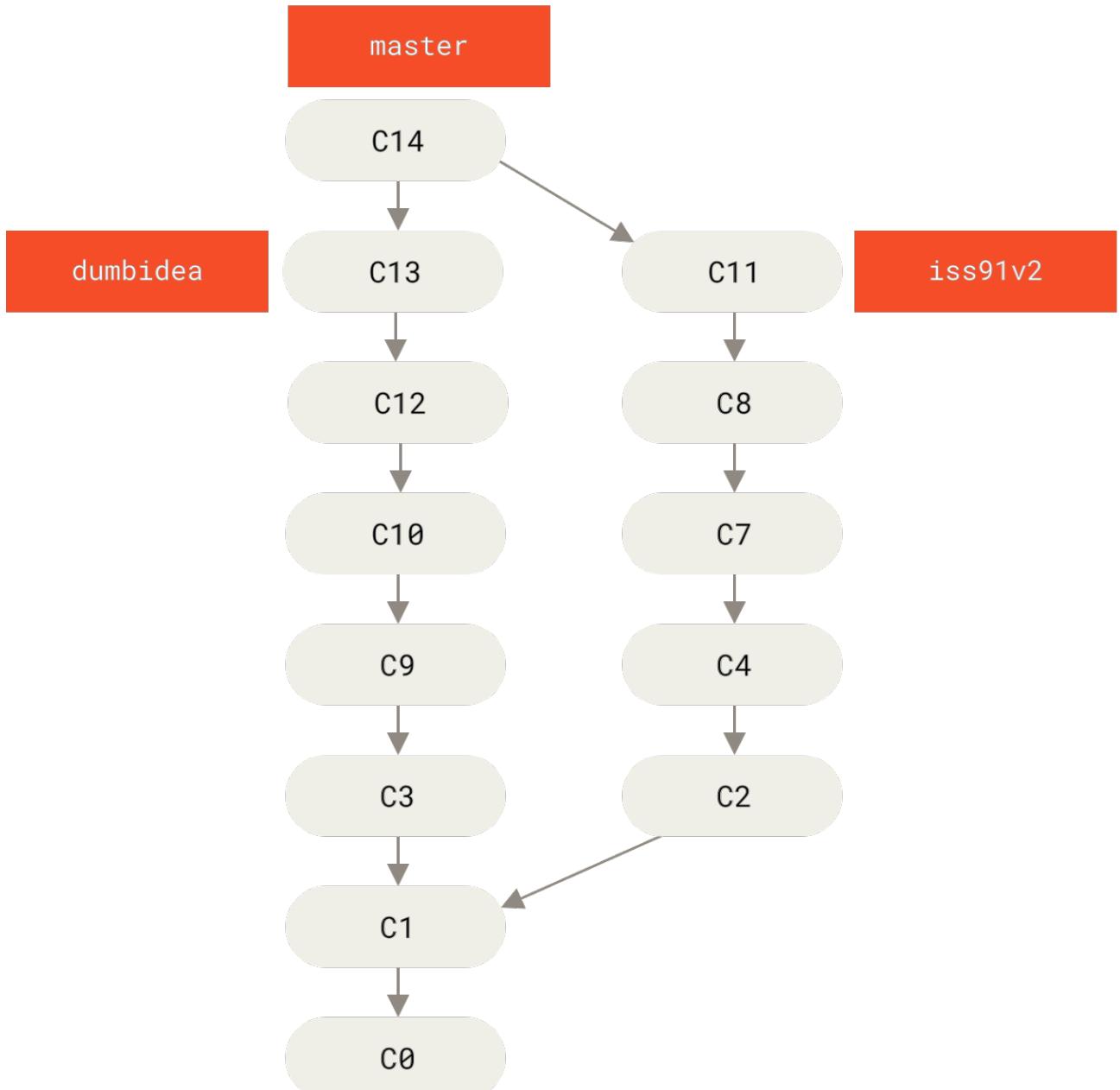
Видели сте ово у преходном одељку са **iss53** и **hotfix** гранама које сте направили. Одрадили сте неколико комитова у њима и обрисали сте их непосредно након спајања у главну грану. Ова техника омогућава да брзо и потпуно мењате контекст у коме радите — пошто је рад подељен у силосе где све промене у једној грани имају везе само са том темом, приликом каснијег прегледа је лакше видети шта се десило са кодом. Промене можете ту да задржите минутима, данима или месецима, па их спојите када буду спремне, без обзира на редослед у којем су креиране и у којем је рађено на њима.

Размотрите пример где радите неки посао (у **master** грани), разгранате се на тикет (**iss91**), неко време радите на томе, разгранате још једну грану у којој пробате да решите исту ствар на други начин (**iss91v2**), враћате се назад на **master** грану у којој радите још неко време, а онда одатле правите грану у којој радите неке ствари за које нисте сигурни да су добра идеја (грана **dumbidea**). Ваша историја комитова ће изгледати овако некако:



Слика 28. Више тематских грана

Рецимо да сада одлучите да вам се друго решење проблема више свића (`iss91v2`), а када својим сарадницима покажете `dumbidea` грану, испостави се да је идеја заправо била генијална. Можете да одбаците оригиналну `iss91` грану (чиме губите комитове `c5` и `c6`) и спајате друге две. Историја сада изгледа овако:



Слика 29. Историја након спајања грана `dumbidea` и `iss91v2`

Детаљније ћемо обрадити остале могуће процесе рада вашег Гит пројекта у [Дистрибуирани Гит](#), тако да обавезно прочитајте и то поглавље пре него што се одлучите какву шему гранања ћете користити у свом наредном пројекту.

Битно је запамтити да су гране потпуно локалне док радите све ово. Док правите гранате и спајате их, све што радите се дешава само у вашем Гит репозиторијуму - не постоји никакав вид комуникације са сервером.

Удаљене гране

Удаљене гране су референце (показивачи) у вашим удаљеним репозиторијумима, укључујући гране, ознаке и тако даље. Комплетну листу удаљених референци можете експлицитно добити командом `git ls-remote <име_удаљеног>`, или `git remote show <име_удаљеног>` за удаљене гране заједно са још више информација. Ипак, много чешћи начин је

да се искористи предност удаљених праћених грана (*remote-tracking branches*).

Удаљене праћене гране су референце на стање удаљених грана. Оне су локалне референце које не можете да померите; програм Гит их аутоматски помера када радите било какав вид комуникације преко мреже, тако да обезбеди да прецизно представљају стање удаљеног репозиторијума. Можете да их посматрате као маркере који треба да вас подсете где су се гране у вашим удаљеним репозиторијумима налазиле када сте се последњи пут повезали са њима.

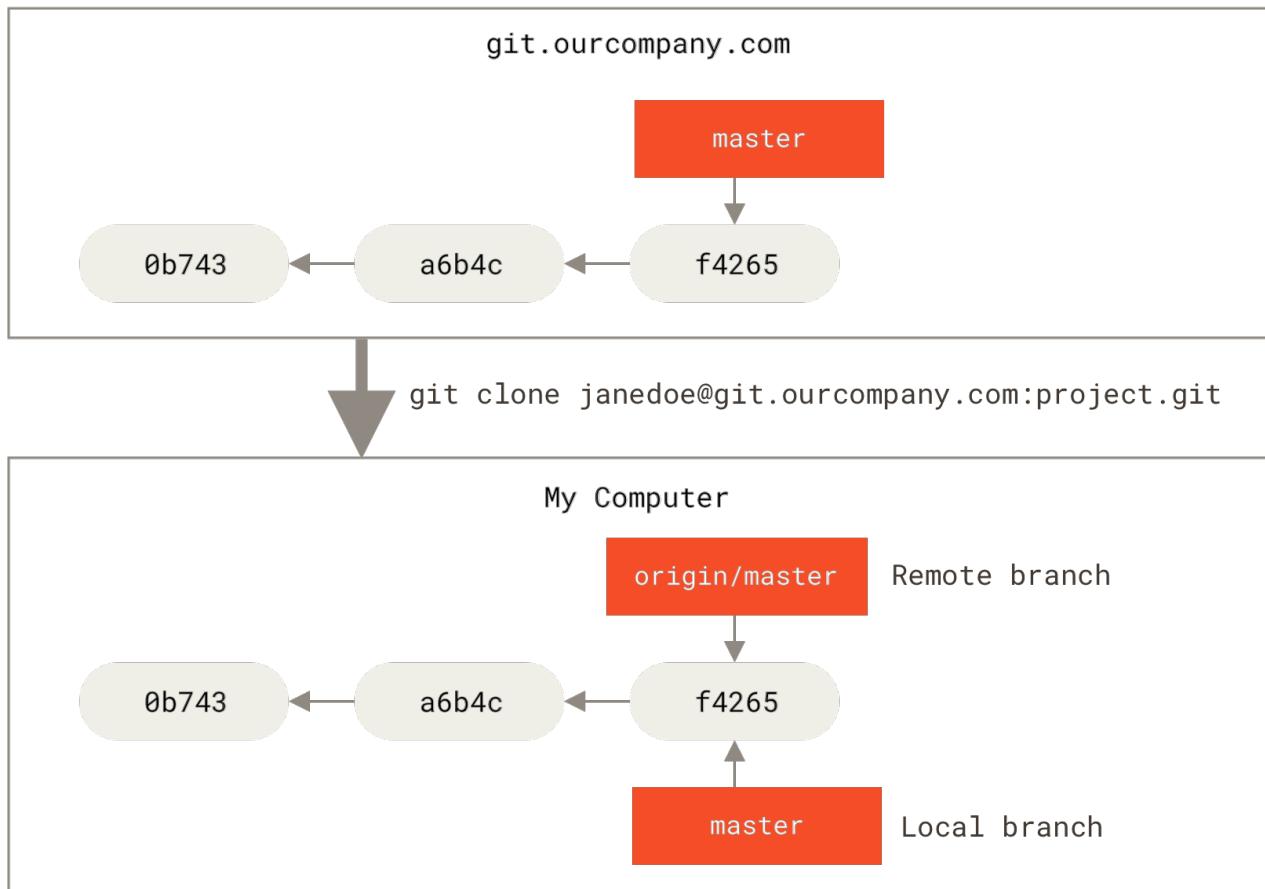
Имена удаљених праћених грана имају облик `<име_удаљеног>/<грана>`. На пример, ако желите да погледате како је `master` грана на вашем `origin` удаљеном репозиторијуму изгледала последњи пут када сте комуницирали са њим, проверили бисте `origin/master` грану. Ако сте радили на тикету са партнерима који су турали измене у `iss53` грану, ви вероватно имате своју локалну `iss53` грану; а на грану која се налази на серверу би показивала удаљена праћена грана `origin/iss53`.

Ово је може бити помало збуњујуће, па хаде да погледамо пример. Речимо да на својој мрежи имате Гит сервер на адреси `git.ourcompany.com`. Ако са њега клонирате, команда `clone` програма Гит ће то аутоматски назвати `origin`, повући ће све податке са њега, направиће показивач на место где се налази `master` грана и локално ће је назвати `origin/master`. Програм Гит вам такође даје властиту локалну `master` грану која почиње на истом месту као и `master` грана на `origin` репозиторијуму, тако да имате одакле да кренете са радом.

`origin` није ништа посебно

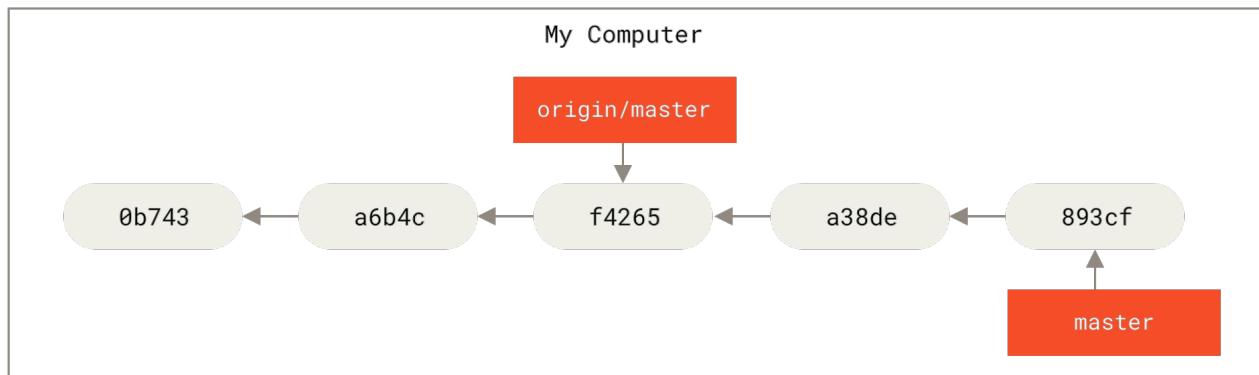
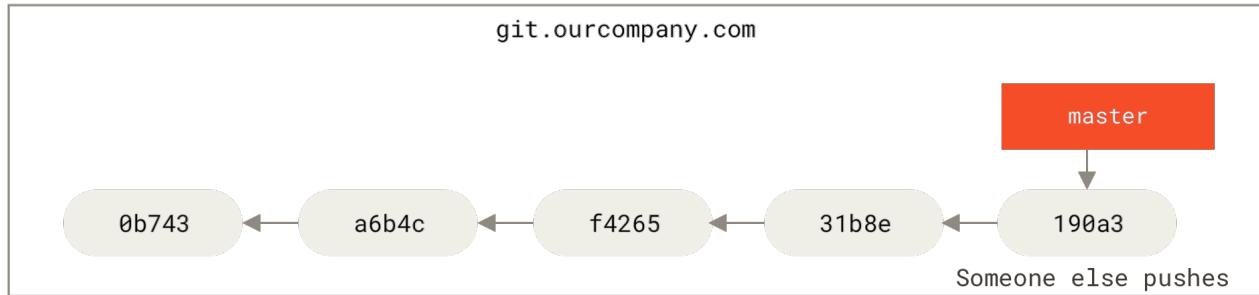


Као што грана са именом „`master`” у програму Гит нема никакво посебно значење, тако нема ни „`origin`”. Док је „`master`” подразумевано име за почетну грану која се добија након команде `git init`, па је то и једини разлог због кога се тако распрострањено користи, „`origin`” је подразумевано име за удаљени репозиторијум када извршите `git clone`. Ако уместо тога извршите `git clone -o booyah`, онда ће `booyah/master` бити подразумевана удаљена грана.



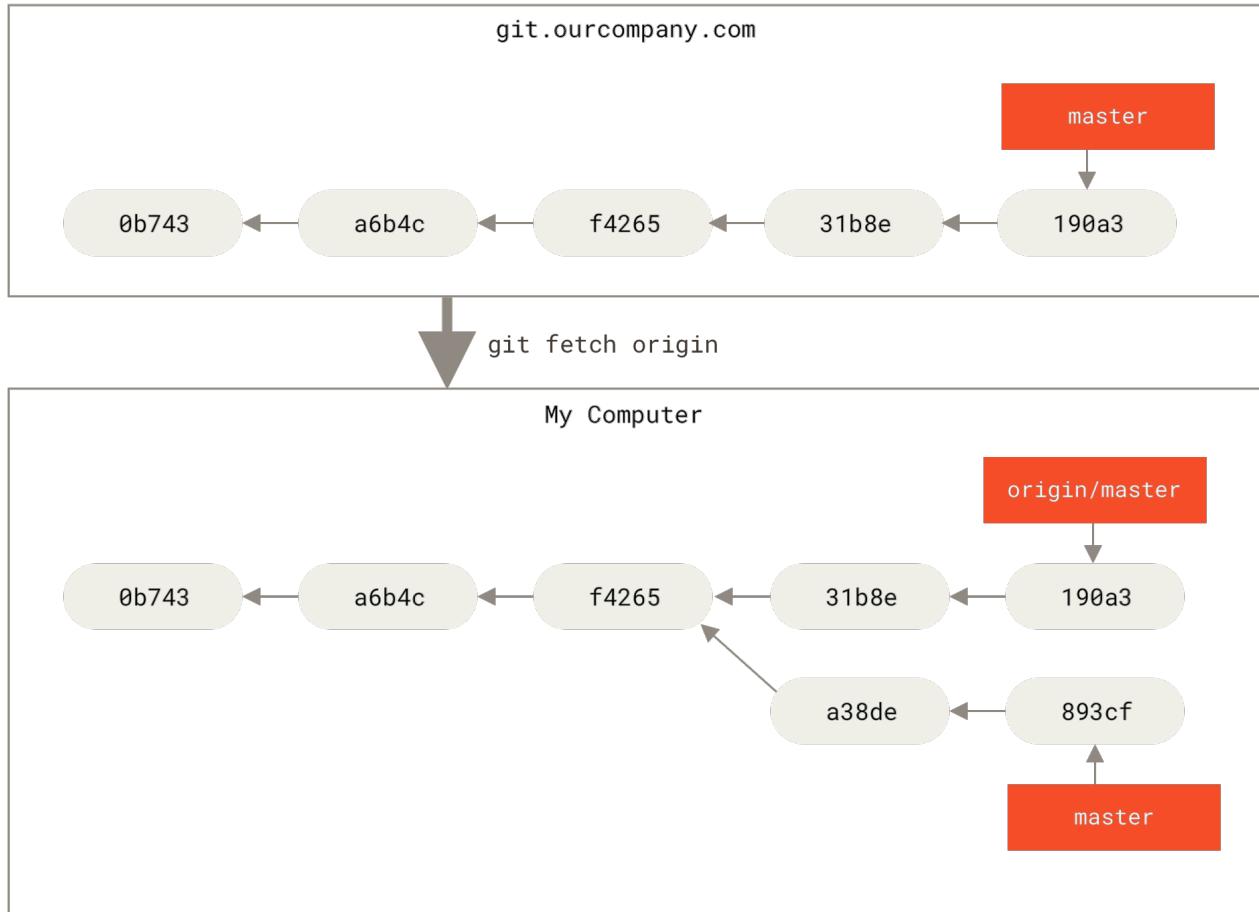
Слика 30. Сервер и локални репозиторијуми после клонирања

Ако одрадите нешто на локалној `master` грани, па у међувремену неко други гурне нешто на `git.ourcompany.com` и ажурира `master` грани на серверу, онда се ваше историје крећу унапред другачијим током. Такође, све док не ступите у контакт са сервером, `origin/master` показивач се не помера.



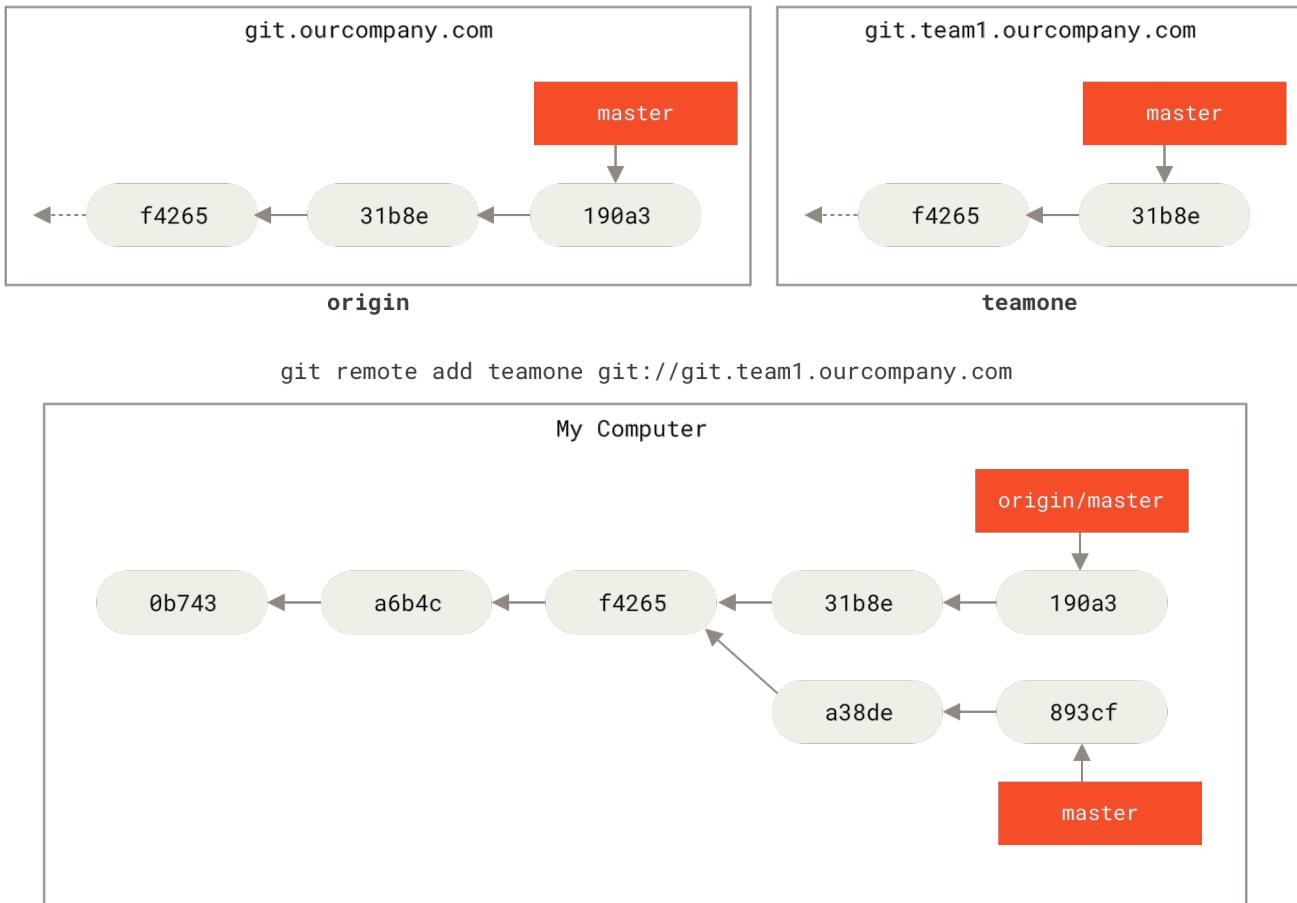
Слика 31. Локални и удаљени рад могу да се разиђу

Да бисте синхронизовали ваш рад са датим удаљеним репозиторијумом, треба да извршите команду `git fetch <име_удаљеног>`. Ова команда ће да потражи сервер који је подешен као `origin` (у овом случају је то `git.ourcompany.com`), преузме са њега све податке које још увек немате, и ажурира вашу локалну базу података, померајући `origin/master` показивач на нову, актуелну позицију.



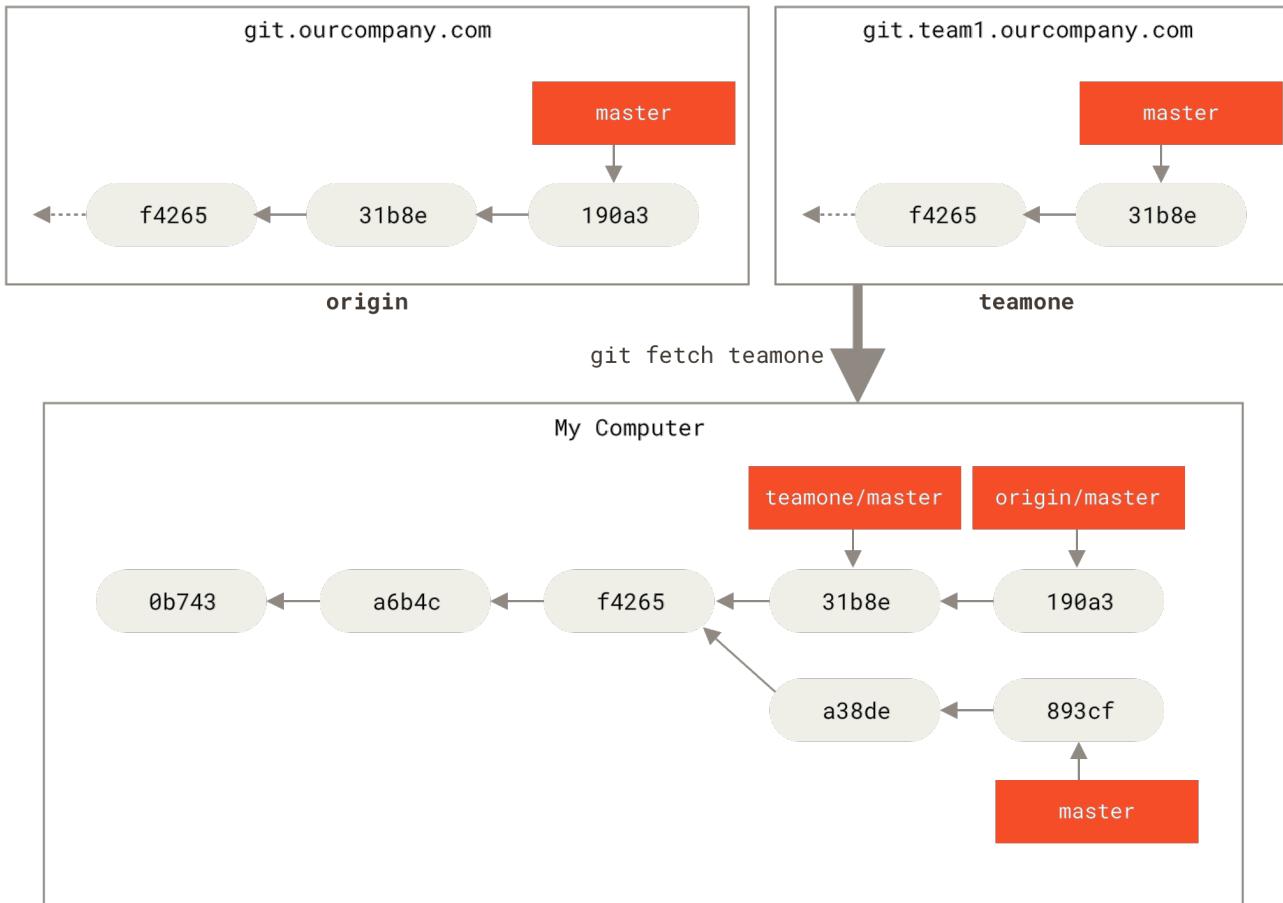
Слика 32. `git fetch` ажурира важе удаљене праћене гране

Да бисмо показали ситуацију када имате неколико удаљених сервера и објаснили како изгледају удаљене гране за те удаљене пројекте, хајде да претпоставимо да имате још један интерни Гит сервер који за развој користи само један од ваших спринг тимова. Овај сервер се налази на адреси `git.team1.ourcompany.com`. Можете да га додате као нову удаљену референцу у пројекат на ком тренутно радите тако што ћете извршити команду `git remote add` као што смо већ објаснили у [Основе програма Гит](#). Ову удаљену грану назовите `teamone`, што ће представљати кратко име за тај цео URL.



Слика 33. Додање још једног удаљеног сервера

Сада можете да извршите `git fetch teamone` и преузмете све што се налази на удаљеном серверу `teamone`, а ви још увек немате. Пошто тај сервер има подскуп података који се тренутно налазе на вашем `origin` серверу, програм Гит не уопште не преузима податке већ поставља удаљену праћену грану под именом `teamone/master` која показује на комит који `teamone` има као своју `master` грану.



Слика 34. Удаљена праћена грана за teamone/master

Гурање

Када грану желите да поделите с остатком света, морате да је гурнете ка удаљеном серверу на коме имате право уписа. Ваше локалне гране се неће аутоматски синхронизовати са удаљеним серверима на које уписујете - морате експлицитно да пошаљете („гурнете“) гране које желите да поделите. На тај начин, можете да користите приватне гране за оно што не желите да поделите са осталима, и да гурате само у тематске гране у којима желите да сарађујете.

Ако имате грану која се зове `serverfix` на којој желите да радите са другима, можете да је гурнете навише на исти начин на који сте гурнули и и своју прву грану. Извршите `git push <име_удаљеног> <грана>`:

```
$ 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
```

Ово је донекле пречица. Програм Гит аутоматски развија име гране `serverfix` у

`refs/heads/serverfix:refs/heads/serverfix`, што значи „Узми моју локалну `serverfix` грану и турни је тако да се ажурира `serverfix` грана на удаљеном серверу.“ Део `refs/heads/` ћемо детаљније погледати у [Гит изнутра](#), али у оштетм случају можете да га изоставите. Можете да урадите и `git push origin serverfix:serverfix`, што ради исту ствар — каже „Узми моју `serverfix` грану и уреди да постане `serverfix` грана удаљеног сервера.“. Овај формат можете употребити да турнете локалну грану у удаљену грану која има другачије име. Ако не желите да се на удаљеном серверу зове `serverfix`, можете да извршите `git push origin serverfix:awesomebranch` чиме локалну `serverfix` грану гурате у `awesomebranch` грану на удаљеном пројекту.

Немојте да куцате своју лозинку сваки пут

Ако за гурање користите HTTPS URL, Гит сервер ће вас ради аутентификације питати за корисничко име и лозинку. Подразумевано ће вас у терминалу питати за ову информацију тако да сервер може одредити да ли вам је дозвољено да обавите операцију `push`.



Ако не желите да куцате ове податке сваки пут када вршите гурање, можете да подесите „кеш акредитива“. Најједноставнији начин је да их једноставно задржите у меморији на неколико минута, то лако можете подесити извршавањем `git config --global credential.helper cache`.

За више информација о разним опцијама за кеширање акредитива које су вам доступне, погледајте [Складиште акредитива](#).

Следећи пут када неко од ваших сарадника преузме податке са сервера, добиће референцу на место где се налази серверова верзија `serverfix` гране као удаљену грану `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 је приметити да када извршите преузимање података, тиме се довлаче и нове удаљене праћене гране, а ви немате аутоматски њихове локалне копије над којима можете да радите. Другим речима, у овом случају ви немате нову `serverfix` грану - имате само `origin/serverfix` показивач који не можете да мењате.

Да бисте спојили овај рад у грану на којој тренутно радите, можете да извршите `git merge origin/serverfix`. Ако желите своју личну `serverfix` грану на којој ћете да радите, можете да је базирате према удаљеној праћеној грани:

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

Ово вам прави локалну грану на којој можете да радите, а која почиње од места где се налази `origin/serverfix`.

Гране за праћење

Одјављивање локалне гране од удаљене праћене гране автоматски креира нешто што се зове „грана за праћење“ (а грана коју она прати се назива „узводна грана“). Гране за праћење су локалне гране које имају директну везу са удаљеном граном. Ако сте на грани за праћење и укуцате `git pull`, програм Гит автоматски зна са ког сервера треба да преузме податке и у коју грану треба да их споји.

Када клонирате репозиторијум, у општем случају се автоматски креира `master` грана која прати `origin/master`. Међутим, ако желите можете да поставите и друге гране за праћење—оне које прате гране на другим удаљеним серверима, или које не прате `master` грану. Прост случај је пример који сте управо видели, извршавање `git checkout -b <грана> <име_удаљеног>/<грана>`. Ова операција је толико честа да за њу програм Гит обезбеђује скраћеницу `--track`:

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

Уствари, ово се толико често користи да постоји чак и скраћеница ове скраћенице. Ако име гране коју покушавате да одјавите (а) не постоји и (б) потпуно се подудара са именом на само једном удаљеном серверу, програм Git ће за вас креирати грану за праћење:

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

Да бисте подесили локалну грану са именом које се разликује од оног које користи удаљена грана, можете лако да искористите прву верзију са другачијим именом локалне гране:

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

Сада ће ваша локална грана `sf` автоматски повлачiti са `origin/serverfix`.

Ако већ имате локалну грану и желите да је подесите на удаљену грану коју сте управо повукли, или желите да промените узводну грану коју пратите, можете да уз команду `git branch` употребите опцију `-u` или `--set-upstream-to` чиме је експлицитно задајете у било ком тренутку.

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

upstream пречица



Када имате подешену грану за праћење, њену узводну грану можете да референцирате пречицом `@{upstream}` или `@{u}`. Дакле, ако сте на `master` грани која прати `origin/master`, можете уместо `git merge origin/master` да задате нешто као `git merge @{u}` ако желите.

Да бисте видели све подешене гране за праћење, употребите опцију `-vv` уз команду `git branch`. Ово ће излисти ваше локалне гране са више информација, укључујући и то шта свака од грана прати, као и да ли је локална грана испред, иза или оба.

```
$ 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
```

Дакле, овде можемо да видимо да наша `iss53` грана прати `origin/iss53` и да се налази „испред” за два, што значи да локално имамо два комита која нису гурнута на сервер. Можемо да видимо и то да наша `master` грана прати `origin/master` и да је иста као и актуелна верзија на серверу. Даље, видимо да наша `serverfix` грана прати `serverfix-fix-good` грану на `teamone` серверу и да је испред за три и иза за један, што значи да постоји један комит на серверу који још увек нисмо спојили, али и да постоје три локална комита која још увек нисмо гурнули. Коначно, видимо да наша `testing` грана не прати ниједну удаљену грану.

Важно је запазити да су ови бројеви релативни у односу на тренутак када сте последњи пут преузели податке са сервера. Ова команда не ступа у везу са сервером, она вам говори само о ономе што је локално кеширала са тих сервера. Ако желите потпуно прецизне *ahead* и *behind* бројеве, мораћете најпре да преузмете податке са свих удаљених сервера пре него што покренете ову команду. То можете урадити на следећи начин:

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

Повлачење

Мада ће команда `git fetch` да преузме све промене на серверу које још увек немате, она уопште неће изменити ваш радни директоријум. Само ће преузети податке и оставиће вама да их спојите. Међутим, постоји команда која се зове `git pull` која је суштински у већини случајева `git fetch` за којом непосредно следи `git merge`. Ако имате грану за праћење подешену на начин који је показан у претходном одељку, било да сте је експлицитно подесили, или сте је добили као резултат команде `clone` или `checkout`, `git pull` ће погледати који сервер и грану ваша грана тренутно прати, преuzeће податке са тог сервера и онда ће пробати да их споји у ту грану за праћење.

У општем случају је боље да просто користите `fetch` и `merge` команде експлицитно, јер команда `git pull` често зна да уноси забуну.

Брисање удаљених грана

Претпоставимо да сте завршили са удаљеном граном — рецимо да сте ви и ваши сарадници завршили рад на некој могућности и спојили сте је са `master` граном вашег удаљеног сервера (или с којом год граном спајате стабилан кôд). Удаљену грану можете да обришете користећи `--delete` опцију уз команду `git push`. Ако желите да са сервера обришете вашу `serverfix` грану, извршите следеће:

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

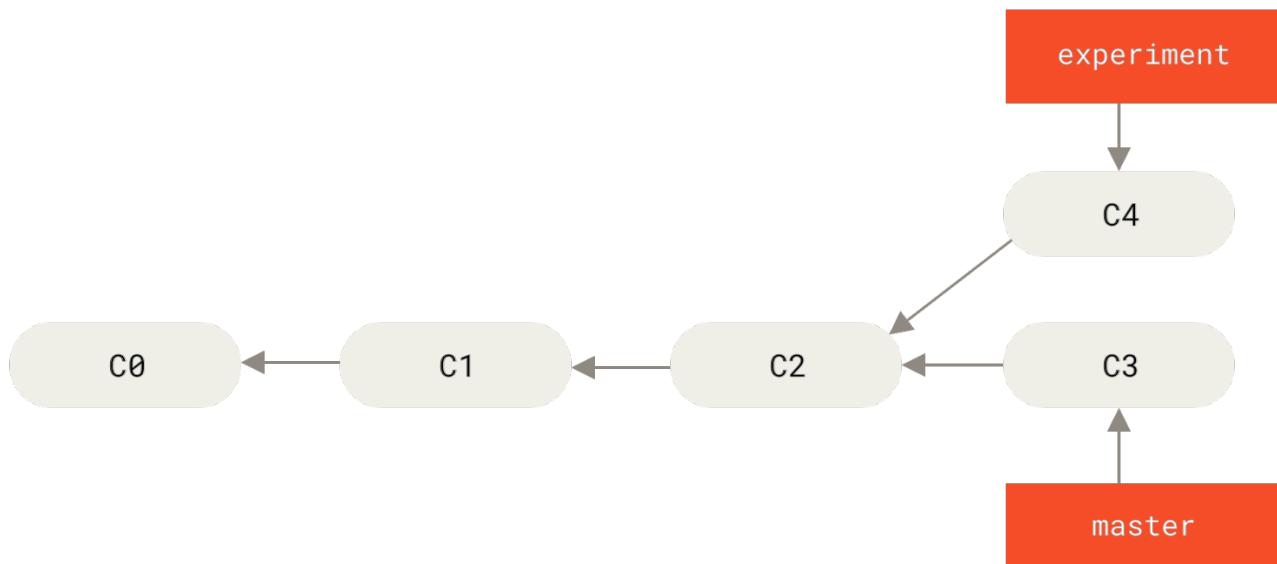
У суштини, ово само уклања показивач са сервера. Гит сервер ће у општем случају задржати податке тамо неко време док се не покрене *garbage collector*, тако да ако је грана обрисана случајно, често ће обнова података бити једноставна.

Ребазирање

У програму Гит постоје два основна начина за интеграцију промена из једне гране у другу: `merge` и `rebase`. У овом одељку ћете научити шта је ребазирање, како се ради, зашто је то прилично добар алат, као и када треба а када не треба да га користите.

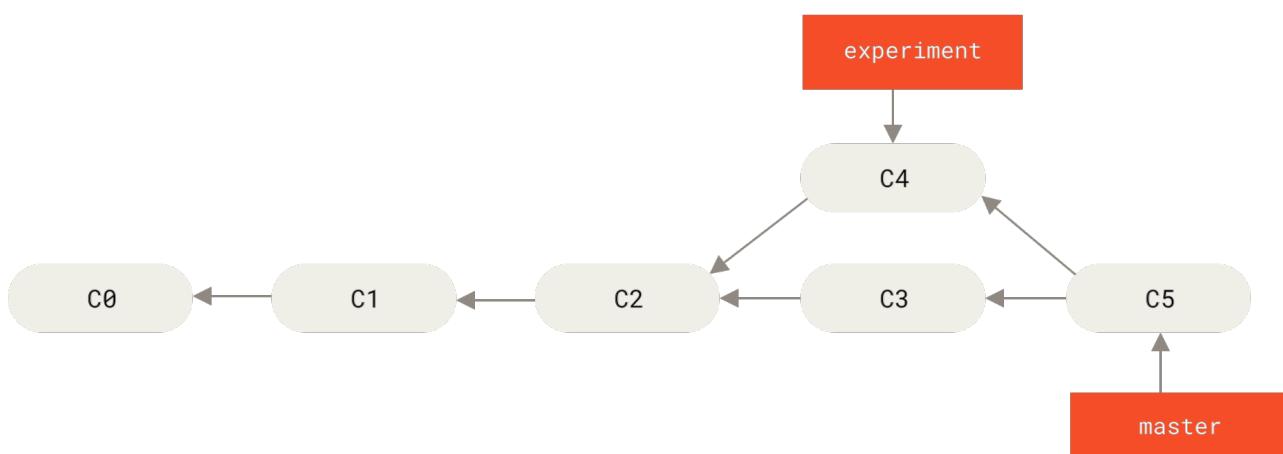
Основно ребазирање

Ако погледате ранији пример из [Основе спајања](#), видећете да сте разгранали свој рад и направили комитове на две различите гране.



Слика 35. Једноставна разграната историја

Како што смо већ раније показали, најлакши начин да интегришете гране је помоћу команде `merge`. Она ће урадити трострукот спајање између два последња снимка са грана (`C3` и `C4`) и њиховог најсвежијег заједничког претка (`C2`), стварајући нови снимак (и комит).



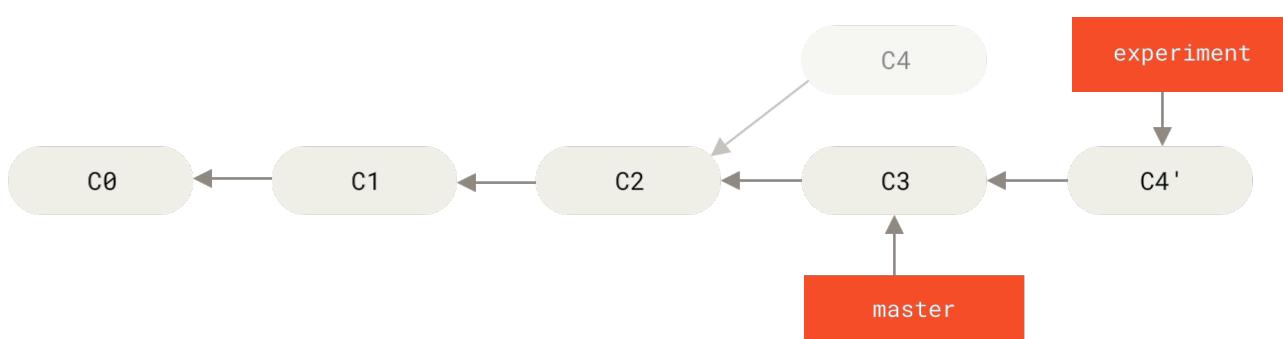
Слика 36. Спајање ради интеграције разгранате историје рада

Међутим, постоји још један начин: можете да узмете закрпу промене која је уведена у `C4` и да је поново примените преко `C3`. У програму Гит се ово зове *ребазирање*. `rebase` командом можете да узмете све промене које су комитоване у једну грану и да их поновите у некој другој.

У овом примеру, одјавили бисте грану `experiment`, па је затим ребазирали преко `master` гране на следећи начин:

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

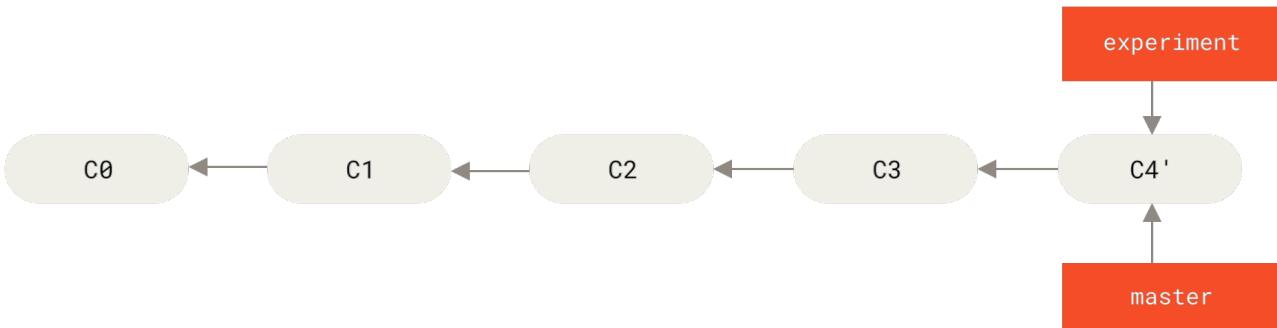
Ова операција функционише тако што оде на заједничког претка двеју грана (оне на којој се тренутно налазите и оне преко које ребазирате), узима разлику која је створена сваким комитом у грани на којој се налазите, чува те разлике у привремене фајлове, ресетује тренутну грану на исти комит на коме је и грana преко које ребазирате и коначно, редом примењује сваку промену.



Слика 37. Ребазирање промена које су уведене у `C4` преко `C3`

У овом тренутку можете да се вратите назад на `master` грану и да урадите спајање техником премотавања унапред.

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



Слика 38. Премотавање `master` гране унапред

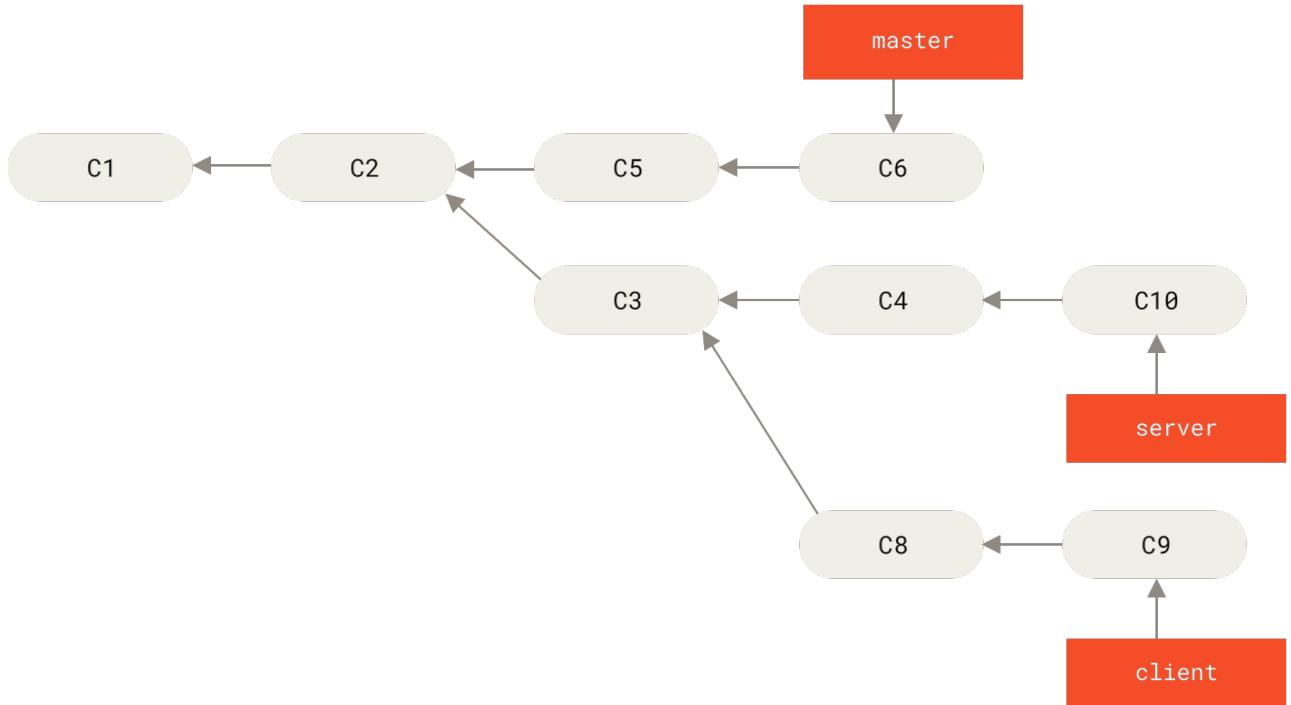
Сада је снимак на који показује `C4'` потпуно исти као и онај на који је показивао `C5` у примеру спајања. Нема разлике у крајњем производу интеграције, али ребазирањем се постиже чистија историја. Ако истражите лог ребазиране гране, изгледа као линеарна историја: изгледа као да се сваки рад одвијао серијски, иако су се ствари заправо одвијале паралелно.

Ово ћете често радити када желите се ваши комитови примене чисто на удаљену грану—можда у пројекту којем желите да дате допринос, али који не одржавате. У том случају, свој посао бисте радили у једној грани, па када будете спремни да пошаљете своје закрпе главном пројекту, ребазираћете свој рад преко `origin/master`. На овај начин, одржавалац не мора да ради никакав посебан посао око интеграције—само треба да премота унапред или одради чисто примењивање.

Обратите пажњу на то да је завршни снимак на који показује коначни комит, било да је то последњи од ребазираних комитова у случају ребазирања, или коначни комит спајања након спајања, један те исти—само се историја разликује. Ребазирање понавља у некој другој све промене урађене у једној линији рада и то редом којим су прављене, док спајање узима крајње тачке и спаја их.

Интересантнији случајеви ребазирања

Ребазирање може да понови измене и над нечemu другом, што није циљна грана ребазирања. На пример, узмите историју као што је приказана на [Историја са тематском граном разгранатом од друге тематске гране](#). Разгранали сте тематску грану (`server`) да бисте у свој пројекат додали неку функционалност са серверске стране, па направили комит. Онда сте разгранали и од ње да бисте направили неке промене на клијентској страни (`client`), па комитовали неколико пута. Коначно, вратили сте се на `server` грану и направили још неколико комитова.

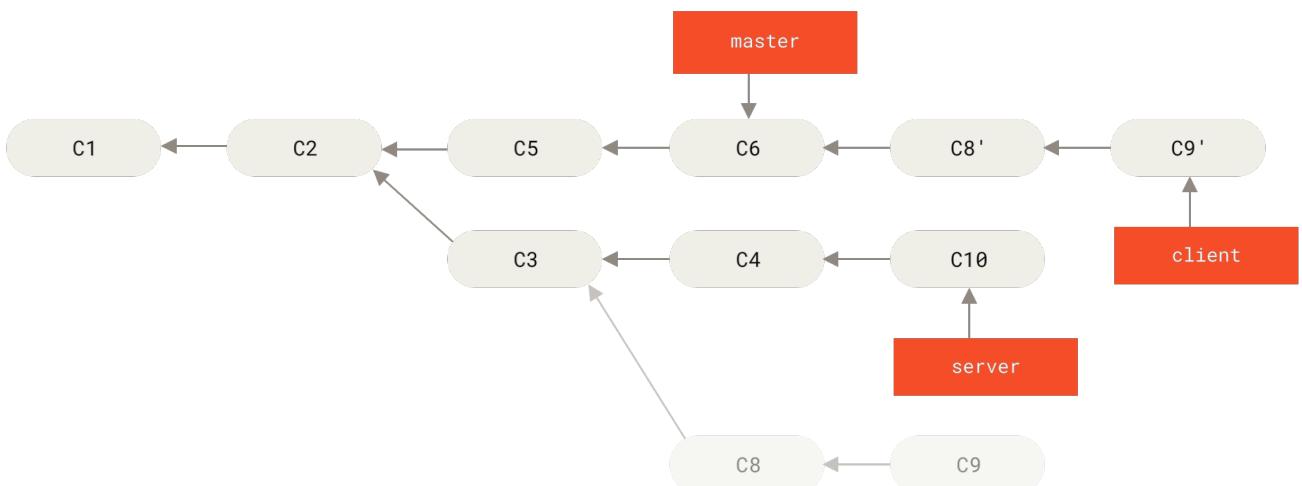


Слика 39. Историја са тематском граном разгранатом од друге тематске гране

Претпоставимо да сте одлучили да желите спојити ваше промене на клијентској страни са главном граном како би се објавиле, али желите да одложите промене на серверској страни док их боље не тестирате. Можете да узмете промене са `client` гране које нису на `server` грани (`C8` и `C9`) и да их поновите преко `master` гране користећи опцију `--onto` команде `git rebase`:

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

Ово у суштини каже „Провери грану `client`, одреди закрпе које су настале након што се одвојила од `server` гране, па их онда поново примени у грани `client` као да је уместо од `server` била одвојена директно од `master` гране“. Мало је сложено, али резултат је одличан.

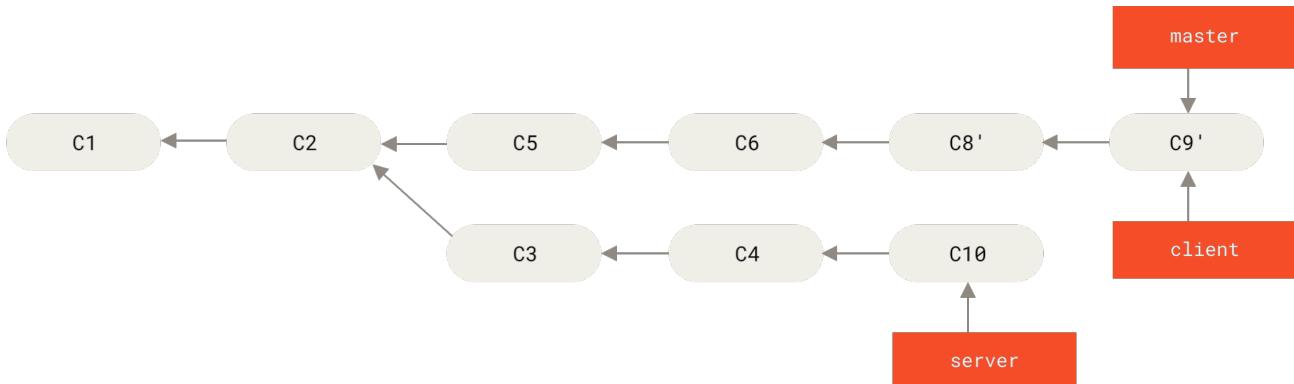


Слика 40. Ребазирање тематске гране разгранате из друге тематске гране

Сада можете да премотате унапред грану `master` (погледајте [Премотовање master гране](#)

унапред тако да обухвати промене са гране `client`):

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

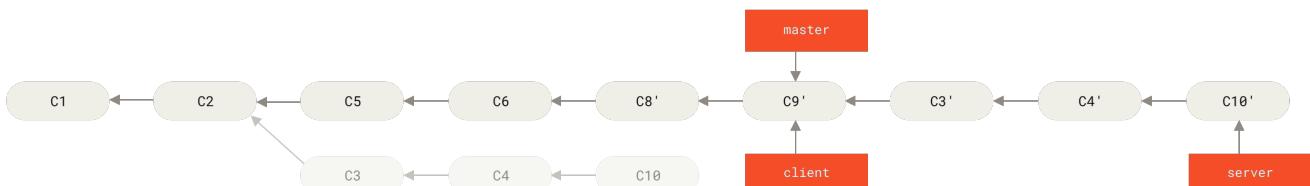


Слика 41. Премотавање `master` гране унапред тако да обухвати промене са гране `client`

Рецимо да сте одлучили да повучете и промене из `server` гране. `server` грану можете да ребазирате преко `master` гране са `git rebase <основна_грана> <тематска_грана>` без потребе да је прво одјавите—ова команда прво одјави тематску грану (у овом случају `server`) и примењује пронађене промене на основну грану `master`):

```
$ git rebase master server
```

Ово понавља рад са `server` гране преко `master` гране, као што се види на [Ребазирање `server` гране преко `master` гране](#).



Слика 42. Ребазирање `server` гране преко `master` гране

Затим премотате унапред основну грану (`master`):

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

Сада можете да обришете гране `client` и `server` јер је сви рад обављен на њима интегрисан и више вам неће бити потребне, а историја рада након овог процеса ће изгледати као на [Коначна историја комитова](#):

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



Слика 43. Коначна историја комитова

Опасности ребазирања

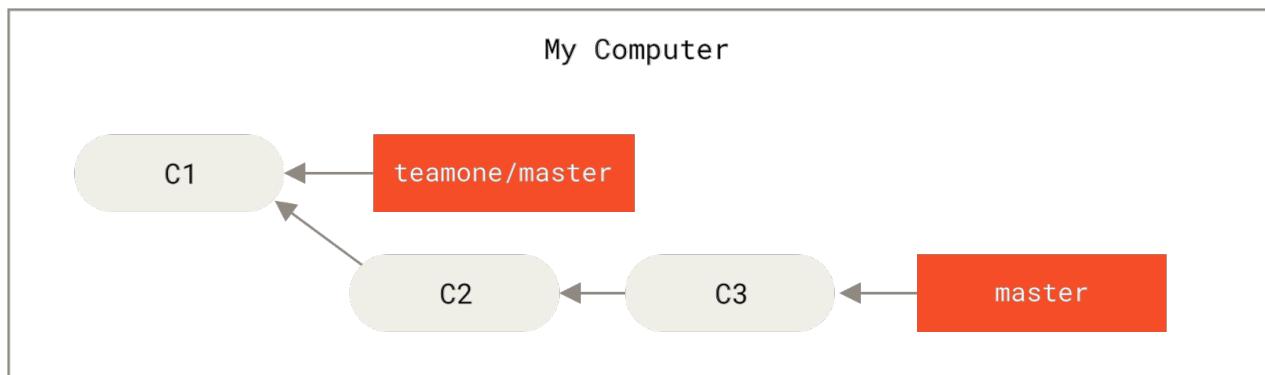
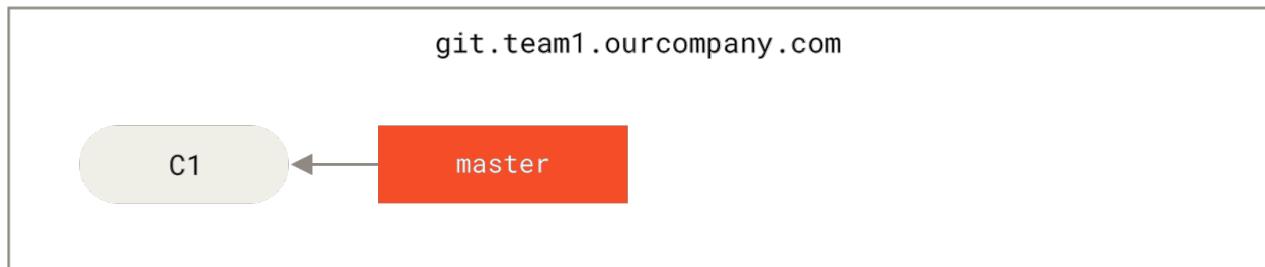
Ах, али блаженство ребазирања није без мана, што се може сумирати само једном реченицом:

Не ребазирајте комитове који постоје ван вашег репозиторијума и на којима су људи можда засновали свој рад.

Ако се држите ове смернице, све ће бити у реду. У супротном ће вас људи мрзети, а породица и пријатељи ће вас презирати.

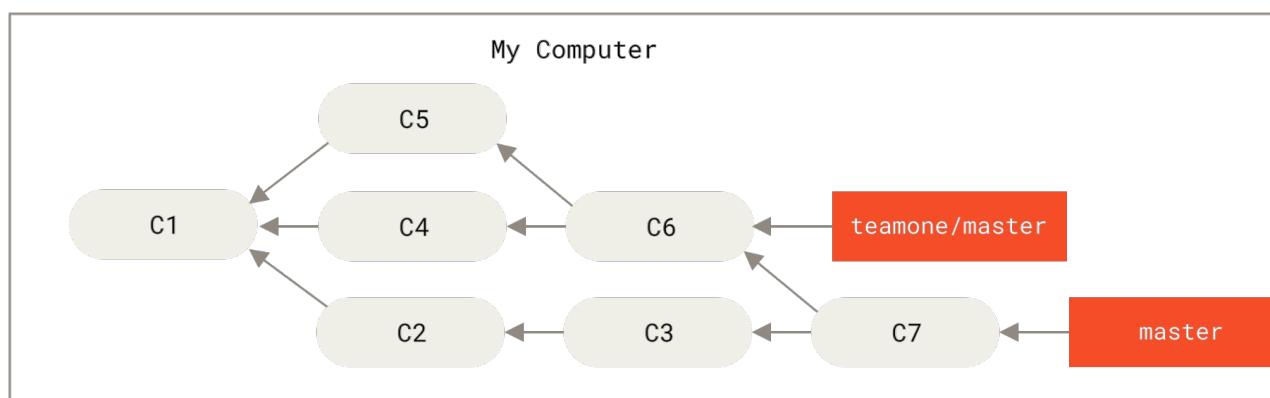
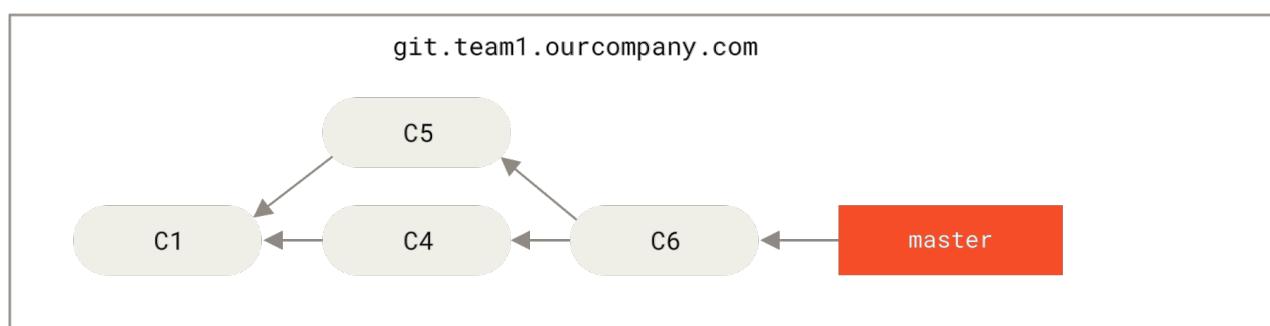
Када нешто ребазирате, ви напуштате постојеће комитове и стварате нове који су им слични, али су ипак другачији. Ако комитове гурнете негде и остали их повуку, па базирају свој рад над њима, а ви затим поново напишете те комитове са `git rebase` и гурнете их поново, ваши сарадници ће морати да поново споје свај рад и онда ће настати хаос када пробате да повучете њихов рад назад у свој.

Погледајмо пример који показује како ребазирани рад који сте учинили јавно доступним може изазвати проблеме. Претпоставимо да сте направили клон са централног сервера и онда радили нешто почевши од њега. Историја комитова изгледа овако:



Слика 44. Клонирани репозиторијум над којим сте обавили неки посао

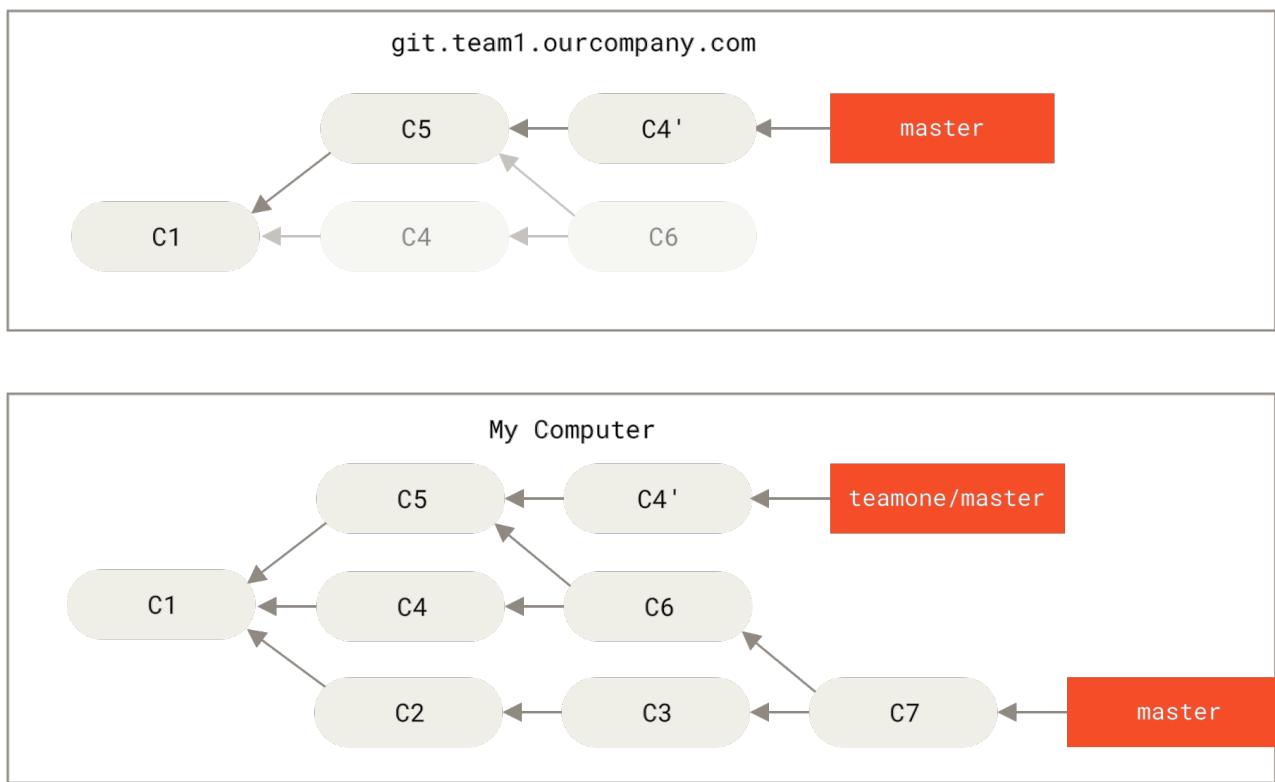
Сада, неко други уради још нешто што укључи и спајање, а затим турне све на централни сервер. Ви то преузмете и спојите нову удаљену грану са оним што сте урадили, тако да историја изгледа некако овако:



Слика 45. Преузимање још комитова и спајање са личним радом

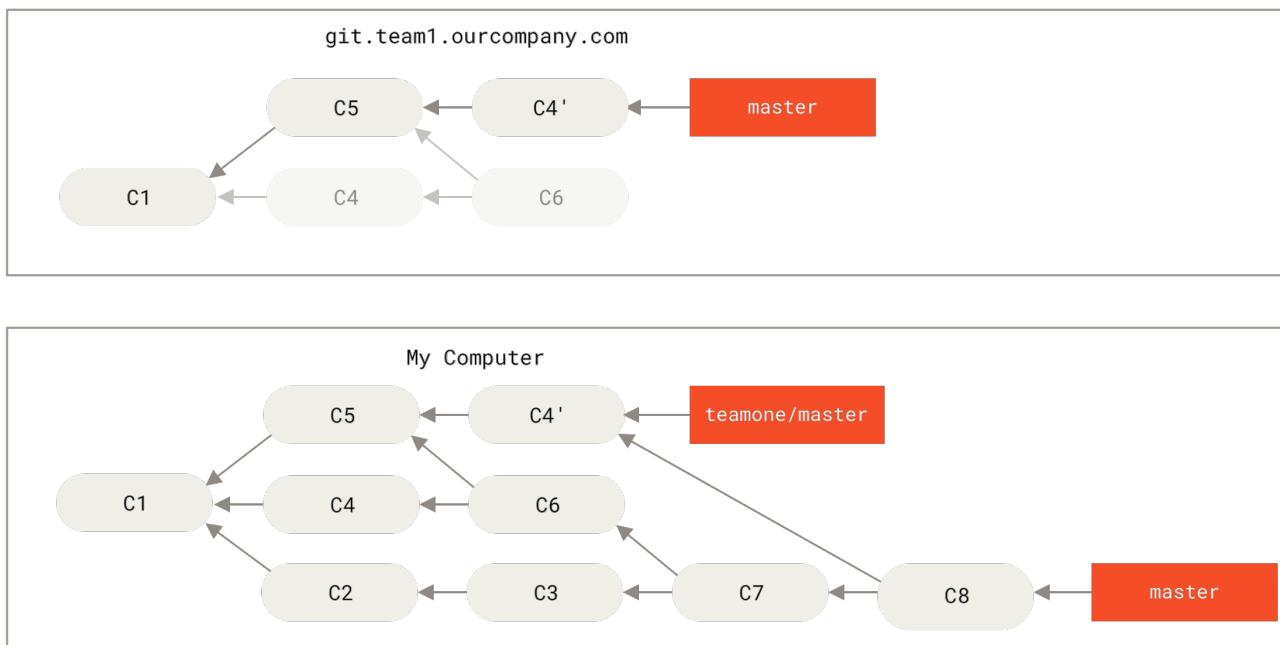
Затим, особа која је гурнула спојен рад одлучи да се врати назад и уместо спајања ребазира

оно што је одрадила; изврши `git push --force` како би се преписала историја на серверу. Ви онда преузмете податке са тог сервера, довлачећи нове комитове.



Слика 46. Неко гурне ребазирање комитове, напуштајући комитове над којима сте базирали ваш рад

Сада сте обоје у сосу. Ако извршите `git pull`, направићете комит спајања који укључује обе линије историје, и ваш репозиторијум ће изгледати овако:



Слика 47. Поновно спајање истог рада у нови комит спајања

Ако извршите `git log` када ваша историја изгледа овако, видећете два комита који имају

истог аутора, време и поруку, што ће унети забуну. Штавише, ако гурнете ову историју назад на сервер, поново ћете увести све те ребазиране комитове на централни сервер, што ће још више збунити људе. Прилично је безбедно претпоставити се да други програмер не жели да се [C4](#) и [C6](#) нађу у историји; то је разлог зашто су и радили ребазирање.

Ребазирање када ребазирате

Ако се **ипак** нађете у оваквој ситуацији, програм Гит има још неке чаролије које вам могу помоћи. Ако неко из тима насиљно гурне промене које препишу рад над којем сте ви базирали свој рад, изазов који вам се намеће је да одредите шта је ваше, а шта је та особа преписала.

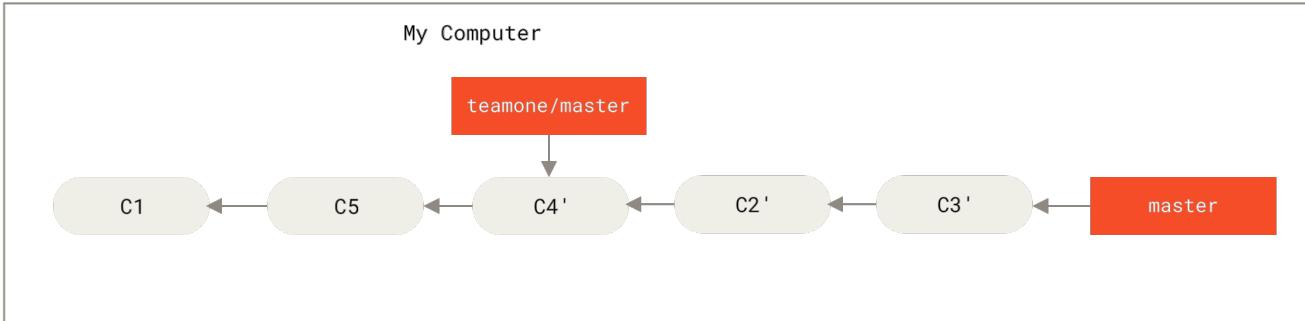
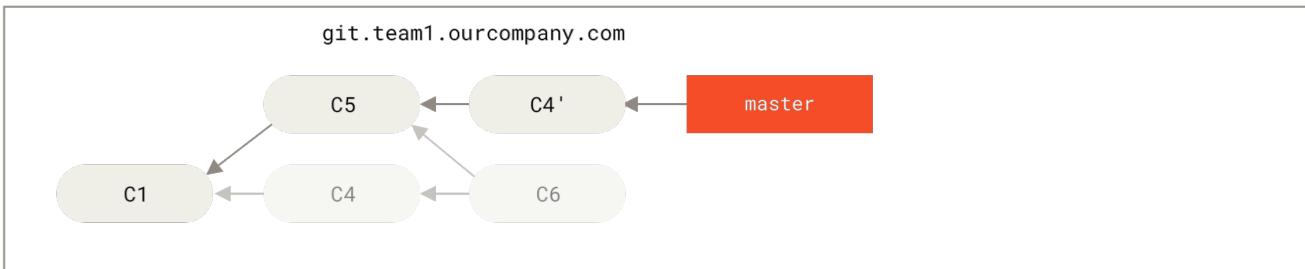
Испоставља се да поред SHA-1 контролне суме комита, програм Гит рачуна и контролну суму која је базирана само на закрпи која је уведена комитом. Ово се зове „идентификациони број закрпе” (*patch-id*).

Ако повучете рад који је преписан и ребазирате га преко нових комитова вашег партнера, програм Гит често сам може успешно да одреди шта је јединствено ваше и да примени то назад на врх нове гране.

На пример, ако у претходном сценарију када смо били код [Неко гурне ребазиране комитове, напуштајући комитове над којима сте базирали ваш рад](#) уместо спајања извршимо `git rebase teamone/master`, програм Гит ће:

- одредити који рад је јединствен за нашу грану ([C2](#), [C3](#), [C4](#), [C6](#) и [C7](#)),
- одредити шта нису комитови спајања ([C2](#), [C3](#) и [C4](#)),
- одредити шта није било преписано у одредишну грану (само [C2](#) и [C3](#), пошто је [C4](#) иста закрпа као и [C4'](#)) и
- применити те комитове на врх `teamone/master` гране.

Тако да ћемо, уместо резултата који видимо на [Поновно спајање истог рада у нови комит спајања](#), добити нешто што више подсећа на [Ребазирање преко насиљно гурнутог ребазираног рада](#).



Слика 48. Ребазирање преко насиљно гурнутог ребазираног рада

Ово ће функционисати само ако су **C4** и **C4'** који је ваш партнери направио скоро идентична закрпа. У супротном, ребазирање неће моћи да установи да је то дупликат и додаће још једну закрпу која подсећа на **C4** (и која вероватно неће моћи чисто да се примени, јер би промене бар донекле већ биле тамо).

Ово можете да упростите и извршавањем `git pull --rebase` уместо обичног `git pull`. Или можете то ручно да урадите са `git fetch` за којим у овом случају следи `git rebase teamone/master``.

Ако користите `git pull` и желите да `--rebase` буде подразумевана опција, можете да подесите `pull.rebase` вредност из конфигурационог фајла на `true` са `git config --global pull.rebase true`.

Ако само ребазирате комитове који никада нису напустили ваш рачунар, све ће бити у реду. Ако ребазирате комитове који су били гурнути, али нико други није на њима базирао свој рад, такође неће бити проблема. Ако ребазирате комитове који су већ гурнути јавно и могуће је да су људи базирали свој рад на тим комитовима, онда ћете се наћи у фрустријујућим ситуацијама и бићете мета презира својих сарадника.

Ако ви или партнери у неком тренутку схватите да је овакав след догађаја неопходан, постарајте се да сви остали знају да треба да изврше `git pull --rebase` и тако пробају да макар донекле упросте проблем који настаје након ребазирања.

Ребазирање против спајања

Сада када сте видели како функционише ребазирање а како спајање, можда се питате шта је боље. Пре него што дамо одговор на ово, хајде да начинимо корак уназад и попричамо мало о томе шта је заправо историја.

Једна тачка гледишта је да историја комитова вашег репозиторијума представља **запис**

онога што се заправо догодило. То је историјски документ, вредан сам по себи, па не би требало да се преправља. Из овог угла, мењање историје комита је скоро па богохуљење; ви лажете о ономе што се заправо догодило. Шта онда радити када се додги серија збрканих комитова спајања? Па, ствари су се тако додгиле и репозиторијум треба да сачува то за потомство.

Супротна тачка гледишта је да историја комитова представља **причу о томе како је пројекат направљен**. Не бисте објавили прву скицу књиге, па зашто да прикажете свој траљави посао? Када радите на пројекту, може вам бити потребан запис о свим погрешним корацима које се начинили и свим ћорсокацима у које сте ушли, али када дође време да објавите свету свој рад, пожелећете да прикажете прецизнију причу о томе како се долази од тачке А до тачке Б. Овај табор користи алате као што је ребазирање и филтер гране да поново испише комитове пре него што се споје у главну грану. Употребом алата `rebase` и `filter-branch` они причају причу на начин који је најбољи за будуће читоце.

Сада, што се тиче питања да ли је боље спајање или ребазирање: надамо се да ћете увидети да ствари нису тако једноставне. Програм Гит је моћан алат, допушта вам да урадите многе ствари са својом историјом, али сваки тим и сваки пројекат је другачији. Сада када знате како обе ове ствари раде, на вами је да одлучите шта је боље за вашу конкретну ситуацију.

А можете да добијете и најбоље из оба света: ребазирајте локалне промене него што их гурнете како бисте пречистили свој рад, али никада немојте да ребазирате било шта ште сте негде гурнули.

Резиме

Покрили смо основе гранања и спајања у програму Гит. Сада би требало да вам креирање и скакање на нове гране, као и преласци с гране на грану, или спајање локалних грана делује потпуно природно. Требало би да сте у стању да делите своје гране тако што ћете их гурнути на дељени сервер, да радите са осталима на дељеним гранама, као и да ребазирате своје гране пре него што их поделите. Сада ћемо објаснити шта све морате знати да бисте поседовали свој лични сервер на коме ћете хостовати репозиторијум.

Гит на серверу

Сада би требало да будете у стању да урадите већину свакодневних задатака помоћу програма Гит. Међутим, да би постојао било какав вид сарадње у програму Гит, неопходно је да имате удаљени Гит репозиторијум. Мада технички можете да гурате промене као и да их вучете са репозиторијума појединачних људи, такав приступ се не препоручује јер ако не будете пажљиви врло лако може доћи до забуне око тога на чиму они раде. Штавише, пожелећете да ваши сарадници могу приступити репозиторијуму чак и када је ваш рачунар ван мреже—често је корисно имати поузданији заједнички репозиторијум. Зато је пожељна метода за сарадњу са неким постављање посредничког репозиторијума којем сви имају приступ, и гурање и повлачење с њега.

Управљање Гит сервером је врло једноставно. Најпре изаберете протоколе које желите да ваш сервер подржава. Први одељак овог поглавља ће представити доступне протоколе и навести предности и мање сваког од њих. Следећи одељци ће објаснити неке типичне поставке које користе ове протоколе и начин за подешавање сервера тако да их користи током извршавања. На крају, прећи ћемо неколико опција за хостовање, ако вам не смета да свој код хостујете на туђем серверу и не желите да се мучите да поставите и одржавате сопствени сервер.

Ако вас не занима вођење сопственог сервера, можете да прескочите на последњи одељак овог поглавља и погледате неке опције за подешавање хостованог налога и онда да пређете на следеће поглавље, где ћемо дискутовати о разним добрим и лошим странама рада у дистрибуираном окружењу за контролу извornog кода.

Удаљени репозиторијум је у општем случају *огољени репозиторијум*—Гит репозиторијум који нема радни директоријум. Пошто се репозиторијум користи само као тачка за сарадњу, нема разлога да на диску има одјављен снимак; то су само подаци програма Гит. Најједноставније речено, огољени репозиторијум је садржај `.git` директоријума вашег пројекта и ништа друго.

Протоколи

Програм Гит може да користи четири главна протокола за пренос података: *Локални*, *HTTP*, *Secure Shell* (SSH) и *Git*. Овде ћемо размотрити шта су они и у каквим околностима бисте желели (или не бисте желели) да их користите.

Локални протокол

Најосновнији је *Локални протокол*, код кога се удаљени репозиторијум налази у неком другом директоријуму на диску. Ово се често користи ако сви у тиму имају приступ дејеном фајл систему као што је [NFS](#), или у мање вероватном случају ако се сви пријављују на исти рачунар. Други случај не би био идеalan, јер би се све инстанце кода из репозиторијума налазиле на истом рачунару, чиме би катастрофални губитак података био вероватнији.

Ако имате прикачен дејени фајл систем (*shared mounted filesystem*), онда можете да клонирате локални репозиторијум, да гурате на њега и да повлачите са њега. Да бисте клонирали овакав репозиторијум или га додали као удаљени неком постојећем пројекту,

употребите путању до репозиторијума као URL. На пример, да бисте клонирали локални репозиторијум, извршите нешто овако:

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

Можете да урадите и следеће:

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

Програм Гит ради мало другачије ако експлицитно наведете `file://` на почетку УРЛ адресе. Ако наведете само путању, програм Гит покушава да користи чврсте линкове или да директно копира фајлове који су му потребни. Ако наведете `file://`, програм Гит покреће процес који обично користи за пренос података преко мреже, а који је у општем случају много мање ефикасна метода за пренос података. Главни разлог због кога бисте можда желели да наведете префикс `file://` је уколико желите чисту копију репозиторијума са избаченим сувишним референцама или изостављеним објектима — рецимо после импорта са другог система за контролу верзије или нешто слично (погледајте [Гит изнутра](#) у вези послова око одржавања.) Овде ћемо користити нормалну путању јер је тако скоро увек брже.

Да бисте додали локални репозиторијум у постојећи Гит пројекат, можете да извршите нешто овако:

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

Онда можете да гурате и да повлачите са тог удаљеног репозиторијума преко новог имена удаљеног `local_proj` као да то радите преко мреже.

Предности

Предности репозиторијума заснованих на фајловима су чињеница да су једноставни и да користе постојеће дозволе над фајловима и приступ мрежи. Ако већ имате дељени фајл систем коме цео тим приступа, подешавање оваквог репозиторијума је веома лако. Сместите огољену копију репозиторијума негде где сви имају дељени приступ и подесите дозволе за читање и упис као што бисте урадили и код било ког другог дељеног директоријума. У [Постављање програма Гит на сервер](#) ћемо дискутовати о томе како да извезете огољену копију репозиторијума за ову намену.

Ово је такође добра опција да брзо зграбите рад са туђег радног репозиторијума. Ако ви и сарадник радите на истом пројекту тај сарадник жели да нешто одјавите, извршавањем команде као што је `git pull /home/john/project`, често је то лакше него да сарадник тура податке на удаљени сервер, а да их ви након тога повучете.

Мане

Мане ове методе су то што је код дељеног приступа подешавање и приступање серверу са

различитих локација у општем случају теже него код основног приступа мрежи. Ако желите да гурнете са лаптопа када сте кући, морате да монтирате удаљени диск, што може да буде тешко и споро у поређењу са приступом преко мреже.

Важно је поменути да ово није увек и најбржа опција ако користите неку врсту дељеног монтираног. Локални репозиторијум је брз само ако имате брз приступ подацима. Репозиторијум на NFS систему је често спорији него репозиторијум преко SSH на истом серверу, јер допушта програму Гит да се на сваком систему извршава са локалних дискова.

Конечно, овај протокол не штити репозиторијум од случајног оштећења. Сваки корисник има потпуни приступ „удаљеном“ директоријуму преко командног окружења и ништа га не спречава да промени или уклони интерне фајлове програма Гит, па на тај начин поквари репозиторијум.

HTTP протоколи

Програм Гит може да комуницира преко HTTP протокола у два различита режима. Пре верзије 1.6.6 програма Гит, постојао је само један начин на који се ово могло обавити; био је веома једноставан и у општем случају је дозвољавао само читање. У верзији 1.6.6 је представљен нови, интелигентнији протокол који је омогућио програму Гит да на паметан начин може да преговара о размени података, на сличан начин на који то ради преко SSH. У последњих неколико година, овај нови HTTP протокол је постао веома популаран зато што је једноставнији за кориснике и паметнији у обављању комуникације. Нова верзија се често назива *Паметни* HTTP протокол, а стари начин *Приглуп* HTTP. Прво ћемо покрити нови, Паметни HTTP протокол.

Паметни HTTP

Паметни HTTP протокол ради слично као SSH или Гит протоколи, с тим што се обавља преко стандардних HTTPS портова и може да користи разне HTTP механизме за аутентификацију, што значи да је из перспективе корисника једноставнији за употребу него нешто као што је SSH, пошто омогућава употребу на пример, основне аутентификације корисничким именом и шифром, уместо SSH кључевима.

Ово је сада вероватно постао најпопуларнији начин за коришћење програм Гит, пошто се може подесити тако да пружа услуге и анонимно као протокол `git://`, али и помоћу аутентификације и шифровања као SSH протокол. Уместо да подешавате различите URL адресе за ове ствари, сада можете да користите само један URL за оба. Ако пробате да гурнете на репозиторијум који захтева аутентификацију (што би и требало да буде случај), сервер може да вам затражи корисничко име и лозинку. Исто важи и за приступ читањем.

Заправо, за сервисе као што је GitHub, URL адреса коју користите да на мрежи погледате репозиторијум (на пример, <https://github.com/schacon/simplegit>) је иста URL адреса коју можете користити за клонирање, а ако имате приступ, и за гурање.

Приглуп HTTP

Ако сервер не одговара паметним HTTP сервисом програма Гит, Гит клијент ће покушати да се повеже користећи једноставнији Приглуп HTTP протокол. Приглуп протокол очекује да се

огољени Гит репозиторијум сервира као обични фајлови са веб сервера. Лепота Приглупог HTTP протокола је једноставност његовог подешавања. У основи, све што треба да урадите јесте да објавите огољени Гит репозиторијум под кореном HTTP докумената и да подесите специфичну `post-update` куку и то је то (погледајте [Гит куке](#)). Сада свако ко може да приступи веб серверу на који сте поставили репозиторијум може и да клонира ваш репозиторијум. Да бисте дозволили читање репозиторијума преко HTTP протокола, урадите нешто слично следећем:

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

И то је све. `post-update` кука која подразумевано долази уз програм Гит покреће одговарајућу команду (`git update-server-info`) која омогућава да HTTP преузимање и клонирање ради како треба. Ова команда се покреће када гурате на овај репозиторијум (на пример преко SSH); затим други људи онда могу да клонирају користећи команду налик следећој:

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

У овом случају, користимо путању `/var/www/htdocs` која је честа код Apache сервера, али можете да користите и било који статички веб сервер—само поставите огољени репозиторијум у његову путању. Гит подаци се сервирају као обични статички фајлови (погледајте [Гит изнутра](#) за детаље о томе како се тачно обавља сервирање).

У општем случају треба да направите избор између Паметног HTTP сервера који нуди читање и упис, или да омогућите једноставан приступ само за читање користећи Приглуп приступ. Ова два сервиса се ретко мешају.

Предности

Концентрисаћемо се на предности Паметне верзије HTTP протокола.

Једноставност чињенице да постоји јединствена URL адреса за све врсте приступа, као и то што сервер од корисника захтева аутентификацију само онда када је она неопходна чини ствари много једноставнијим за крајњег корисника. То што је могућа аутентификација помоћу корисничког имена и шифре је такође велика предност над SSH протоколом, јер корисници не морају локално да генеришу SSH кључеве, па да јавни кључ постављају на сервер пре него што могу да успоставе комуникацију са њим. За мање софистициране кориснике, или за кориснике на системима где SSH није тако чест избор комуникације, ово је велика предност код употребљивости. Ово је такође веома брз и ефикасан протокол, сличан SSH протоколу.

Сем тога, преко HTTPS протокола своје репозиторијуме можете сервирати и само за читање, што значи да пренос садржаја можете да шифрујете; или можете да идете до те мере да натерате клијенте да користе посебне потписане SSL сертификате.

Још једна лепа ствар је то што су HTTP и HTTPS толико често коришћени протоколи да су корпоративни фајеролови често подешени тако да дозвољавају пренос таквог саобраћаја кроз своје портove.

Мане

Гит преко HTTPS протокола уме да буде мало незгоднији за подешавање у поређењу са подешавањем SSH на неким серверима. Сем тога, када је у питању пренос Гит садржаја, постоји врло мало предности које други протоколи нуде у односу на Паметни HTTP.

Ако користите HTTP за аутентификовани гурање, достављање акредитива је понекад компликованије него коришћење кључева преко SSH. Ипак, постоји неколико алата за кеширање акредитива које можете користити, укључујући *Keychain* приступ на мекОС и *Credential Manager* за Виндууз, што ће учинити овај поступак прилично безболним. Прочитајте [Складиште акредитива](#) да бисте видели како да на свом систему подесите сигурни HTTP систем за кеширање лозинки.

SSH протокол

Уобичајени протокол за пренос података који се користи када сами обављате хостинг јесте SSH. То је зато што је SSH приступ серверима већ подешен на већини места — а ако и није, лако се подешава. SSH је такође и мрежни протокол са аутентификацијом, а пошто је свеприсутан, у општем случају се лако подешава и користи.

Да бисте клонирали репозиторијум преко SSH, можете на следећи начин да наведете `ssh://` URL:

```
$ git clone ssh://[корисник@]сервер/проект.git
```

Или можете употребити краћу синтаксу која подсећа на `scp` за SSH протокол:

```
$ git clone [корисник@]сервер:проект.git
```

У оба претходна случаја није обавезно да наведете корисника, програм Гит ће у том случају претпоставити да се ради о кориснику који је тренутно пријављен на систем.

Предности

Постоји много предности SSH протокола. За почетак, SSH је једноставан за подешавање — SSH демони су опште присутни, многи мрежни администратори имају искуство с њима и многе дистрибуције оперативних система су подешене тако да имају алате за рад са њима. Затим, приступ преко SSH је безбедан — сваки пренос података се шифрује и аутентификује. За крај, као и HTTPS, Гит и Локални протокол, SSH је ефикасан, јер компресује податке што је више могуће пре него што започне пренос.

Мане

Лоша страна SSH протокола је то што не можете пружити анонимни приступ репозиторијуму. Ако користите SSH, људи *морају* да имају SSH приступ машини, чак и само за читање, што SSH не чини погодним за пројекте отвореног кода, где људи понекад само желе да клонирају ваш репозиторијум како би истражили кôд. Ако га користите само унутар корпоративне мреже, SSH би могао да буде једини протокол који треба да користите. Ако својим пројектима желите да дозволите анонимни приступ само за читање, али истовремено желите да користите SSH, за вас ћете морати да подесите SSH за гурање промена, али и нешто друго преко чега ће остали моћи да преузимају податке.

Гит протокол

На реду је Гит протокол. Ово је посебни демон који долази уз програм Гит; он слуша на одређеном порту (9418) и нуди услуге сличне SSH протоколу, али без икакве аутентификације. Да би се репозиторијум сервирао преко Гит протокола, морате да креирате датотеку `git-daemon-export-ok`—демон неће сервирати репозиторијум ако тај фајл не постоји—али осим тога, нема никаквих сигурносних мера. Гит репозиторијум је или доступан свима или није уопште. Ово значи да у општем случају није могуће гурање преко овог протокола. Можете да омогућите приступ за писање; али пошто нема аутентификације, ако то укључите, свако на интернету ко нађе URL адресу вашег пројекта ће моћи да гура своје измене на њега. Нема потребе посебно наглашавати да је оваква употреба реткост.

Предности

Гит протокол је често најбржи доступни мрежни протокол за пренос података. Ако треба опслужити велики саобраћај за јавни пројекат или веома велики пројекат који не захтева аутентификацију корисника за читање, вероватно ћете пожелели да подесите Гит демона који ће сервирати ваш пројекат. Користи исти механизам за пренос података као SSH протокол, али без додатних трошкова за шифровање и аутентификацију.

Мане

Мана Гит протокола је непостојање аутентификације. У општем случају није пожељно да Гит протокол буде једини начин за приступ пројекту. Обично ћете га упарити са SSH или HTTPS приступом за неколико програмера који имају дозволу за гурање (упис), а сви остали могу да користе `git://` само за читање. Такође је вероватно најкомплекснији протокол за подешавање. Мора да покреће сопствени демон, који захтева `xinetd,systemd` или неку сличну конфигурацију, што није увек баш толико једноставно. Сем тога, захтева да фајервол допусти приступ порту 9418, што није стандардни порт који би корпоративни фајерволови допустили. Из великих корпоративних фајеролова, овај неуобичајени порт је најчешће блокиран.

Постављање програма Гит на сервер

Сада ћемо показати како се подешава Гит сервис тако да извршава ове протоколе на вашем сопственом серверу.



Овде ћемо демонстрирати команде и кораке неопходне ради основне, поједностављене инсталације на серверу базираном на Линуксу, мада је могуће покренути ове сервисе и на мекОС или Виндоуз серверима. Заправо, постављање производног сервера унутар ваше властите инфраструктуре ће несумњиво подразумевати и неке разлике по питању сигурносних мера или алата које пружа оперативни систем, али надамо се да ће вам ово дати општу слику о томе шта треба урадити.

Да бисте иницијално поставили било који Гит сервер, постојећи репозиторијум морате прво да извезете у нови огольени репозиторијум—репозиторијум који не садржи радни директоријум. У општем случају се то ради једноставно. Да бисте клонирали свој репозиторијум и направили нови огольени репозиторијум, можете да извршите команду `clone` уз опцију `--bare`. По конвенцији, огольени репозиторијуми се завршавају са `.git`, на пример:

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

Сада би требало да имате копију података из Гит директоријума у директоријуму `my_project.git`.

Ово је отприлике еквивалентно са следећим:

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

Постоји неколико малих разлика у конфигурационом фајлу; али за ову сврху, у питању јеовољно приближна алтернатива. Ова команда узима сâм Гит репозиторијум, без радног директоријума, и креира директоријум намењен посебно њему.

Постављање огольеног репозиторијума на сервер

Сада када имате огольену копију репозиторијума, све што треба да урадите јесте да је окачите на сервер и поставите протоколе. Речимо да сте поставили сервер `git.example.com` коме имате SSH приступ и све своје Гит репозиторијуме желите да ускладиштите у директоријум `/srv/git`. Под претпоставком да `/srv/git` постоји на том серверу, нови репозиторијум можете да подесите тако што ћете тамо прекопирати свој огольени репозиторијум:

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

Сада и други корисници који имају SSH приступ истом серверу за читање директоријума `/srv/git` могу да клонирају ваш репозиторијум извршавањем:

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

Ако корисник изврши SSH приступ серверу и има приступ писања у директоријум `/srv/git/my_project.git`, аутоматски ће имати и привилегију да гурају измене.

Програм Гит ће аутоматски додати групне дозволе за упис у репозиторијум ако покренете команду `git init` уз опцију `--shared`.

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

Видите колико је једноставно узети Гит репозиторијум, креирати његову огольену верзију и поставити је на сервер коме ви и ваши сарадници имате SSH приступ. Сада сте спремни за сарадњу над истим пројектом.

Важно је да приметити да је ово буквально све што треба да урадите ако желите да покренете употребљив Гит сервер коме неколико људи може да приступи — само додајте налоге који поседују SSH приступ на сервер и поставите огольени репозиторијум негде где сви ти корисницима имају приступ за читање и упис. Спремни сте да кренете — ништа више вам није потребно.

У следећих неколико одељака, видећете како да начините нека софистицирања подешавања. Ова дискусија ће показати и начин којим не морате да креирате корисничке налоге за сваког корисника, затим додавање јавног приступа за читање репозиторијумима, подешавање веб корисничких интерфејса и још тога. Ипак, имајте на уму да вам је за сарадњу неколико људи на приватном пројекту *потребно* да имате само SSH сервер и огольени репозиторијум.

Мали системи

Ако сте мала група људи или једноставно испробавате програм Гит за своју организацију и имате само неколико програмера, ствари за вас могу бити врло једноставне. Један од најкомплекснијих аспеката подешавања Гит сервера је управљање корисницима. Ако желите да неки репозиторијуми неким корисницима буду доступни само за читање, а другима и за читање и за упис, подешавање приступа и дозвола може бити нешто компликованије.

SSH приступ

Ако имате сервер коме сви програмери већ имају SSH приступ, у општем случају је најједноставније да прво тамо поставите први репозиторијум, јер скоро ништа више није потребно да урадите (као што смо видели у претходном одељку). Ако желите дозволе сложенијег типа контроле приступа над репозиторијумима, можете да их обрадите користећи уобичајене дозволе фајл система које поседује оперативни систем вашег сервера.

Ако своје репозиторијуме желите да поставите на сервер који нема налоге за сваку особу из

тима којој желите да доделите права уписа, онда морате да подесите SSH за сваког од њих. Претпостављамо да ако имате сервер на којем ово можете остварити, већ имате инсталиран SSH сервер и да је то начин којим приступате серверу.

Постоји неколико начина на које можете свима из тима дозволити приступ. Први је да подесите налоге за свакога, што је просто, али може бити заморно. Можда не желите да покрећете `adduser` (или могућу алтернативу `useradd`) и да морате постављати привремене шифре за сваког новог корисника.

Друга метода је да на машини креирате јединственог корисника 'git', па да питате сваког корисника који има приступ за упис на сервер да вам пошаље свој јавни SSH кључ и да додате тај кључ у фајл `~/.ssh/authorized_keys` тог новог корисника 'git'. Сада сви могу да приступе машини кроз 'git' налог. Ово ни на који начин не утиче на комитоване податке — SSH корисник који се повезује никако не утиче на комитове које бележите.

Други начин да ово урадите јесте да подесите ваш SSH сервер тако да аутентификацију ради помоћу LDAP сервера или неког другог централизованог извора аутентификације ког сте већ подесили. Све док сваки корисник има приступ машини преко командног окружења, било који механизам за аутентификацију преко SSH који вам пада на памет би требало да функционише.

Генерирање јавног SSH кључа

Многи Гит сервери обављају аутентификацију користећи јавне SSH кључеве. Да би вам доставили своје јавне кључеве, сваки корисник у вашем систему ће морати да генерише један ако га већ немају. Овај процес је сличан у свим оперативним системима. Прво, треба да проверите да ли већ имате кључ. Корисникovi SSH кључеви се подразумевано чувају у `~/.ssh` директоријуму тог корисника. Лако можете проверити да ли већ имате кључ тако што ћете отићи до тог директоријума и излистати његов садржај.

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

Тражите пар фајлова који се зову нешто као `id_dsa` или `id_rsa` и одговарајући фајл са екstenзијом `.pub`. Фајл `.pub` је ваш јавни кључ, а други фајл је приватни. Ако немате ове фајлове (или чак немате ни `.ssh` директоријум), можете да их креирате покретањем програма који се зове `ssh-keygen`, који се испоручује уз SSH пакет на Линукс и Мек системима, а за Виндууз долази уз Гит.

```
$ 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
```

Прво потврђује где желите да сачувате кључ (`.ssh/id_rsa`), а онда вас двапут пита за лозинку, коју треба да оставите празну ако не желите да уносите лозинку када користите кључ. Међутим, ако хоћете да употребите лозинку, обавезно наведите опцију `-o`; он чува приватни кључ у формату који је отпорнији на провалјивање шифре грубом силом у односу на подразумевани формат. Такође можете да користите и `ssh-agent` алат са којим нема потребе да сваки пут уносите шифру.

Дакле, сваки корисник треба да уради ово и да пошаље свој јавни кључ вама или некој особи која је задужена за администрацију Гит сервера (под претпоставком да користите поставку са SSH сервером који захтева јавне кључеве). Потребно је само да копирају садржај `.pub` фајла и да га пошаљу мејлом. Јавни кључ изгледа отприлике овако:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAQEAk1OUpkDHrfHY17SbrmTIpNLTK9Tjom/BWDSU
GPl+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrvjQzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1WXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvS1VK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprgx88XypNDvjYNby6vw/Pb0rwert/E
mZ+AW40ZPnP189ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

За детаљнији туторијал о креирању SSH кључева на разним оперативним системима, погледајте GitHub водич за SSH кључеве на <https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>.

Подешавање сервера

Прођимо кроз подешавање SSH приступа на серверској страни. У овом примеру, користићете `authorized_keys` методу за аутентификацију својих корисника. Претпостављамо и да имате стандардну Линукс дистрибуцију као што је Убунту.



Већи део овога што је приказано у овом одељку може да се аутоматизује употребом команде `ssh-copy-id` command, без потребе да ручно копирате и инсталирате јавне кључеве.

На почетку креирате `git` кориснички налог и `.ssh` директоријум за тог корисника.

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

Затим је потребно да додате неколико јавних SSH кључева програмера у фајл `authorized_keys` корисника `git`. Претпоставимо да имате неке јавне кључеве којима се верује и да сте их сачували у привремене фајлове. Опет, јавни кључеви изгледају некако овако:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpgW1GYEIgS9Ez
Sdf8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtpofwFB1gc+myiv
07TCUSBdLQlgMVOfq1I2uPWQ0kOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgTZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Само их надовежите на постојећи садржај фајла `authorized_keys` корисника `git` у његовом `.ssh` директоријуму:

```
$ 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
```

Сада за њих можете да подесите празан репозиторијум извршавањем команде `git init` уз опцију `--bare`, која иницијализује репозиторијум, а не креира радни директоријум:

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

Џон, Џоси или Џесика тада могу да гурну прву верзију свог пројекта на тај репозиторијум тако што ће га додати као удаљени репозиторијум и гурнути грани на њега. Обратите пажњу на то да неко мора искористити љуску за приступ машини и да направи огњени репозиторијум сваки пут када желите да додате нови пројекат. Користићемо `gitserver` као име хоста сервера на коме сте подесили `git` корисника и репозиторијум. Ако се интерно извршава и подесили сте DNS за `gitserver` тако да показује на тај сервер, команде онда можете да користите на уобичајен начин (под претпоставком да је пројекат `myproject` постоји и садржи неке фајлове):

```
# на џоновом компјутеру
$ 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
```

Сада остали могу да клонирају и гурају промене подједнако лако:

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

Овом методом можете брзо да поставите и покренете Гит сервер са дозволом читања и уписа за неколико програмера.

Треба да обратите пажњу на то да тренутно сви корисници такође могу и да се пријаве на сервер и уђу у љуску као корисник **git**. Ако то желите да спречите, љуску ћете у фајлу **passwd** морати да промените на нешто друго.

git корисника ћете лако моћи да ограничите тако да му буду дозвољене само Гит активности користећи алат ограничене љуске под називом **git-shell** који долази уз програм Гит. Ако ово подесите као логин љуску корисника **git**, он неће имати класичан приступ серверу из љуске. То се ради тако као корисникову логин љуску уместо **bash** или **csh** поставите **git-shell**. Најпре треба да у **/etc/shells** додате име потпуне путање **git-shell** команде ако се тамо већ не налази.

```
$ cat /etc/shells  # погледај да ли је 'git-shell' већ тамо. Ако није...
$ which git-shell  # проверите да је git-shell инсталiran на ваш систем.
$ sudo -e /etc/shells # па додајте из последње команде путању до git-shell
```

Сада можете изменити љуску за корисника командом **chsh <име_корисника> -s <љуска>**:

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

Сада корисник **git** још увек може да користи SSH конекцију да гура и повлачи Гит репозиторијуме, али нема приступ машини преко класичног љуске. Ако покуша, видеће поруку одбијања пријаве сличну овом:

```
$ 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.
```

У овом тренутку корисници још увек могу да користе SSH прослеђивање порта за приступ било ком хосту који је доступан са гит сервера. Ако то желите да спречите, можете да уредите фајл `authorized_keys` и да ставите следеће опције испред сваког кључа који желите да ограничите:

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

Резултат би требало да изгледа овако:

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4LojG6rs6h
PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpgW1GYEIgS9EzSdfd8AcC
IicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv07TCUSBd
LQlgMVOFq1I2uPWQOkOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgtzg2AYYgPqdAv8JggJ
ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQDEwENNMoTboYI+LJieaAY16qiXiH3wuvENhBG...
```

Сада ће мрежне команде програма Гит и даље радити како се очекује, али корисници неће имати приступ љуски. Како излаз налаже, можете да подесите и директоријум у почетном директоријуму корисника `git` који донекле прилагођава команду `git-shell`. На пример, можете ограничiti Гит команде које ће сервер прихватити, или можете изменити поруку коју корисници виде када покушају да се повежу преко SSH. За више информација о начину прилагођавања љуске, извршите `git help shell`.

Гит демон

Сада ћемо подесити демон који сервисира репозиторијуме преко „Гит” протокола. Ово је чест избор за брз неаутентификовани приступ Гит подацима. Запамтите да ово није аутентификовани сервис, и све што сервисирате преко овог протокола ће бити јавно на тој мрежи.

Ако покрећете ово на серверу ван свог фајервола, требало би да га користите само за пројекте који су јавно видљиви свету. Ако је сервер који користите унутар фајервола, можете га користити за пројекте којима велики број људи или рачунара (континуална интеграција или сервери за изградњу) има приступ само за читање, а када не желите да додајете SSH кључ за свакога.

У сваком случају, Гит протокол је релативно једноставан за подешавање. У суштини, следећу команду треба да покренете демонизовану:

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

Опција `--reuseaddr` дозвољава серверу да се рестартије, без потребе за чекањем да старе конекције прво истекну, опција `--base-path` омогућава људима да клонирају пројекат без потребе да наводе комплетну путању, а путања на крају говори Гит демону где да тражи репозиторијуме за извоз. Ако имате укључен фајервол, мораћете и да отворите порт 9418 на машини на којој ово подешавате.

Овај процес можете демонизовати на неколико начина, зависно од тога који оперативни систем извршавате на машини.

Пошто је `systemd` најраспрострањенији систем за иницијализацију у модерним Линукс дистрибуцијама, можете га употребити за ову сврху.

```
[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
```

Можда сте приметили да се овде Гит демон покреће тако да је `git` и група и корисник. Промените име тако да одговара вашим потребама и обезбедите да тај корисник постоји на систему. Такође, проверите да ли се Гит бинарни фајл заиста налази у `/usr/bin/git`, па у случају потребе, промените путању.

На крају, извршићете `systemctl enable git-daemon` чиме се сервис аутоматски покреће приликом подизања система, а сервис можете да покренете и зауставите са `systemctl start git-daemon` и `systemctl stop git-daemon` респективно.

На осталим системима бисте могли да употребите `xinetd`, скрипту у `sysvinit` систему, или нешто друго — штагод вам омогућава да некако демонизујете комаду и да је надгледате.

Затим, програму Гит морате навести којим репозиторијумима треба да дозволи неаутентификован приступ базиран на Гит серверу. То можете да урадите у сваком репозиторијуму тако што креирате фајл под именом `git-daemon-export-ok`.

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

Присуство тог фајла говори програму Гит да је у реду да овај пројекат сервира без аутентификације.

Паметан HTTP

Сада имамо аутентификован приступ кроз SSH и неаутентификован приступ преко `git://`, али постоји и протокол који може радити обе ствари истовремено. Подешавање Паметног HTTP протокола се у принципу своди на укључивање CGI скрипте која долази уз програм Гит под именом `git-http-backend` на серверу.

Овај CGI ће прочитати путању и заглавља које пошаљу `git fetch` или `git push` на HTTP URL и одредити да ли клијент може да комуницира преко HTTP протокола (што је тачно за сваког клијента почевши од верзије 1.6.6). Ако CGI види да је клијент паметан, комуницираће с њим на паметан начин; иначе ће му приступити приглупо (тако да је компатибилан уназад за читање са старијим верзијама клијената).

Хајде да прођемо кроз веома једноставно постављање. Користићемо *Apache* као CGI сервер. Ако немате подешен *Apache*, овако га можете поставити на Линукс машини:

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

Ово такође укључује модуле `mod_cgi`, `mod_alias` и `mod_env` који су неопходни за исправан рад.

Мораћете и да подесите Јуникс групу корисника директоријума `/srv/git` на `www-data` како би ваш веб сервер могао да има приступ за читање и писање репозиторијума, јер ће се инстанца *Apache* сервера која покреће CGI скрипту подразумевано извршавати као тај корисник:

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

Следеће што треба да урадимо јесте да додамо неке ствари у *Apache* конфигурацију како бисмо могли да извршавамо `git-http-backend` као обрађивач свега што дође на `/git` путању вашег веб сервера.

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

Ако изоставите променљиву окружења `GIT_HTTP_EXPORT_ALL`, програм Гит ће онда неаутентификованим клијентима серверати само репозиторијуме које у себи садрже фајл `git-deamon-export-ok`, баш као што је био случај код Гит демона.

За крај, потребно је да *Apache* серверу наложите да дозволи захтеве ка `git-http-backend` скрипти и да на неки начин аутентификује захтеве за упис, на пример са *Auth* блоком на следећи начин:

```
<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>
```

За ово је неопходно да направите `.htpasswd` фајл који садржи лозинке свих важећих корисника. Ево примера додавања корисника „schacon” у тај фајл:

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

Има гомила начина но које *Apache* може да аутентификује кориснике, па ћете морати да изаберете и имплементирате један од њих. Ово је само најједноставнији пример који нам је пао на памет. Скори сигурно ћете ово желети да подесите преко SSL како би сви подаци били шифровани.

Не желимо да се превише упустимо у детаље конфигурације *Apache* сервера, пошто је прилично вероватно да користите и неки други сервер, или имате другачије потребе за аутентификацијом. Идеја је да уз програм Гит долази CGI скрипта који се назива `git-http-backend` и која обавља све неопходне преговоре о слању и примању података преко HTTP протокола када се покрене. Она сама не имплементира било какву аутентификацију, али се то једноставно може контролисати на нивоу веб сервера који је позива. Ово можете имплементирати употребом скори сваког веб сервера који има подршку за CGI, тако да је најбоље да изаберете онај који већ добро познајете.



За више информација о конфигурацији аутентификације у *Apache* серверу, погледајте *Apache* документацију на следећој адреси: <http://httpd.apache.org/docs/current/howto/auth.html>

GitWeb

Сада када имате подешене основне начине приступа пројекту, можда ћете желети да подесите и једноставан визуелизатор базиран на вебу. Гит долази са CGI скриптом *GitWeb* која се понекад користи за ово.

The screenshot shows the GitWeb interface for a repository. At the top, there's a navigation bar with links like 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. On the right, there are buttons for 'commit', 'search' (with a field), and 're'. Below the navigation, there's a section for 'description' (empty), 'owner' (Ben Straub), and 'last change' (Wed, 11 Jun 2014 12:20:23 -0700). The main area is divided into sections: 'shortlog' (listing commits from 2014-06-11 to 2014-06-07), 'tags' (listing tags from v0.21.0-rc1 down to v0.11.0), and 'blob' (empty). Each commit entry includes a link to the commit, shortlog, log, and snapshot.

Слика 49. GitWeb кориснички интерфејс базиран на вебу

Ако желите да погледате како би GitWeb изгледао на вашем пројекту, програм Гит долази са командом која може да подигне привремену инстанцу ако имате лаган сервер на систему као што је `lighttpd` или `webrick`. На Линукс машинама је често инсталiran `lighttpd`, тако да можете да га покренете тако што ћете укуцати `git instaweb` у директоријуму пројекта. Ако користите Мек, Леопард долази са већ инсталираним Рубијем, тако да би требало да имате највише поверења у `webrick`. Да бисте покренули `instaweb` помоћу неког другог обрађивача, а не са `lighttpd`, можете да га покренете помоћу опције `--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]
```

Ово покреће HTTPD сервер на порту 1234 и онда аутоматски покреће веб прегледач који отвара ту страницу. Прилично је једноставно с ваше стране. Када сте завршили разгледање и желите да угасите сервер, можете да покренете исту команду користећи опцију `--stop`:

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

Ако желите да се за ваш тим или за пројекат отвореног кода који хостујете веб интерфејс извршава на серверу све време, мораћете подесите да CGI скрипту сервира ваш уобичајени веб сервер. Неке Линукс дистрибуције имају пакет `gitweb` који можете да инсталирате путем `apt` или `dnf`, па можете прво то да пробате. Убрзо ћемо прећи на ручно инсталирање GitWeb.

Прво, морате да преузмете изворни кôд програма Гит уз који долази GitWeb, па да генеришете CGI скрипту.

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

Обратите пажњу на то да комади морате навести где да пронађе ваше Гит репозиторијуме помоћу променљиве `GITWEB_PROJECTROOT`. Сада треба да подесите *Apache* тако да користи CGI за ту скрипту, за шта можете да додате *VirtualHost*:

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

Да поновимо, GitWeb може да се сервира било којм веб сервером који је способан да извршава CGI или Perl; а чак и ако вам више одговара нешто друго, не би требало да буде превише тешко за подешавање. Сада би требало да можете да посетите <http://gitserver/> и да на мрежи погледате своје репозиторијуме.

GitLab

Ипак је GitWeb прилично једноставан. Ако вам треба модернији Гит сервер са пуним могућностима, постоји неколико решења отвореног кода које можете да инсталirate уместо њега. Пошто је GitLab један од популарнијих, покрићемо његову инсталацију и коришћење као пример. Ово је мало сложеније од GitWeb опције и захтеваће више напора око одржавања, али због тога пружа потпуне могућности.

Инсталација

GitLab је веб апликација која се ослања на базу података, тако да је њена инсталација мало компликованија од инсталације неких других Гит сервера. Срећом, процес је доста добро

документован и подржан. GitLab топло препоручује да GitLab инсталirate на сопствени сервер помоћу званичног Omnibus GitLab пакета.

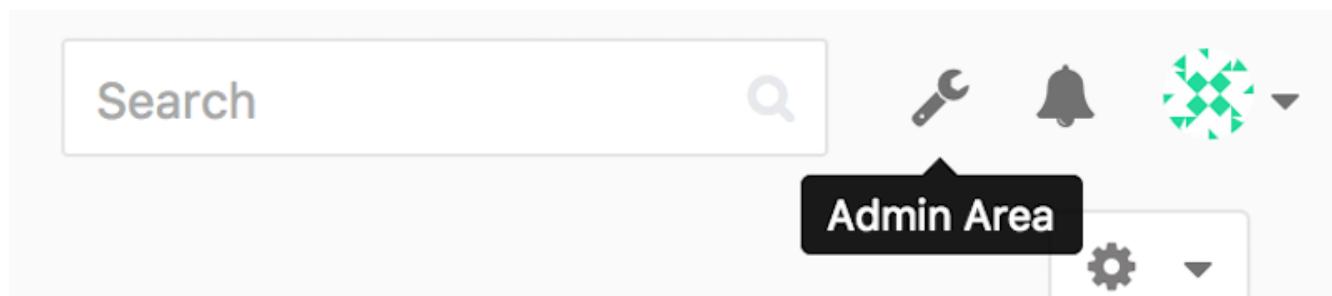
Остале опције за инсталацију су:

- GitLab Helm карта, за употребу са *Kubernetes*.
- Докеризовани GitLab пакети за употреби са програмом *Docker*.
- Фајлови извornог кода.
- Провајдери клауд услуга као што су *AWS*, *Google Cloud Platform*, *Azure*, *OpenShift* и *Digital Ocean*.

За више информација прочитајте [GitLab Community Edition \(CE\) 'прочитајме' фајл](#).

Администрација

GitLab интерфејсу за администрацију се приступа преко веба. Једноставно усмерите свој интернет прегледач на име хоста или IP адресу на коју је GitLab инсталариран и пријавите се као администратор. Подразумевано корисничко име је `admin@local.host`, а подразумевана лозинка је `5iveL!fe` (коју морате да промените чим је укуцате). Када се пријавите, кликните на иконицу „Admin area” у горњем десном углу.



Слика 50. „Admin area” и GitLab менију

Корисници

Свако ко користи ваш GitLab сервер мора да поседује кориснички налог. Кориснички налоги су заиста једноставни, садрже углавном личне информације везане за податке за пријаву. Сваки кориснички налог има **простор имена**, односно логичко груписање пројектата који припадају том кориснику. Ако корисник јави има пројекат под именом project, URL адреса тог пројекта би била <http://server/jane/project>.

Name	Action
Administrator Admin It's you! admin@example.com	Edit
Betsy Rutherford II marlin@lednerlangworth.biz	Edit
Brenden Hayes laney_dubuque@cormier.biz	Edit
Cassandra Kilback caterina@beer.com	Edit
Cathryn Leffler DVM desmond@crooks.ca	Edit
Cecil Medhurst winnifred@glover.co.uk	Edit
Dr. Joany Fisher milan@hueels.us	Edit
Jazmin Sipes juliet.turner@leannnon.co.uk	Edit

Слика 51. GitLab екран за администрацију корисника

Уклањање корисничког налога се може обавити на два начина: „Блокирањем” се корисник спречава да се пријави на GitLab инстанцу, али се задржавају сви подаци под простором имена тог корисника, а комитови потписани мејл адресом тог корисника и даље имају линк до његовог профила.

С друге стране, „ уништавање ” корисника га у потпуности брише из базе података и фајл система. Уклањају се сви пројекти и подаци у његовом простору имена, као и све групе које поседује. Ово је очигледно много трајнија и деструктивнија операција која ће вам ретко бити потребна.

Групе

GitLab група је скуп пројектата заједно са подацима о начину на којем корисници могу да приступе тим пројектима. Свака група има свој простор имена (као што га имају и корисници), тако да ако група training има пројекат materials, његова URL ће бити <http://server/training/materials>.

The screenshot shows the GitLab.org group administration interface. At the top, there's a navigation bar with 'Group' selected, followed by 'Activity', 'Labels', 'Milestones', 'Issues 8,501', 'Merge Requests 701', 'Members', and 'Contribution Analytics'. A search bar says 'This group Search'. Below the navigation is the group logo (@gitlab-org) and the tagline 'Open source software to collaborate on code'. There are buttons for 'Leave group' and 'Global'. The main area shows a list of projects under 'All Projects': 'GitLab Development Kit', 'kubernetes-gitlab-demo', 'omnibus-gitlab', 'GitLab Enterprise Edition', 'gitlab-shell', and 'gitlab-ci-multi-runner'. Each project has a small icon, a name, a description, and two circular status indicators.

Слика 52. GitLab екран за администрацију група

Свака група је придружена одређеном броју корисника, при чему сваки од њих има ниво дозволе за групне пројекте као и за саму групу. Ове дозволе се крећу од „Гост” (само за приступ тикетима и чету) до „Власник” (потпуна контрола над групом, њеним члановима и пројектима). Постоји превише типова дозвола да би се сви овде навели, али GitLab има користан линк на екрану за администрацију.

Пројекти

GitLab пројекат грубо одговара једном Гит репозиторијуму. Сваки пројекат припада једном простору имена, било кориснику или групи. Ако пројекат припада кориснику, власник пројекта има непосредну контролу над тиме ко има приступ пројекту; ако пројекат припада групи, користиће се групне дозволе за кориснички ниво.

Сваки пројекат има и ниво видљивости који контролише ко има приступ читања страница и репозиторијума тог пројекта. Ако је пројекат *Private*, власник пројекта мора експлицитно да дозволи приступ одређеним корисницима. *Internal* пројекат је видљив свим пријављеним корисницима, а *Public* свима. Обратите пажњу на то да ово контролише и **git fetch** приступ, као и приступ веб корисничком интерфејсу тог пројекта.

Куке

GitLab поседује и подршку за куке, како на нивоу пројекта, тако и на нивоу система. У оба случаја, кад год се дододи неки релевантан догађај, GitLab сервер ће обавити HTTP POST са неким описним JSON. Ово је одличан начин да повежете своје Гит репозиторијуме и GitLab инстанцу са остатком аутоматизације тока развоја, као што су CI сервери, чет собе, или алати за развој.

Основна употреба

Прва ствар коју ћете хтети да урадите са GitLab је да креирате нови пројекат. Ово се

једноставно обавља кликом на иконицу „+” из траке са алатима. Поставиће вам се питање за име пројекта, ком простору имена треба да припада и који би требало да буде ниво видљивости. Већина онога што овде дефинишете није трајно и може се накнадно променити путем интерфејса за подешавања. Кликните на „Create project” и то је то.

Сада када пројекат постоји, вероватно ћете хтети да га повежете са локалним Гит репозиторијумом. Сваки пројекат је доступан преко HTTPS или SSH и оба се могу користити за конфигурисање удаљеног Гит репозиторијума. URL адресе су видљиве на врху насловне странице пројекта. За већ постојећи локални репозиторијум, следећа команда ће креирати удаљени репозиторијум `gitlab` везан за хостовану локацију:

```
$ git remote add gitlab https://сервер/простор_имена/пројекат.git
```

Ако немате локалну копију репозиторијума, можете једноставно да урадите следеће:

```
$ git clone https://сервер/простор_имена/пројекат.git
```

Веб кориснички интерфејс нуди неколико корисних погледа на сам репозиторијум. Почетна страница сваког пројекта показује скорашињу активност, а линкови при врху ће вас одвести до погледа на фајлове пројекта и лог комитова.

Заједнички рад

Најједноставнији начин заједничког рада на GitLab пројекту је давање сваком кориснику дозволу за гурање на Гит репозиторијум. Корисника можете додати у пројекат одласком у одељак „*Members*” у подешавањима тог пројекта и повезивањем новог корисника са нивоом приступа (о различitim нивоима приступа се дискутује [Групе](#).) Ако кориснику дате ниво приступа „*Developer*” или виши, тај корисник ће моћи да гура комитове и гране директно у репозиторијум.

Још један развојенији начин сарадње је помоћу захтева за спајањем. Ова могућност дозвољава да било који корисник који види пројекат може и да му доприноси на контролисани на начин. Корисници са директним приступом могу једноставно да направе грану, гурну комитове на њу, и отворе захтев за спајањем из њихове гране на грану `master`, или на било коју другу грану. Корисници који немају дозволу за гурање на репозиторијум могу да га „форкују“ (kreирају своју копију), гурну комитове на своју копију, отворе захтев за спајањем из своје рапче назад на главни пројекат. Овај модел дозвољава власнику да има потпуну контролу над тиме шта иде у репозиторијум и када, а допушта да допринос пројекту дају и корисници који немају поверење.

Захтеви за спајањем и тикети су главне јединице за дугорочне GitLab дискусије. Сваки захтев за спајањем допушта да се о предложеној изменама расправља линија по линију (што подржава једноставну врсту прегледа кода), као и општу дискусију о целокупној промени. Оба могу да се доделе корисницима, или да се организују у прекретнице.

Овај одељак се највише усредсредио на GitLab могућности које имају везе са програмом Гит, али будући да је зрео пројекат, GitLab нуди и многе друге могућности које ће вам помоћи у

раду са тимом, као што су викији пројекта и алати за одржавање система. Једна од предности GitLab система је то што када се сервер једном подигне и покрене, ретко ћете имати потребу да мењате конфигурациони фајл или да приступате серверу преко SSH; већина административних и општих задатака може се обавити преко интерфејса из интернет прегледача.

Опције за хостовање које нуде трећа лица

Ако не желите да пролазите кроз свак посао постављања сопственог Гит сервера, имате неколико опција за хостовање својих Гит пројеката на спољним хостинг сајтовима намењеним за то. Овакав приступ има многе предности: хостинг сајт се обично лако поставља и започињање нових пројеката је врло једноставно, а нема ни муке око одржавања сервера. Чак и ако интерно поставите сопствени сервер, можда ћете и поред тога пожелети да за свој отворени код користите јавни хостинг сајт — заједница отвореног кода ће вас тако лакше наћи и помоћи вам.

Ових дана постоји велики број опција за хостовање које можете да изаберете, од којих свака има своје предности и мане. Ако желите да погледате актуелну листу, пођите на [GitHosting](https://git.kernel.org/index.php/GitHosting) страницу са главног Гит викија на <https://git.kernel.org/index.php/GitHosting>.

Коришћење *GitHub* сервиса ћемо детаљније прећи у [GitHub](#), јер је то највећи Гит хост који постоји и вероватно ћете свакако морати да имате интеракцију са пројектима који су тамо хостовани, али постоји још на десетине опција које можете да изаберете у случају да не желите да подешавате сопствени Гит сервер.

Резиме

Доступно вам је неколико опција за постављање и покретање удаљеног Гит репозиторијума помоћу којег можете да сарађујете са другима, или да делите свој рад.

Одржавање сопственог сервера вам даје доста контроле и омогућава вам да сервер покрећете унутар сопственог фајервола, али такав сервер захтева доста вашег времена за подешавање и одржавање. Ако своје податке сместите на хостовани сервер, подешавање и одржавање ће бити једноставно; међутим, мораћете да држите свој код на туђим серверима, а неке организације то не дозвољавају.

Требало би да буде прилично једноставно да одлучите које решење или комбинације решења је прикладна за вас и вашу организацију.

Дистрибуирани Гит

Сада када имате подешен удаљен Гит репозиторијум као централну тачку на којој сви програмери деле свој код и пошто сте упознати са основним командама програма Гит у локалном току рада, време је да представимо начин за искоришћавање неких од дистрибуираних токова рада које вам нуди програм Гит.

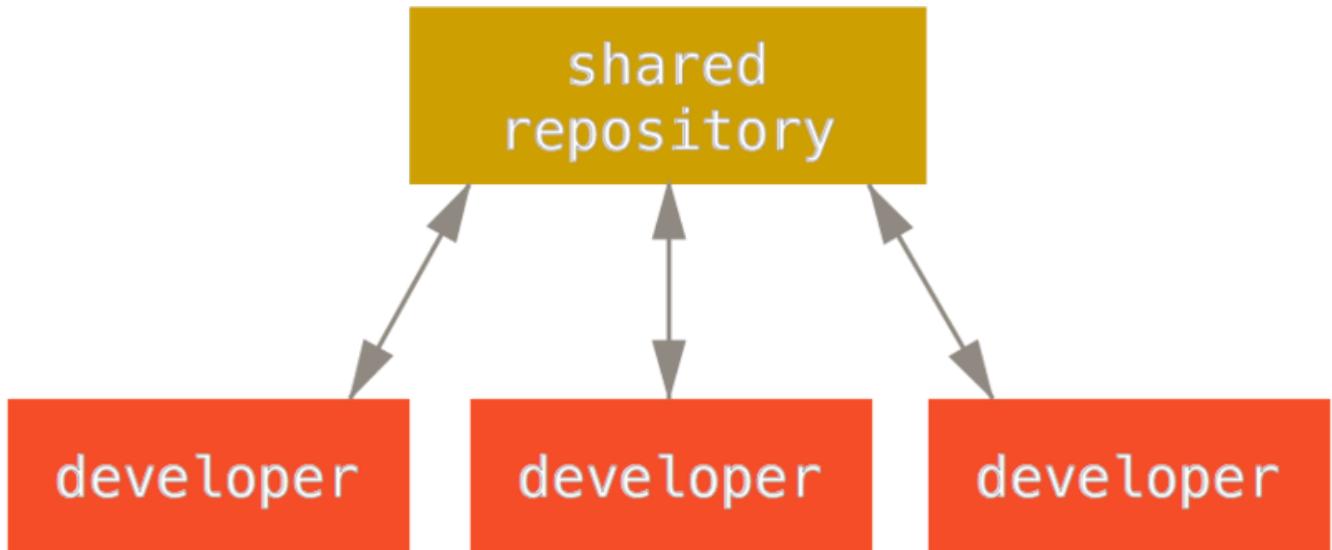
У овом поглављу ћете видети како да радите са програмом Гит у дистрибуираном окружењу као сарадник и као интегратор. Тачније, научићете како да успешно допринесете код пројекту и како да себи и одржаваоцу пројекта олакшате посао што је више могуће, као и како да успешно одржавате пројекат са већим бројем програмера који доприносе садржају.

Дистрибуирани процеси рада

За разлику од централизованих система за контролу верзије (CVCS), дистрибутивна природа програма Гит вам омогућава да будете много флексибилнији по питању начина на који програмери сарађују на пројектима. Код централизованих система, сваки програмер је чвор који ради мање више исто као и централни хаб. Међутим, у програму Гит сваки програмер је потенцијално и чвор и хаб — односно, сваки програмер може и да даје допринос коду у другим репозиторијумима и да одржава јавни репозиторијум према коме други могу да базирају свој рад и коме могу да дају допринос. Ово отвара огроман спектар могућности за процес рада вашег пројекта и/или вашег тима, тако да ћemo прећи неколико честих парадигми које користе предност ове прилагодљивости. Представићемо предности и могуће мане сваког дизајна; можете да изаберете само један од њих који ћете користити, или можете да помешате могућности сваког од њих.

Централизован процес рада

У централизованим системима, у општем случају постоји јединствен модел сарадње — централизовани процес рада. Један централни хаб, или *репозиторијум*, може да прихвати код, а сви синхронизују свој рад са њим. Већи број програмера су чворови — потрошачи тог хаба — и синхронизују се са том централизованим местом.



Слика 53. Централизовани процес рада

Ово значи да ако два програмера клонирају са хаба и обојица направе промене, први програмер који гурне своје промене назад на хаб може то да уради без проблема. Други програмер мора да споји рад првог у свој пре него што гурне узведно своје промене, како не би преписао промене које је направио први програмер. Овај концепт је важи у програму Гит исто као и у *Subversion* (или у било ком другом CVCS), а овај модел ради савршено добро у програму Гит.

Ако сте се већ навикли на централизовани процес рада у својој компанији или тиму, лако можете наставити да користите тај процес рада са програмом Гит. Једноставно подесите један репозиторијум, дајте свим члановима тима дозволу за гурање на њега; програм Гит неће дозволити да корисници пишу преко туђих промена.

Рецимо да Марко и Милица почињу да раде истовремено. Марко завршава своју промену и гура је на сервер. Онда Милица покушава да гурне своје промене, али сервер их одбија. Сервер јој каже да покушава да гурне промене које не могу да се премотају унапред и да то неће моћи да уради све док не преузме податке и не споји их са својим. Овај процес рада је привлачан многим људима јер је то парадигма која им је позната и која им одговара.

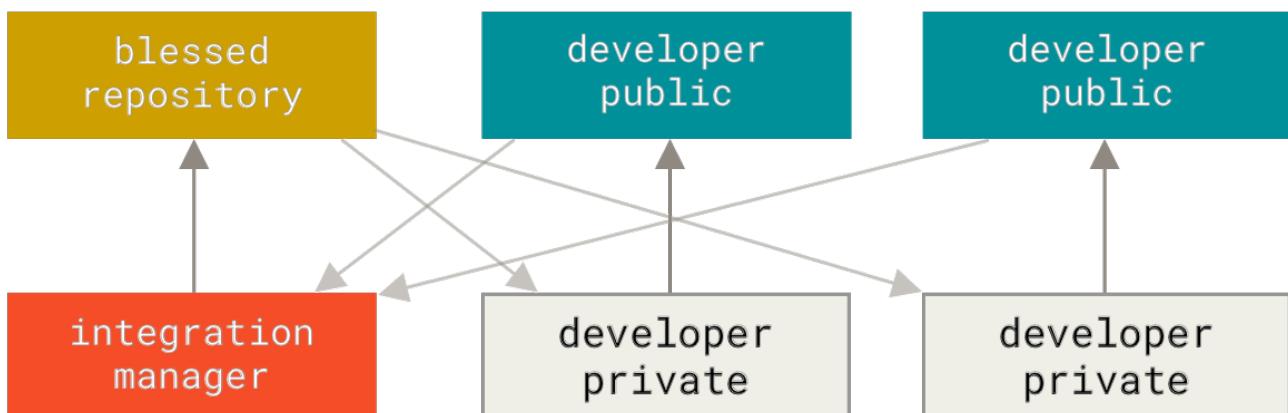
Ово није ограничено само на мале тимове. Са Гитовим моделом гранања је могуће да на стотине програмера успешно ради на једном пројекту користећи истовремено на десетине грана.

Процес рада са руководиоцем интеграције

Пошто вам програм Гит допушта да имате више удаљених репозиторијума, могуће је користити процес рада у коме сваки програмер има приступ уписа у сопствени јавни репозиторијуму и приступ читања свих осталих. Овакав сценарио често укључује и канонички репозиторијум који представља „званичан“ пројекат. Ако желите да дате допринос таквом пројекту, треба да направите сопствени јавни клон пројекта и на њега гурате измене. Онда можете да пошаљете захтев одржаваоцу главног пројекта да повуче ваше измене. Одржавалац онда може да дода ваш репозиторијум као удаљени репозиторијум, локално тестира ваше промене, споји их у сопствену грану, па онда гурне назад на свој репозиторијум. Овај процес ради на следећи начин (погледајте [Процес рада са](#)

[руководиоцем интеграције](#)):

1. Одржавалац пројекта гурне промене на свој јавни репозиторијум.
2. Особа која даје допринос клонира тај репозиторијум и прави промене.
3. Особа која даје допринос гура промене на своју личну јавну копију.
4. Особа која даје допринос шаље одржаваоцу имејл са молбом да повуче промене.
5. Одржавалац додаје репозиторијум особе која даје допринос као удаљени репозиторијум и спаја локално.
6. Одржавалац гура спојене промене на главни репозиторијум.



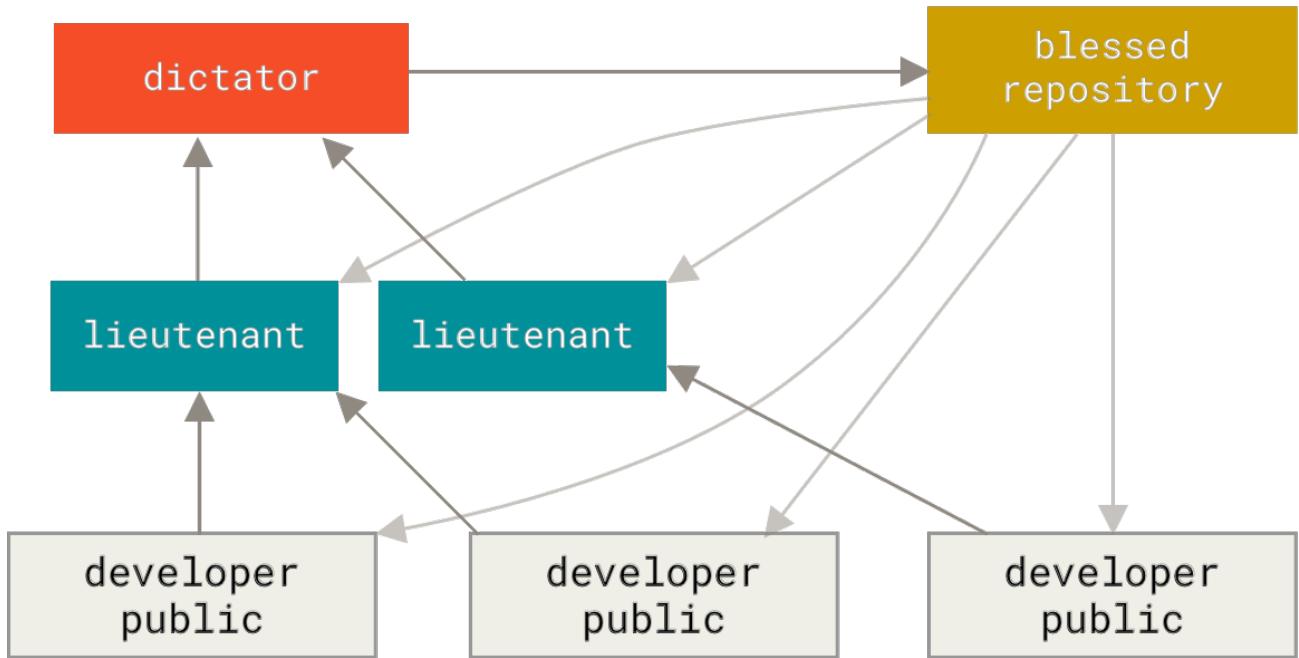
Слика 54. Процес рада са руководиоцем интеграције

да Ово је веома чест процес рада са алатима који су базирани на хабовима као што је GitHub или GitLab, када је лако рачвати пројекат и гурнути промене у своју рачву које ће сви видети. Једна од главних предности овог приступа је то што можете да наставите са својим радом, а одржавалац главног репозиторијума може да повуче ваше промене у било ком тренутку. Особе које дају допринос не морају чекати да пројекат прихвати њихове промене — свако може да ради својим темпом.

Процес рада са диктатором и поручницима

Ово је варијанта процеса рада са више репозиторијума. У општем случају га користе огромни пројекти са стотинама сарадника; један познати пример је Линукс кернел. Више руководиоца интеграције су задужени за одређене делове репозиторијума; они се називају *поручници*. Сви поручници имају једног руководиоца интеграцијом који се назива благонаклони диктатор. Благонаклони диктатор повлачи са њихових репозиторијума у референтни репозиторијум из ког сви сарадници морају да повлаче. Овај процес ради на следећи начин (погледајте [Процес рада са благонаклоним диктатором](#)):

1. Обични програмери раде на својим тематским гранама и ребазирају свој рад на врх `master` гране. `master` грана је у референтном репозиторијуму у који гура диктатор.
2. Поручници спајају тематске гране програмера у своју `master` грану.
3. Диктатор спаја `мастар` гране поручника у диктаторову `master` грану.
4. Напокон, диктатор гура своју `master` грану на референтни репозиторијум тако да остали програмери могу да га ребазирају.



Слика 55. Процес рада са благонаклоним диктатором

Овакав процес рада није уобичајен, али може да буде користан код великих пројекта, или у окружењима у којима је хијерархија јако изражена. Дозвољава да вођа пројекта (диктатор) делегира велики део посла и сакупља велике подскупове кода са више места пре него што их интегрише.

Резиме процеса рада

То су били неки често коришћени процеси рада које је могуће користити у дистрибуираном систему као што је Гит, али јасно је да постоје многе варијације које одговарају вашем одређеном процесу рада у пракси. Сада када (надамо се) можете да одлучите која комбинација процеса рада вам одговара, приказаћемо неке специфичније примере начина на које можете да постигнете главне улоге које чине различите процесе рада. У следећем одељку ћете сазнати нешто о неколико честих образца по којима се доприноси пројекту.

Како се даје допринос пројекту

Главни проблем код описивања начина на који се даје допринос пројекту је то што постоји велики број варијација на који се то може учинити. Пошто је програм Гит веома флексибилан, људи могу да раде на много начина, па је проблематично описати како треба да дате допринос — сваки пројекат је помало другачији. Неки од фактора који су укључени у то је број људи који активно дају доприносе, изабрани процес рада, приступ вашим комитовима, и евентуална метода за спољне доприносе.

Први фактор је број људи који активно учествују — колико корисника активно даје допринос коду овог пројекта и колико често. У многим случајевима, имаћете два или три програмера са неколико комитова на дан, или можда и мање за не тако активне пројекте. За веће компаније или пројекте, број програмера би могао да пређе неколико хиљада, са стотинама или хиљадама комитова на дан. Ово је важно јер са више програмера можете да нађете на више проблема, јер код неће увек моћи да се глатко споји. Промене које

поднесете би могле да застаре или буду прегажене радом који је неко спојио док сте ви радили, или док су ваше промене чекале да се одобре или примене. Како да одржите свој код константно актуелним, а комитове валидним?

Следећи фактор је процес рада који се примењује у пројекту. Да ли је централизован, тако да сваки програмер има једнак приступ за писање у главној линији кода? Да ли пројекат има одржаваоца или руководиоца интеграцијом који проверава све закрпе? Да ли се све закрпе прегледају и одобравају? Да ли сте ви укључени у тај процес? Да ли је примењен систем са поручницима, и да ли морате прво њима да пошаљете свој рад?

Следећи фактор је ваш комит приступ. Неопходни процес рада за давање доприноса пројекту је много другачији ако имате приступ писања пројекту него ако га немате. Ако немате приступ писања, како се одлучује да ли ће ваш рад бити прихваћен у пројекат? Да ли уопште постоји таква политика? Колико ствари мењате у једном тренутку? Колико често дајете допринос?

Сва ова питања могу да утичу на то колико ефикасно дајете допринос пројекту и какви процеси рада су пожељни или доступни вама. Покрићемо аспекте свих ових фактора у низу случајева коришћења, почевши од једноставних ка компликованијим; требало би да на основу ових примера будете у стању да конструишете одређене процесе рада који су вам потребни у пракси.

Смернице за комитове

Пре него што пређемо на одређене случајеве коришћења, ево неких кратких напомена о комит порукама. Поседовање добре смернице за креирање комитова и њено поштовање ће у великој мери учинити рад са програмом Гит и сарадњу са осталима много једноставнијом. Гит пројекат обезбеђује документ који излаже бројне добре савете за креирање комитова које треба употребити за слање закрпи — можете да их прочитате у извornом коду програма Гит у фајлу [Documentation/SubmittingPatches](#).

У првом реду, не желите да рад који пошаљете поседује грешке у вези празних карактера. (*whitespace errors*). Програм Гит вам обезбеђује једноставан начин да ово проверите — пре него што комитујете, извршите `git diff --check`, што ће вам идентификовати и приказати евентуалне грешке у вези празних карактера.

```
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)
```

Слика 56. Излаз команде `git diff --check`.

Ако ову команду покренете пре комитовања, видећете има ли грешака у вези празних карактера које бисте комитовали и које би сметале другим програмерима.

Затим, потрудите се да сваки комит буде логички издвојен скуп промена. Ако можете, пробајте да промене учините лако сварљивим — немојте да кодирате цео викенд радећи на пет различитих проблема, па да их онда у понедељак све стрпате у један огроман комит. Чак и ако не комитујете током викенда, употребите стејџ у понедељак тако да поделите свој рад на барем један комит по проблему који сте решавали, са корисним комит порукама. Ако неке промене модификују исти фајл, пробајте да употребите `git add --patch` да парцијално стејџујете фајлове (детаљније је обрађено у [Интерактивно стејџовање](#)). Снимак пројекта на врху гране је исти било да комитујете једном или пет пута, само је потребно да у неком тренутку све промене буду додате, зато се трудите да олакшате посао својим колегама програмерима када буду морали да прегледају ваше промене.

Овај приступ такође олакшава да се касније неки скуп измена издвоји или да се врати на неко од пређашњих стања. [Поновно исписивање историје](#) описује бројне корисне Гит трикове за преписивање историје и интерактивно стејџовање фајлова — употребите ове алате за изградњу чисте и јасне историје пре него што свој рад пошаљете другима.

Последња ствар коју треба да имате на уму је комит порука. Стицање навике да креирате квалитетне комит поруке учиниће коришћење програма Гит и сарадњу са другима много лакшом. Као опште правило, поруке треба да почну једном линијом која није дужа од око 50 карактера и која концизно описује скуп промена, за којом следи празна линија, а затим детаљније објашњење. Гит пројекат захтева да детаљније објашњење садржи и мотивацију за промену и да искаже измену понашања који њена примена уводи у односу на претходно понашање — ову смерницу је добро пратити. Такође је добра идеја да комит поруку пишете у императиву: „Fix bug“ а не „Fixed bug“ или „Fixes bug“. Другим речима, издајте наредбе. Ево шаблона који можете да следите, то је донекле изменењена верзија коју је [оригинално написао Тим Поул](#):

Capitalized, short (50 chars or less) summary

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

Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like `git merge` and `git revert`.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, followed by a single space, with blank lines in between, but conventions vary here
- Use a hanging indent

Ако све ваше комит поруке прате овај узор, ствари ће бити много једноставније и за вас и за програмере са којима радите. Гит пројекат има добро форматиране комит поруке — покушајте тамо да покренете `git log --no-merges` и погледајте како изгледа лепо форматирана историја комитова у пројекту.

Радите како вам кажемо, а не како ми радимо.



У циљу краћег изражавања, многи промери у овој књизи немају лепо форматиране комит поруке; уместо тога просто користимо `-m` опцију уз команду `git commit`.

Укратко, радите како вам кажемо, а не како ми радимо.

Мали приватни тим

Најједноставнија поставка на коју ћете вероватно наићи је приватни пројекат са једним или два друга програмера. У овом контексту „приватни” значи да је затвореног кода — није доступан остатку света. Ви и сви остали програмери имате дозволу да гурате промене на репозиторијум.

У оваквом окружењу, можете да следите процес рада сличан оном који се користи када употребљавате *Subversion* или неки други централизован систем. И даље добијате предности због ствари као што су комитовање ван мреже и знатно једноставније гранање и спајање, али процес рада може да буде веома сличан; основна разлика је то што се у време комитовања спајања раде на страни клијента уместо на страни сервера. Хајде да видимо како би ствар могла да изгледа када два програмера почну заједно да раде на дељеном репозиторијуму. Први програмер, Џон, клонира репозиторијум, направи измену и комитује

локално. У овим примерима су поруке протокола замењене са `...` како би се донекле скратиле.

```
# Џонова машина
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(-)
```

Други програмер, Џесика, ради исту ствар — клонира репозиторијум и комитује промену:

```
# Џесикина машина
$ 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(-)
```

Сада Џесика гура свој рад на сервер и то функционише како треба:

```
# Џесикина машина
$ git push origin master
...
To jessica@githost:simplegit.git
 1edee6b..fbff5bc  master -> master
```

Последња линија претходног исписа приказује корисну поруку коју враћа операција гурања. Основни формат је `<старареф>..<новареф>` изреф → уреф, где **старареф** значи стара референца, **новареф** значи нова референца, **изреф** је име локалне референце која се гура и **уреф** је име удаљене референце која се ажурира. Даље у дискусији ћете видети испис сличан овом, тако да ће вам поседовање основне идеје значења помоћи да разумете разна стања у којима могу да се нађу репозиторијуми. Више детаља можете пронаћи у документацији команде [git-push](#).

Даље у овом примеру, мало касније Џон прави неке измене које комитује у свој локални репозиторијум и покушава да их турне на исти сервер:

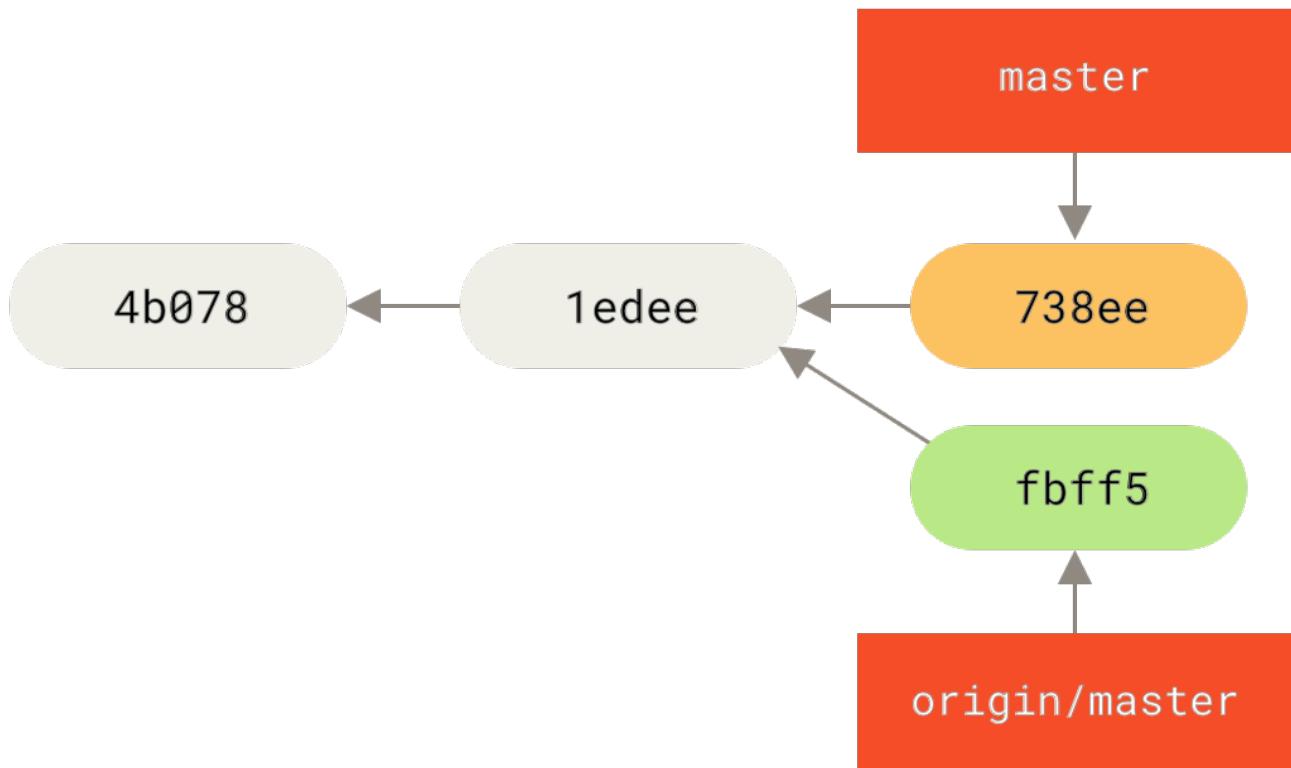
```
# Џонова машина
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]      master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

У овом случају Џону није дозвољено да гурне своје промене јер је већ раније Џесика гурнула своје. Ово је посебно важно разумети ако сте навикли на *Subversion*, јер ћете приметити да њих двоје нису уређивали исти фајл. Мада *Subversion* аутоматски на серверу врши спајање ако су уређивани различити фајлови, и програму Git *прво* морате да спојите комитове локално. Другим речима, Џон најпре мора да преузме Џесикине узводне промене и да их споји са својим локалним репозиторијумом пре него што му буде дозвољено да гурне своје.

Како први корак, Џон преузима Џесикин рад (ово само *преузима* Џесикин узводни рад, још увек га не спаја са Џоновим радом):

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

У овом тренутку би Џонов локални репозиторијум требало да изгледа некако овако:

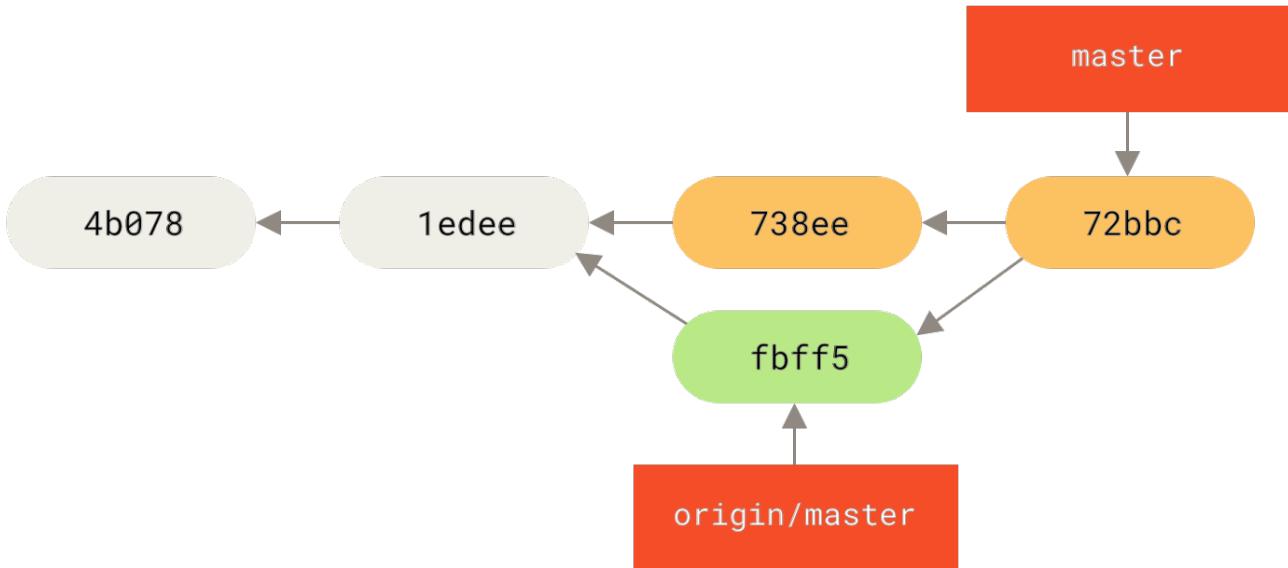


Слика 57. Џонова разграната историја

Сада Џон може да споји преузети Џесикин рад са својим локалним радом:

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

Све док спајање пролази глатко, Џонова ажурирана историја комитова ће сада изгледати овако:

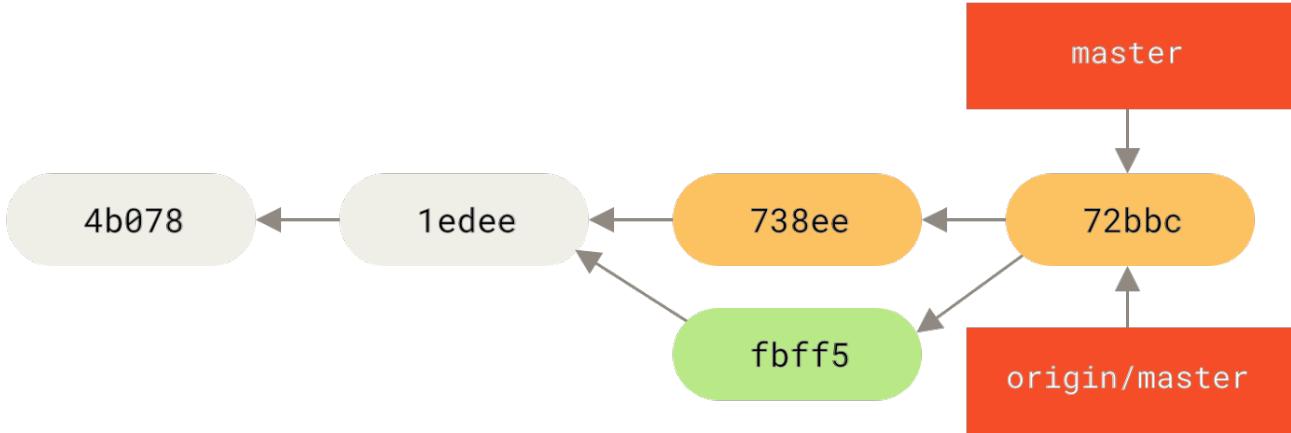


Слика 58. Џонов репозиторијум после спајања `origin/master`

Сада би Џон могао да тестира овај нови кôд како би потврдио да Џесикин рад уопште не утиче на његов, па ако све изгледа у реду, онда да напокон гурне свој ново спојени рад на сервер:

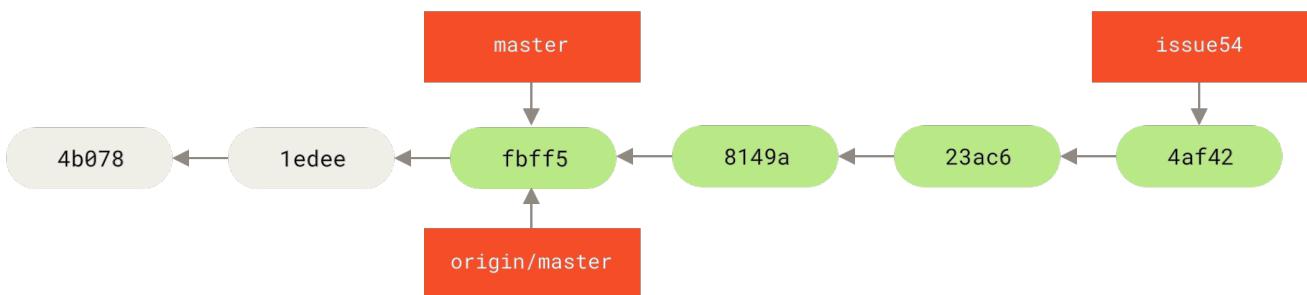
```
$ git push origin master
...
To john@githost:simplegit.git
 fbff5bc..72bbc59  master -> master
```

Коначно, Џонова istorija komitova ће izgledati ovako:



Слика 59. Џонова историја комитова после гурања на `origin` сервер

У међувремену, Џесика је креирала је тематску грану `issue54` и направила три комита на тој грани. Још увек није преузела Џонове промене, тако да њена историја комитова изгледа овако:



Слика 60. Џесикина тематска грана

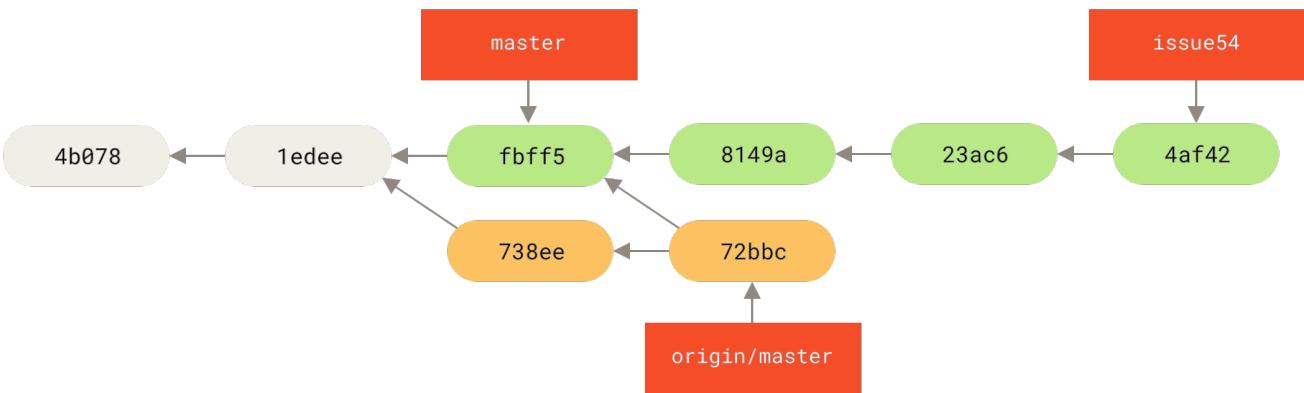
Џесика одједном схвата да је Џон гурнуо нов рад на сервер и жели да га погледа, тако да са сервера може да преузме сви садржај који још увек нема:

```

# Џесикина машина
$ git fetch origin
...
From jessica@githost:simplegit
  fbf5bc..72bbc59  master      -> origin/master

```

Ово повлачи рад који је Џон у међувремену гурнуо. Сада Џесикина историја изгледа овако:



Слика 61. Џесикана историја после преузимања Џонових промена

Џесика мисли да је њена тематска грана спремна, али жељи да сазна који део Џоновог преузетог посла треба да споји у свој рад тако да би могла да гурне промене. Извршава команду `git log` да сазна:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    Remove invalid default value
```

Синтакса `issue54..origin/master` је филтер за лог који од програма Гит тражи да покаже само листу комитова који су на другој грани (у овом случају `origin/master`) а нису на првој грани (у овом случају `issue54`). Детаљније ћемо обрадити ову синтаксу у [Опсези комитова](#).

У горњем излазу видимо да постоји само један комит који је Џон направио, а који Џесика није спојила у свој локални рад. Ако споји `origin/master`, то је једини комит који ће променити њен локални рад.

Сада Џесика може да споји своју тематску грану у своју `master` грану, да споји Џонов рад (`origin/master`) у своју `master` грану, па да онда поново гурне промене на сервер.

Најпре (након што је комитовала сви рад на својој `issue54` тематској грани), Џесика скоче назад на своју `master` грану у циљу припреме за интеграцију свега што је урађено:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

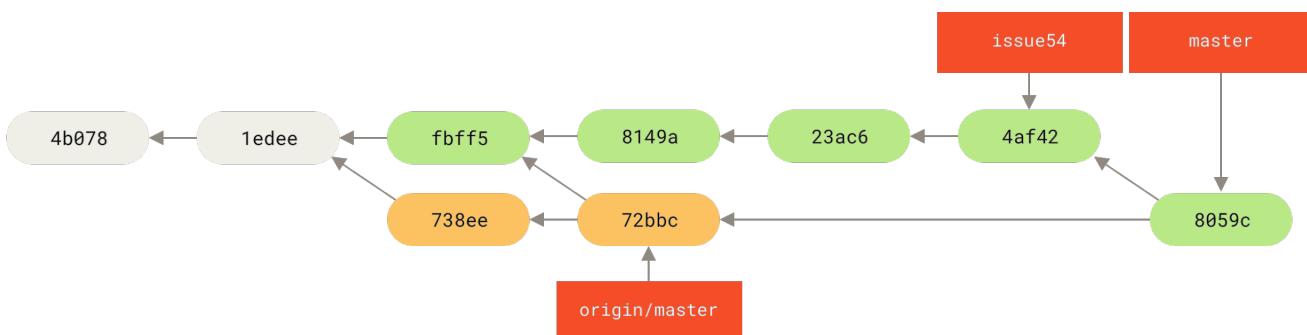
Џесика може прво да споји било `origin/master` или `issue54`—обе се налазе узводно, па редослед спајања није важан. Крајњи снимак би требало да буде идентичан без обзира на то који редослед изабере; само ће историја бити донекле другачија. Бира да прво споји `issue54` грану:

```
$ git merge issue54
Updating fbf5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |   6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Не јављају се проблеми; као што се види, радио се о једноставном спајању премотавањем унапред. Џесика сада довршава процес локалног спајања тако што спаја Џонов рад који раније преузела и који се налази у `origin/master` грани:

```
$ 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(-)
```

Све се чисто спаја и сада Џесикина историја изгледа овако:

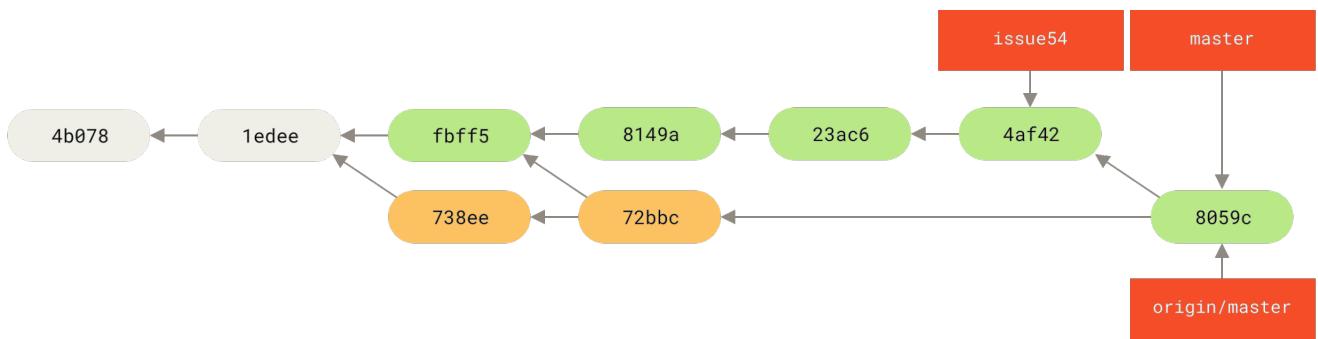


Слика 62. Џесикина историја након спајања Џонових промена

Сада је `origin/master` доступна из Џесикине `master` грани, па би требало да је стању успешно да гурне промене (под претпоставком да Џон у међувремену није гурнуо још промена):

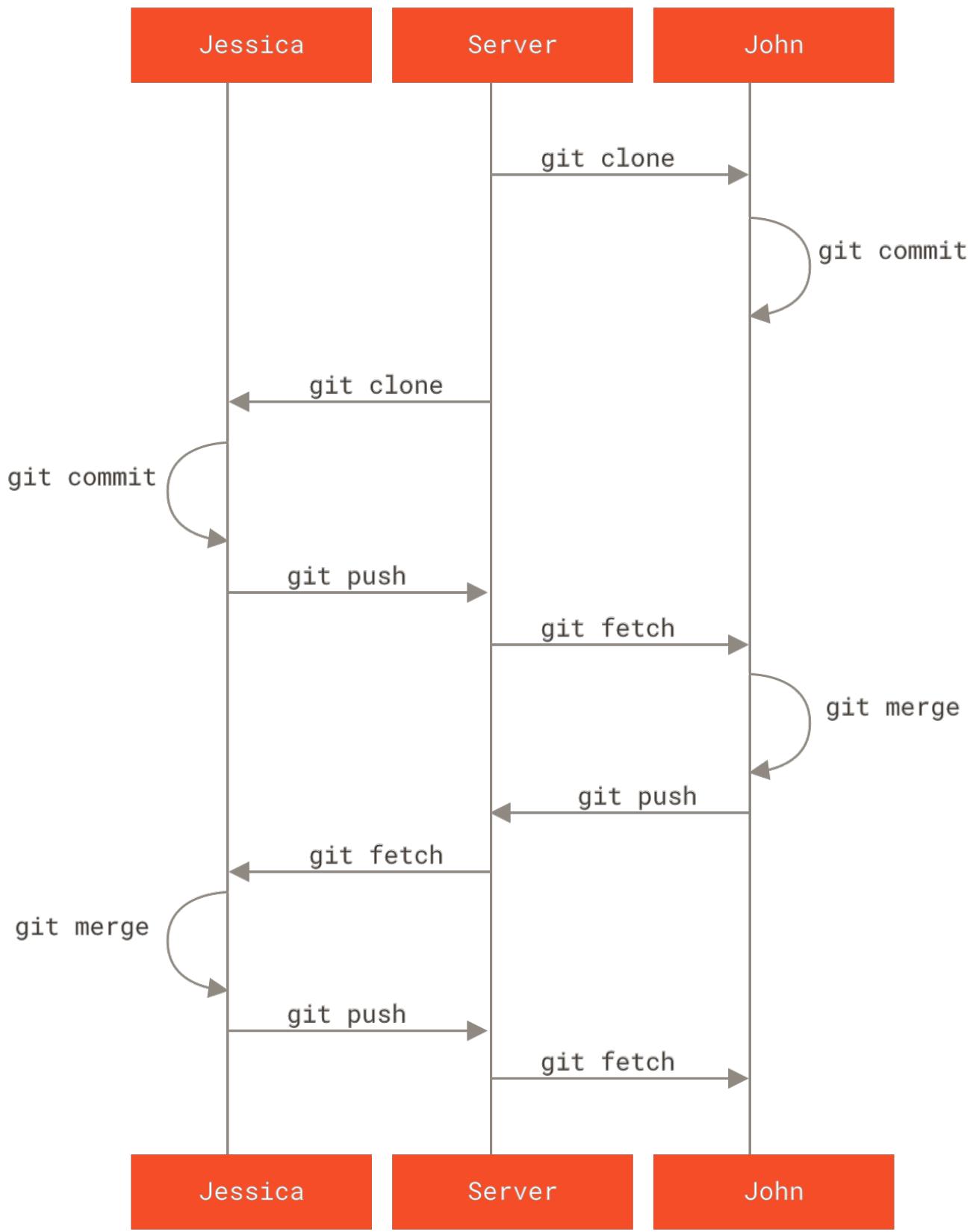
```
$ git push origin master
...
To jessica@githost:simplegit.git
 72bbc59..8059c15  master -> master
```

Сваки програмер је неколико пута комитовао и успешно спојио рад оног другог у свој.



Слика 63. Џесицина историја након гурања свих промена назад на сервер

То је један од најједноставнијих процеса рада. Радите неко време (у општем случају на тематској грани), па спојите тај рад у `master` грани када буде спреман да се интегрише. Када желите да поделите тај рад, преузмете и спојите у сопствену `master` грани из `origin/master` ако се променила, па на крају гурнете `master` грани на сервер. Уопштени низ акција изгледа некако овако:



Слика 64. Уопштени низ догађаја за једноставан процес рада са више програмера

Приватни тим са руководством

У овом наредном сценарију сазнаћете улоге дориносилаца у већој приватној групи. Научићете како да радите у окружењу у којем мале групе сарађују на могућностима, па неко други интегрише те тимске доприносе.

Рецимо да Џон и Џесика раде заједно на једној могућности (нека се то зове „featureA”), док Џесика и трећи програмер Џоси раде на другој (рецимо „featureB”). У овом случају компанија користи неку врсту процеса рада са руководством за интеграцију у којем рад појединачних група интегришу само одређени инжењери и само они могу да ажурирају `master` грану главног репозиторијума. У овом сценарију, сваки рад се обавља на тимски базираним гранама и касније га повлаче интегратори.

Хајде да пратимо Џесикин процес рада док ради на своје две могућности, односно док паралелно сарађује са два различита програмера у овом окружењу. Под претпоставком да већ има клониран репозиторијум, она одлучује да прво ради на `featureA`. Прави нову грану за могућност и ради нешто на њој:

```
# Џесикина машина
$ 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(-)
```

У овом тренутку је потребно да подели свој рад са Џоном, па гура комитове са своје `featureA` гране на сервер. Џесика нема дозволу да гурне на `master` грану—то могу само интегратори—па мора да гурне на неку другу грану како би сарађивала са Џоном:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

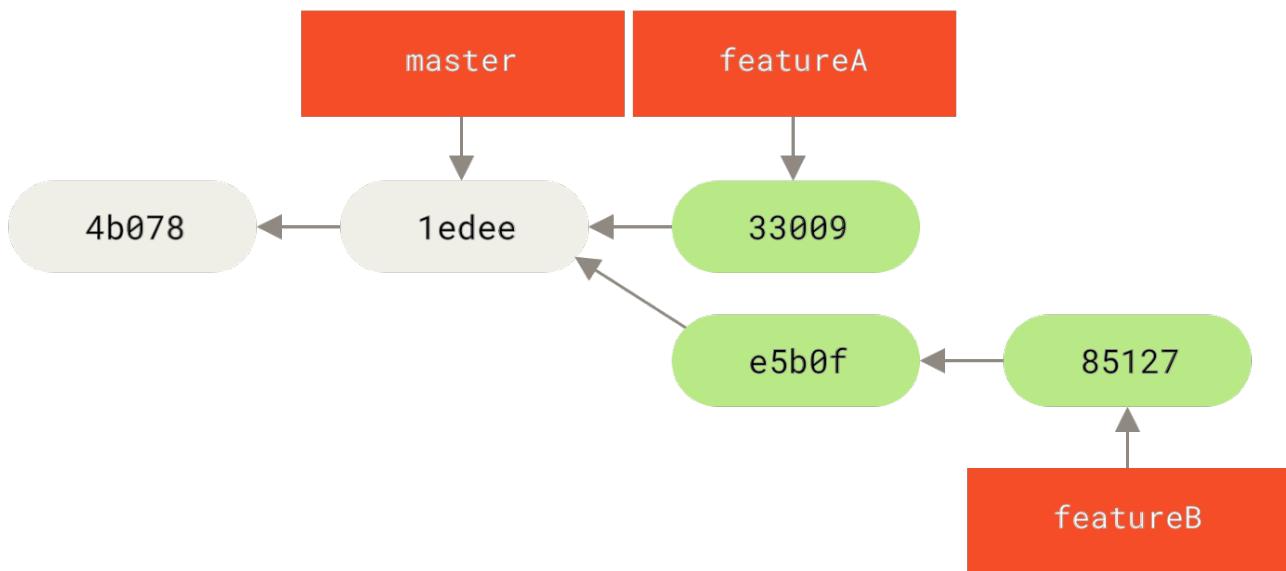
Џесика шаље мејл Џону у којем му каже да је гурнула неке ствари на грану која се зове `featureA` и да он сада може да погледа то. Док чека на повратну информацију од Џона, Џесика одлучује да почне посао на `featureB` са Џоси. За почетак, почиње нову грану за могућност и базира је на `master` грани са севера:

```
# Џесикина машина
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Sada Džesika pravi nekoliko komitova na grani `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(-)
```

Сада Џесикин репозиторијум изгледа овако:



Слика 65. Џесикина иницијална историја комитова

Спремна је да гурне свој рад, али добија мејл од Џоси да је грана са неким почетним радом на „featureB” већ гурнута на сервер под именом `featureBee`. Џесика најпре мора да споји те промене са сопственим пре него што буде у стању да гурне свој рад на сервер. Она преузима Џосине промене са `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Под претпоставком да је Џесика још увек на својој одјављеној грани `featureB`, сада може да споји Џосин рад са својим командом `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(-)
```

У овом тренутку Џесика жели да гурне сав овај спојени „featureB” рад назад на сервер, али не жели просто да гурне своју `featureB` грану. Пошто је Џоси већ започела узводну `featureBee` грану, Џесика жели да гурне у *tu* грану што чини са:

```
$ git push -u origin featureB:featureBee  
...  
To jessica@githost:simplegit.git  
fba9af8..cd685d1  featureB -> featureBee
```

Ово се зове *рефспек* (*refspeс*). За детаљнију дискусију о Гит рефспековима и разним стварима које можете да урадите помоћу њих, погледајте [Рефспек](#). Приметите и `-u` заставницу; ово је скраћено од `--set-upstream`, што конфигурише гране за касније лакше гурање и повлачење.

Одједном Џесика добија мејл од Џона који јој каже да је гурнуо неке промене на `featureA` грану на којој сарађују и моли је да их погледа. Она поново извршава `git fetch` да би преузела те промене:

```
$ git fetch origin  
...  
From jessica@githost:simplegit  
3300904..aad881d  featureA -> origin/featureA
```

Џесика може да погледа лог Џоновог новог рада поређењем садржаја управо преузете `featureA` гране са својом локалном копијом те исте гране:

```
$ 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 јој се допадне оно што види, спојиће Џонов нови рад у своју локалну `featureA` грану командом:

```
$ 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(-)
```

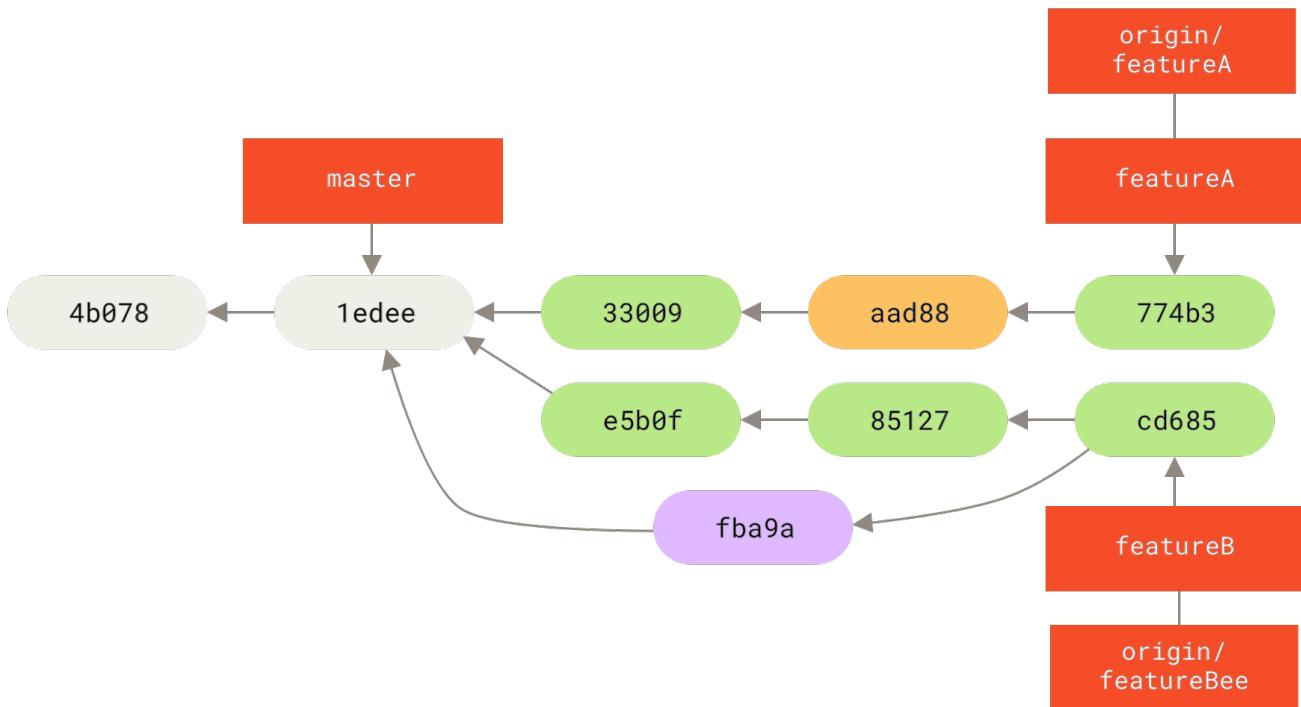
На крају, Џесика пожели да направи неколико малих измена овог коначно спојеног садржаја, може слободно да их уради, па да их комитује у своју локалну `featureA` грану и гурне крајњи резултат назад на сервер:

```

$ 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

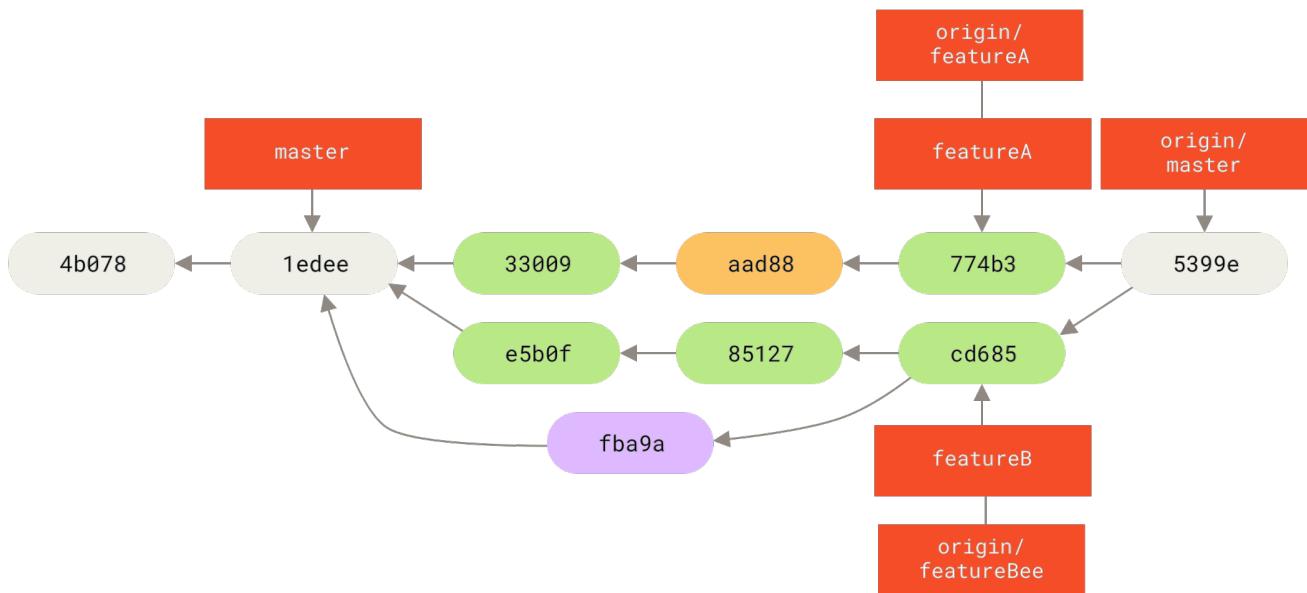
```

Џесикина историја комитова сада изгледа отприлике овако:



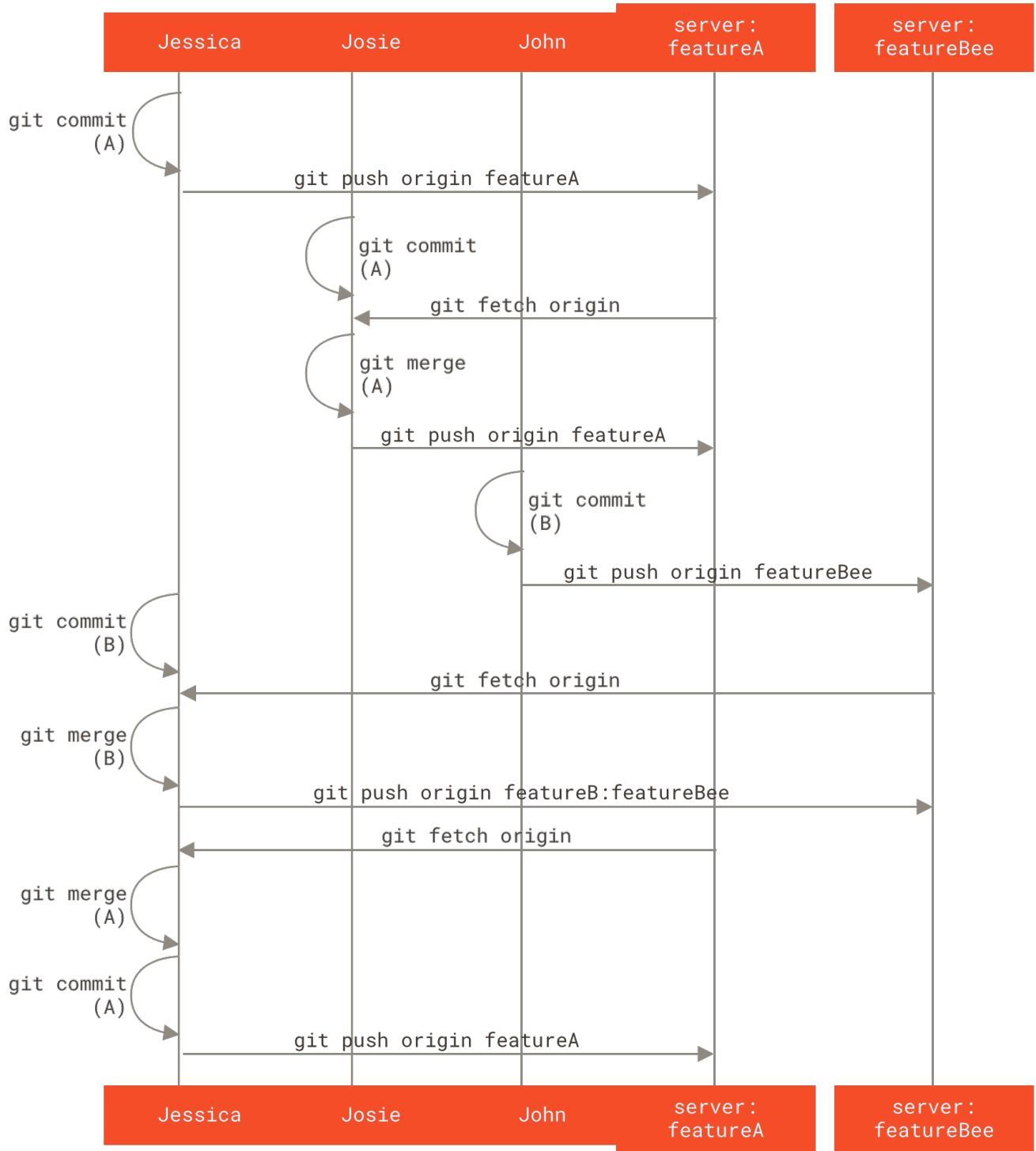
Слика 66. Џесикина историја након комитовања на грани могућности

У једном тренутку, Џесика, Џоси и Џон обавештавају интеграторе да су гране `featureA` и `featureBee` на серверу спремне за интеграцију у главну линију. Када интегратори споје ове гране у главну линију, преузимање ће довлачiti нови спојни комит, па ће историја изгледати овако некако:



Слика 67. Џесицина историја након спајања њених двеју тематских грана

Многе групе прелазе на Гит управо због ове могућности да више тимова ради паралелно, спајајући различите линије рада при крају процеса. Могућност програма Гит да мање подгрупе тима сарађују преко удаљених грана без потребе да у рад укључују или да успоравају читав тим је веома корисна. Процес рада који сте овде видели изгледа отприлике овако:



Слика 68. Основни процес рада оваквог приватног тима са руководством

Рачвани јавни пројекат

Давање доприноса јавним пројектима се ради на мало другачији начин. Пошто немате дозволу да директно мењате гране на пројекту, свој рад морате на неки други начин да проследите одржаваоцима пројекта. Први пример описује давање доприноса пројекту путем рачвања на Гит хостовима који подржавају једноставно рачвање. Ово подржавају многи хостинг сајтови (укључујући GitHub, BitBucket, repo.or.cz, и остале), многи одржаваоци пројекта очекују овакав начин давања доприноса. Наредни одељак описује пројекте који закрпе доприноса радије примају имејлом.

Најпре ћете пожелети да клонирате главни репозиторијум, направите тематску грану за закрпу или низ закрпа које планирате да поднесете, па та ту радите свој посао. Процес у суштини изгледа овако:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... рад ...
$ git commit
... рад ...
$ git commit
```



Можда ћете хтети да искористите `rebase -i` и тако згњечите свај рад у један једини комит, или да преуредите рад у комитовима тако да одржавалац једноставније може прегледати закрпу—погледајте [Поновно исписивање историје](#) за више информација о интерактивном ребазирању.

Када је ваш рад на грани обављен и спремни сте да га проследите одржаваоцима, пређите на оригиналну страницу пројекта и кликните на „Fork” дугме; тако правите своју личну рачву пројекта у коју можете да уписујете. Онда треба да додате овај нови URL репозиторијума као нови удаљени вашег пројекта, назовимо га у овом случају `myfork`:

```
$ git remote add myfork <url>
```

Онда треба да гурнете свој рад на тај репозиторијум. Најлакше је да тематску грану на којој радите гурнете на свој рачвани репозиторијум, а не да тај рад спајате са својом `master` граном, па да онда то гурнете. Разлог за то је што у случају да се ваш рад не прихвати или се из њега нешто селективно узме (*cherry-picked*), не морате да премотавате уназад своју `master` грану (о Гит операцији `cherry-pick` се детаљније говори у [Процеси рада са ребазирањем и одабиром \(cherry-picking\)](#)). Ако одржаваоци изврше `merge`, `rebase` или `cherry-pick` вашег рада, у сваком случају ћете вратити назад тако што ћете повући са њиховог репозиторијума.

Свој рад свакако можете гурнути са:

```
$ git push -u myfork featureA
```

Када ваш рад гурнете на свој рачвани репозиторијум, треба да обавестите одржаваоце оригиналног пројекта да имате рад који бисте желели да споје. Ово се често назива захтевом за повлачење (*pull request*) и обично га генеришете или преко веб сајта—GitHub има свој сопствени „Pull Request” механизам захтева за повлачење који ћемо обрадити у GitHub—или можете да покренете команду `git request-pull` и ручно пошаљете њен излаз на имејл одржаваоца пројекта.

Команда `git request-pull` узима основну грану у коју желите да повучете своју тематску грану и URL Гит репозиторијума са ког желите да се повуче, па штампа сажетак свих промена које тражите да се повуку. На пример, ако Џесика жели да Џону пошаље захтев за

повлачење, а урадила је два комита на тематској грани коју је управо гурнула, може да изврши следеће:

```
$ 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

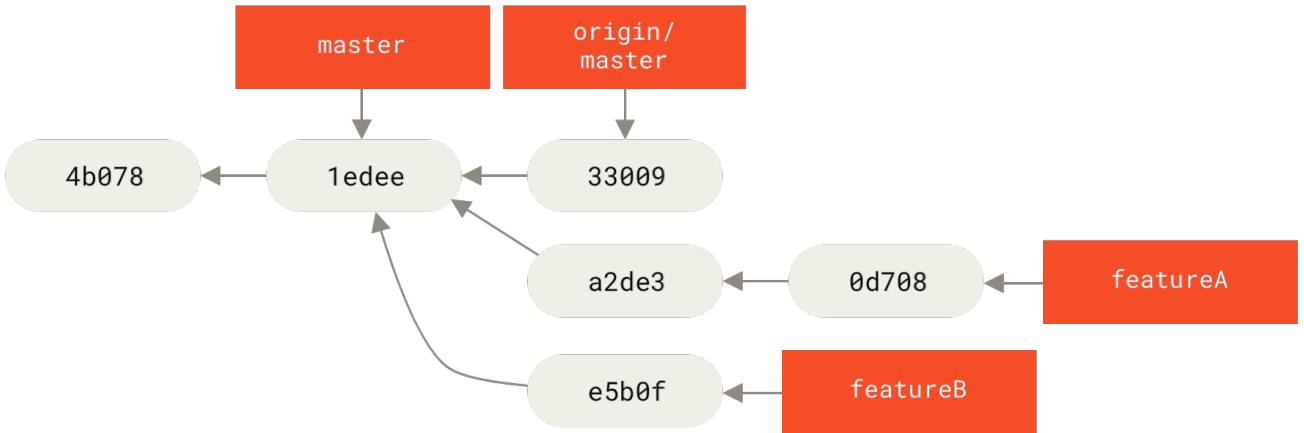
lib/simplegit.rb | 10 ++++++--
1 files changed, 9 insertions(+), 1 deletions(-)
```

Овај излаз може да се пошаље одржаваоцу—он му говори одакле је рад разгранат, резимира комитове и каже му одакле да повуче овај рад.

На пројекту који ви не одржавате, у општем случају је лакше да имате грану као што је `master` која увек прати `origin/master` и да свој посао радите на тематским гранама које лако можете да одбаците ако буду одбијене. Ако се у међувремену врх главног репозиторијума помери, па ваши комитови више не могу чисто да се споје, чињеница да је тематски рад изолован у тематске гране вам омогућава да лакше ребазирате свој рад. На пример, ако желите да пошаљете другу тему рада на пројекат, немојте настављати да радите на тематској грани коју сте управо гурнули узводно—почните од `master` гране главног репозиторијума:

```
$ git checkout -b featureB origin/master
... рад ...
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
... слање генерисаног захтева за повлачење имејлом одржаваоцу ...
$ git fetch origin
```

Сада је свака од тема садржана унутар силоса—слично као ред закрпа—које можете да ребазирате, измените и пишете преко њих а да се теме не мешају међусобно и да не зависе једна од друге, овако:

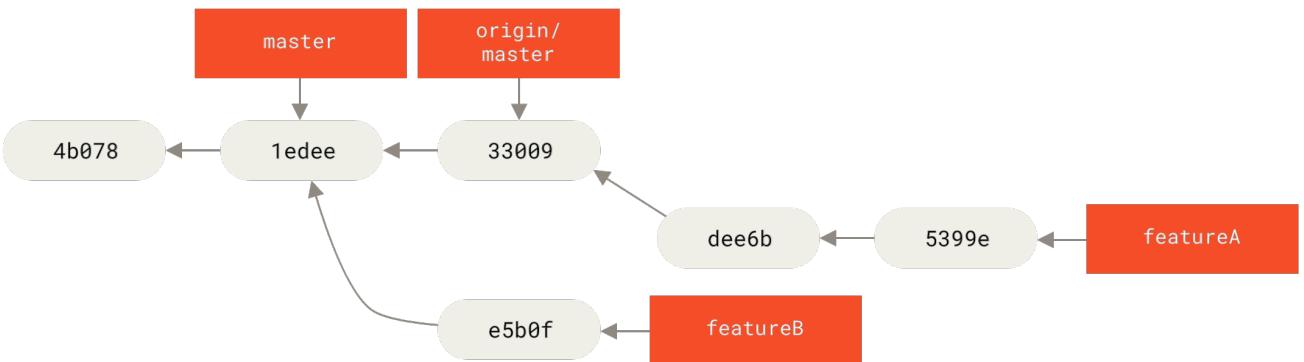


Слика 69. Иницијална историја комитова са радом `featureB`

Рецимо да је одржавалац пројекта повукао гомилу других закрпа и пробао вашу прву грану, али она се више не спаја без проблема. У том случају можете пробати да ребазирате ту грану на врх `origin/master` гране, да решите конфликте за одржаваоца, и онда поново пошаљете своје промене:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Ово поново исписује вашу историје тако да сада изгледа као [Историја комитова после рада `featureA`](#).



Слика 70. Историја комитова после рада `featureA`

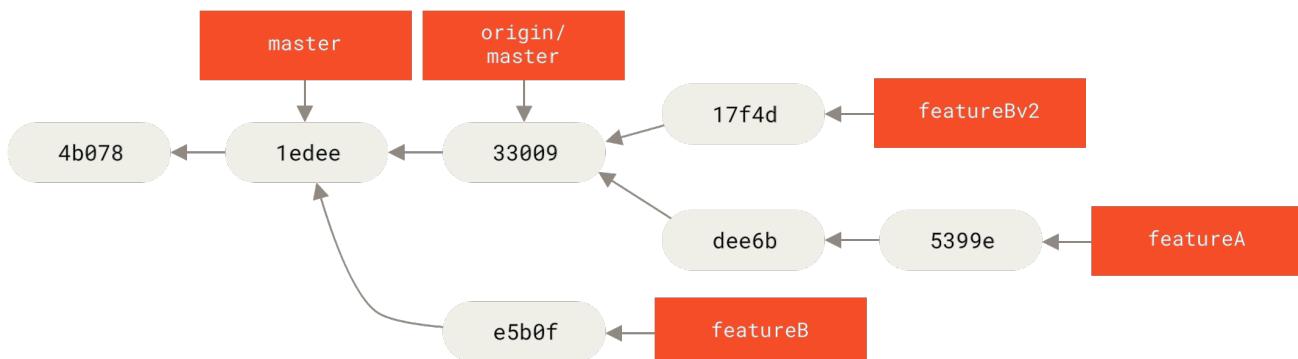
Пошто сте ребазирали грану, да бисте на серверу могли заменити грану `featureA` комитом који није њен потомак, у команди `push` морате да наведете `-f`. Други начин би могао бити да овај нови рад гурнете на другу грану на серверу (која се рецимо зове `featureAv2`).

Погледајмо још један могући сценарио: одржавалац је погледао ваш рад у другој грани и свиђа му се концепт, али би волео да направите промену у неком детаљу имплементације. Искористићете ову прилику и да преместите рад тако да буде базиран на тренутној `master` грани пројекта. Почекните нову грану базирану на тренутној `origin/master` грани, ту ћете да згњечите `featureB` промене, решићете евентуалне конфликте, направити измену имплементације и онда гурнути то као нову грану:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
... измена имплементације ...
$ git commit
$ git push myfork featureBv2
```

Опција `--squash` узима сав рад на спојеној грани и сабија га у један скуп промена, правећи тако стање репозиторијума као да се дододило право спајање, а да заправо не креира комит спајања. Ово значи да ће ваш будући комит имати само једног родитеља и дозвољава вам да уведете све промене са друге гране па да онда направите још промена пре него што забележите нови комит. Опција `--no-commit` такође може да буде корисна да одложите комит спајања у случају подразумеваног процеса спајања.

Сада можете да обавестите одржаваоце да сте направили захтеване измене и да могу да их пронађу у вашој `featureBv2` грани.



Слика 71. Историја комитова након рада на `featureBv2`

Јавни пројекат преко имејла

Многи пројекти имају утврђене процедуре за прихватање закрпи—мораћете да проверите одређена правила за сваки пројекат, јер ће се разликовати. Пошто постоји неколико старијих већих пројеката који прихватају закрпе преко мејлинг листе програмера, сада ћемо прећи тај пример.

Процес рада је сличан као у претходном случају—правите тематске гране за сваки низ закрпа на којем радите. Разлика је у томе како их шаљете пројекту. Уместо да рачвате пројекат и гурате на своју личну верзију за коју имате право уписа, генеришете имејл верзије сваког низа комитова и шаљете их на мејлинг листу програмера.

```
$ git checkout -b topicA
... рад ...
$ git commit
... рад ...
$ git commit
```

Сада имате два комита која желите да пошаљете на мејлинг листу. Извршавате `git format-patch` да генеришете `mbox` форматиране фајлове које можете да шаљете на листу — команда претвара сваки комит у имејл поруку у којој прва линија комит поруке постаје тема имејла, а остатак комит поруке и закрпа коју тај комит постаје тело имејла. Добра ствар у вези с овиме је то што примењивање закрпе из имејла који је генерисала команда `format-patch` исправно очувава све информације о комиту.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
```

Команда `format-patch` исписује имена фајлова закрпа које креира. Заставица `-M` налаже програму Гит да тражи фајлове којима је промењено име. Фајлови на kraju изгледају овако:

```
$ 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
```

Ове фајлове са закрпама можете и да уређујете тако да додате више информација за мејлинг листу које не желите да се појаве у комит поруци. Ако додате текст између `---` линије и почетка закрпе (линија `diff --git`), онда ће програмери моћи да га прочитају, али процес примењивања закрпе га игнорише.

Да бисте ово послали на листу, можете или да налепите фајл у свој имејл програм или да га

пошаљете програмом командне линије. Налепљивање текста често ствара проблеме са форматирањем, поготово са „паметнијим” клијентима који не задржавају карактере прелом линије и остале карактере празног простора. Срећом, програм Гит поседује алат који ће вам помоћи да IMAP протоколом шаљете добро форматиране закрпе, што би могло да вам буде лакше. Показаћемо како да пошаљете закрпу преко сервиса *Gmail*, што је имејл агент који најбоље познајемо; можете да прочитате детаљне инструкције за многе имејл програме на kraju већ поменутог фајла [Documentation/SubmittingPatches](#) у извornom коду програма Гит.

Прво, треба да подесите *imap* одељак у `~/.gitconfig` фајлу. Сваку вредност можете посебно да подесите низом `git config` команди, или можете ручно да их додате, али на крају би ваш `config` фајл требало да изгледа отприлике овако:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Ако ваш IMAP сервер не користи SSL, последње две линије вероватно нису неопходне и `host` вредност ће уместо `imaps://` бити `imap://`. Кад је то постављено, командом `git imap-send` можете да поставите низове закрпа у *Drafts* фолдер наведеног IMAP сервера:

```
$ 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
```

Сада би требало да можете да одете у *Drafts* фолдер, промените *To* поље у мејлинг листу на коју шаљете закрпу и евентуално ставите у *CC* одржаваоца или особу које је одговорна за тај одељак, па да пошаљете имејл.

Закрпе можете да шаљете и преко SMTP сервера. Као и раније, можете да подесите посебно сваку вредност низом `git config` команди, или да их ручно додате у `sendmail` одељак `~/.gitconfig` фајла:

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

Када ово урадите, командом `git send-email` можете да пошаљете своје закрпе:

```
$ 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
```

Програм Гит затим избаци гомилу лог информација које изгледају некако овако за сваку закрпу коју шаљете:

```
(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 помоћ око конфигурисања вашег система и имејла, још савета и трикова, као и за сенданбокс у који можете имејлом да пошаљете пробну закрпу, посетите git-send-email.io.

Резиме

Овим одељком смо покрили неколико процеса рада и говорили о разликама рада када сте део малог тима на пројектима затвореног кода у односу на давање доприноса великим јавним пројекту. Сада знате како да проверите да ли постоје грешке празних карактера пре него што комитујете и можете да пишете одличне комит поруке. Научили сте како да форматирате закрпе и како да их имејлом пошаљете на програмерску мејлинг листу. Такође смо покрили и рад са спајањима у контексту различитих процеса рада. Сада сте добро припремљени да дајете допринос било ком пројекту.

У наставку ћете научити како да направите другу страну новчића: да одржавате Гит пројекат. Научићете како да будете благонаклони диктатор или руководилац интеграцијом.

Одржавање пројекта

Осим што треба да знате како да ефикасно допринесете пројекту, вероватно ћете морати да

научите и како да одржавате пројекат. Ово може да се састоји од прихвататања и примењивања закрпа генерисаних са `format-patch` које су вам послате имејлом, или од интегрисања промена на удаљеним гранама за репозиторијум које сте додали као удаљене свом пројекту. Било да одржавате канонички репозиторијум или желите да помогнете тако што ћете верификовати или одобравати закрпе, треба да знате како да прихватите рад на начин који је најпрегледнији и најјаснији осталим сарадницима и да можете да га одржавате на дуге стазе.

Рад са тематским гранама

Када размишљате о интегрисању новог рада, у општем случају је добра идеја да ствари прво испробате на *тематској грани*—привременој грани коју сте направили само због тестирања тог новог кода. На овај начин је лако унети појединачно мале измене у закрпу и оставити је ако не ради све док не будете имали времена да јој се посветите. Ако изаберете једноставно име гране засновано на теми рада који желите да пробате, као што је `ruby_client` или нешто подједнако описано, лако можете да га запамтите ако морате да је напустите на неко време, па да јој се касније вратите. Одржавалац Гит тежи да прави и простор имена за гране—као што је `sc/ruby_client`, где је `sc` скраћеница за особу која даје допринос. Као што се сећате, можете да направите гране базиране на `master` грани на следећи начин:

```
$ git branch sc/ruby_client master
```

Или, ако желите да одмах и скочите на њу, можете да искористите опцију `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Сада сте спремни да додате допринети рад који сте примили у ову тематску грани и да одлучите да ли желите да га спојите у своју дуготрајну грану.

Примењивање закрпа из имејлова

Ако добијете закрпу преко мејла и треба да је интегришете у свој пројекат, закрпу треба да примените на тематску грану и да је процените. Постоје два начина да се примени закрпа коју сте добили путем имејла: са `git apply` или са `git am`.

Примењивање закрпе са `apply`

Ако сте добили закрпу од неког ко ју је генерисао командом `git diff` или неком варијацијом Јуниксове `diff` команде (што није препоручљиво, погледајте следећи одељак), можете да је примените `git apply` командом. Под претпоставком да сте закрпу сачували у `/tmp/patch-ruby-client.patch`, овако можете да је примените:

```
$ git apply /tmp/patch-ruby-client.patch
```

Ово мења фајлове у радном директоријуму. Скоро је идентично као и извршавање команде

`patch -p1` за примењивање закрпе, мада је више параноична и прихвата мање непотпуних подударања него `patch`. Обрађује и додавањем, брисањем и преименовањем фајлова у случају да је то описано у `git diff` формату, што `patch` неће да уради. Коначно, `git apply` је модел са принципом „примени све или одбаци све” где ће се или применити све или ништа, док `patch` може парцијално да примењује закрпе, остављајући радни директоријум у чудном стању. `git apply` је у целини посматрано много конзервативнији него `patch`. Неће креирати комит уместо вас—када га покренете, уведене промене морате ручно да стејџујете и комитујете.

`git apply` можете да употребите и да видите да ли ће се закрпа применити без проблема пре него што заправо пробате да је примените—само извршите `git apply --check` са закрпом:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
errgor: patch failed: ticgit.gemspec:1
errgor: ticgit.gemspec: patch does not apply
```

Ако нема излаза, то значи да би закрпа требало да се примени без проблема. У случају неуспеха ова команда такође враћа излазну вредност различиту од нуле, тако да је можете користити у скриптама ако желите.

Примењивање закрпе са `am`

Ако сарадник користи програм Гит и ако је био доволно искусан да за генерирање своје закрпе употреби команду `format-patch`, онда ће ваш посао бити много лакши јер закрпа садржи информације о аутору и комит поруку. Ако можете, саветујте своје сараднике да за генерирање закрпа које вам шаљу уместо `diff` користе `format-patch`. `git apply` би требало да користите само за старе закрпе и такве ствари.

Да бисте применили закрпу која је генерирана са `format-patch`, употребите `git am`. Технички, `git am` је створено да чита `mbox` фајлове; то су једноставни фајлови чистог текстуалног формата који служе за чување једне или више имејл порука у једном текстуалном фајлу. Изгледају отприлике овако:

```
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
```

Ово је почетак излаза команде `format-patch` коју сте управо видели у претходном одељку; а такође је и исправан `mbox` имејл формат. Ако вам је неко исправно послао имејл са закрпом користећи `git send-email`, а ви га преузмете у `mbox` формат, онда можете да усмерите `git am` на тај `mbox` фајл, и почеће да примењује све закрпе које види. Ако користите имејл клијент који може да чува неколико имејлова у `mbox` формату, можете да сачувате читаву серију закрпи у фајл, па да онда употребите `git am` над тим фајлом, а команда ће кренути да примењује све закрпе које пронађе у фајлу.

Међутим, ако је неко на систем за тикете или нешто слично окачио закрпу коју је генерирао са `format-patch`, можете локално да сачувате фајл, па да онда тај фајл који је сачуван на диску проследите команди `git am` да бисте га применили:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Као што видите, закрпа је примењена без проблема и аутоматски је креиран нови комит. Информације о аутору су узете из `From` и `Date` заглавља имејла, а порука комита је узета из `Subject` и тела (пре закрпе) имејла. На пример, ако је ова закрпа примењена из `tbox` примера изнад, генерисани комит би изгледао некако овако:

```
$ 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
```

Информација `Commit` указује на особу која је применила закрпу и на време када је то учињено. Информација `Author` је особа која је првобитно направила закрпу и када.

Али може се догодити и да се закрпа не примени без проблема. Можда је ваша главна грана отишла предалеко од гране за коју је закрпа направљена, или закрпа зависи од неке друге закрпе коју још увек нисте применили. У том случају, `git am` процес неће успети и питаће вас шта желите да урадите:

```
$ 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".
```

Ова команда поставља маркере конфликта у све фајлове у којима постоје проблеми, слично као код конфликта при спајању или ребазирању. Проблем се такође решава на исти начин — уредите фајл тако да решите конфликт; затим стејџујете нови фајл, па покренете `git am --resolved` да се настави на следећу закрпу:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

Ако желите да програм Гит покуша мало интелигентније да реши конфликт, можете да му проследите опцију `-3`, што програму Гит налаже да проба троструко спајање. Ова опција подразумевано није укључена јер не функционише ако вам комит на коме закрпа каже да је базирана није у репозиторијуму. Ако имате тај комит — ако је закрпа базирана на јавном комиту — онда је у оваштем случају опција `-3` много паметнија када се примењује закрпа са конфликтом:

```
$ 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.
```

У овом случају, без опције `-3` би се закрпа сматрала за конфликт. Пошто је употребљена опција `-3` закрпа се применила без проблема.

Ако примењујете неколико закрпа из `mbox` фајла, команду `am` можете да покренете и у интерактивном режиму, која стаје на свакој закрпи коју пронађе и пита вас желите ли да је примените:

```
$ 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
```

Ово је лепо ако имате сачуван већи број закрпа, јер најпре можете да видите закрпу ако се не сећате шта је у питању, или можете да не примените закрпу ако сте то већ урадили.

Када су све закрпе за вашу тему примењене и комитоване на вашу грану, можете изабрати да ли ћете и како да их интегришете у дуготрајнију грану.

Одјављивање удаљених грана

Ако је допринос послao Гит корисник који је подесио сопствени репозиторијум, гурнуо неки број промена на њега, па вам послao URL до репозиторијума и име удаљене гране у којој се налазе промене, можете да их додате као удаљени репозиторијум и локално урадите спајања.

Рецимо, ако је вам је Џесика послала мејл у коме каже да има одличну нову ствар у `ruby-client` грани свог репозиторијума, можете да је тестирате додавањем удаљеног репозиторијума и одјављивањем те гране локално:

```
$ git remote add jessica git://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

Ако вам касније пошаље још један имејл и са још једном граном са корисном могућности, могли бисте директно да одрадите `fetch` и `checkout` јер сте већ подесили удаљени репозиторијум.

Ово је најкорисније ако редовно радите са неком особом. Ако неко с времена на време даје допринос само једном закрпом, онда је прихватање преко мејла брже него захтевање да сви имају свој сервер и да сви стално додају и бришу удаљене репозиторијуме само да би добили неколико закрпа. Такође, мало је вероватно да желите да имате на стотине удаљених репозиторијума, сваки за неког ко допринесе само једну или две закрпе. Ипак, скрипте и хостинг сервиси могу ово да учине једноставнијим — зависи највише од тога како развијате програм и како то раде ваши сарадници.

Друга предност овог приступа је то што добијате и историју комитова. Мада можете да имате озбиљне проблеме са спајањем, знате где је у историји базиран њихов рад; право троструко спајање је подразумевано, тако да не морате да убацујете `-3` и да се надате да је закрпа генерисана од јавног комита коме имате приступ.

Ако не радите редовно са особом, али ипак желите да повучете од њих на овај начин, `git pull` команди можете да наведете URL удаљеног репозиторијума. Ово један пут ради повлачење и не чува URL као удаљену референцу:

```
$ git pull https://github.com/onetimeguy/project  
From https://github.com/onetimeguy/project  
 * branch HEAD      -> FETCH_HEAD  
Merge made by the 'recursive' strategy.
```

Како утврдити шта је уведено

Сада имате тематску грану која садржи допринесен рад. У овом тренутку можете да одлучите шта ћете да радите са њим. Овај одељак се поново осврће на неколико команди како бисте видели како да их искористите да бисте видели тачно шта ћете урадити ако спојите ово у главну грану.

Често је корисно да добијете преглед свих комитова који су у овој грани али нису у вашој `master` грани. Додавањем `--not` опције испред имена гране можете да искључите комитове са `master` гране. Ово ради исту ствар као и `master..contrib` формат који смо користили раније. На пример, ако вам сарадник пошаље две закрпе и направите грану која се зове `contrib`, па примените те закрпе тамо, можете да извршите следеће:

```
$ 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 желите да видите које промене уводи сваки од комитова, сетите се да команди `git log` можете проследити опцију `-p` и она ће надовезати разлику која је уведена при сваком комиту.

Да бисте видели комплетну разлику онога што би се дододило ако бисте спојили ову тематску грану са другом граном, можда ћете морати да употребите чудан трик да бисте добили исправне резултат. Вероватно вам пада на памет да извршите следећу команду:

```
$ git diff master
```

Ова команда вам приказује разлику, али може да вас завара. Ако се ваша `master` грана кретала унапред од када сте од ње направили тематску грану, онда ћете добити наизглед чудне резултате. Ово се дешава јер програм Гит директно пореди снимке последњег комита са тематске гране на којој се тренутно налазите и последњег комита `master` гране. На пример, ако сте додали линију у фајл на `master` грани, директно поређење снимака ће изгледати као да ће тематска грана да обрише ту линију.

Ако је `master` непосредни предак тематске гране, ово не представља проблем; али ако су две историје разишле, разлика ће изгледати као да додајете све нове ствари у тематску грану и бришете све што је јединствено у `master` грани.

Оно што уствари хоћете да видите су промене које су додате на тематској грани — рад који ћете увести ако спојите ову грану са `master` граном. То ћете постићи тако што ћете програму Гит наложити да упореди последњи комит на тематској грани са првим заједничким претком који тематска грана има са `master` граном.

Технички, то можете да урадите тако што ћете експлицитно одредити заједничког претка и онда покренути `diff` над њиме:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

или сажетије:

```
$ git diff $(git merge-base contrib master)
```

Међутим, ниједно од ова два није посебно згодно за рад, па програм Гит нуди још једну скраћеницу којом радите исту ствар: синтаксу са три тачке. У контексту `git diff` команде, можете ставити три тачке после друге гране да бисте урадили `diff` између последњег комита гране на које се налазите и његовог заједничког претка са другом граном:

```
$ git diff master...contrib
```

Ова команда вам приказује само рад који уводи текућа тематска грана почевши од заједничког претка на `master` грани. То је веома корисна синтакса коју треба упамтити.

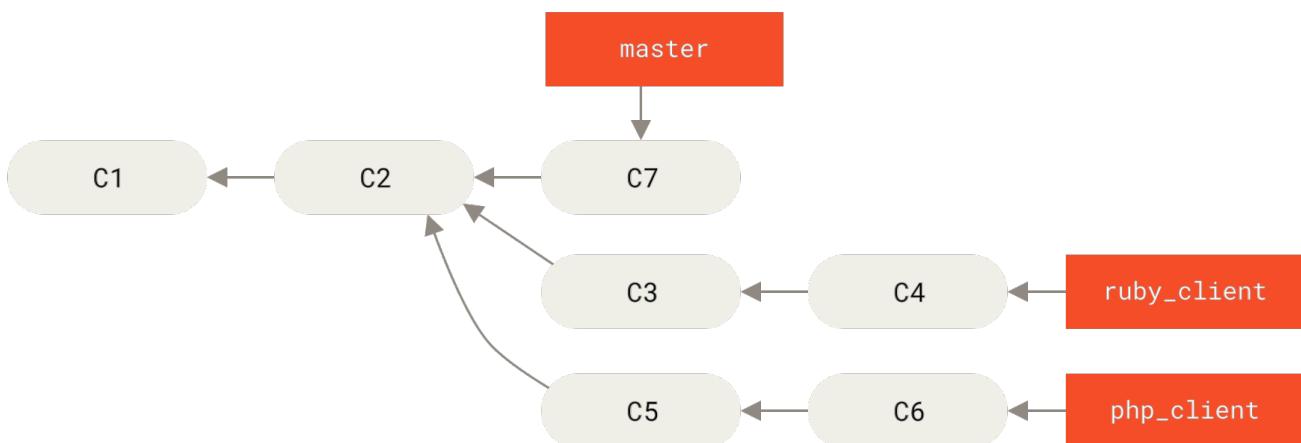
Интегрисање допринесеног рада

Када је сав рад на тематској грани спреман за интегрисање у главнију грану, поставља се питање како то извести. Сем тога, који свеобухватни процес рада желите да користите та одржавање свог пројекта? Имате пуно избора, па ћемо представити њих неколико.

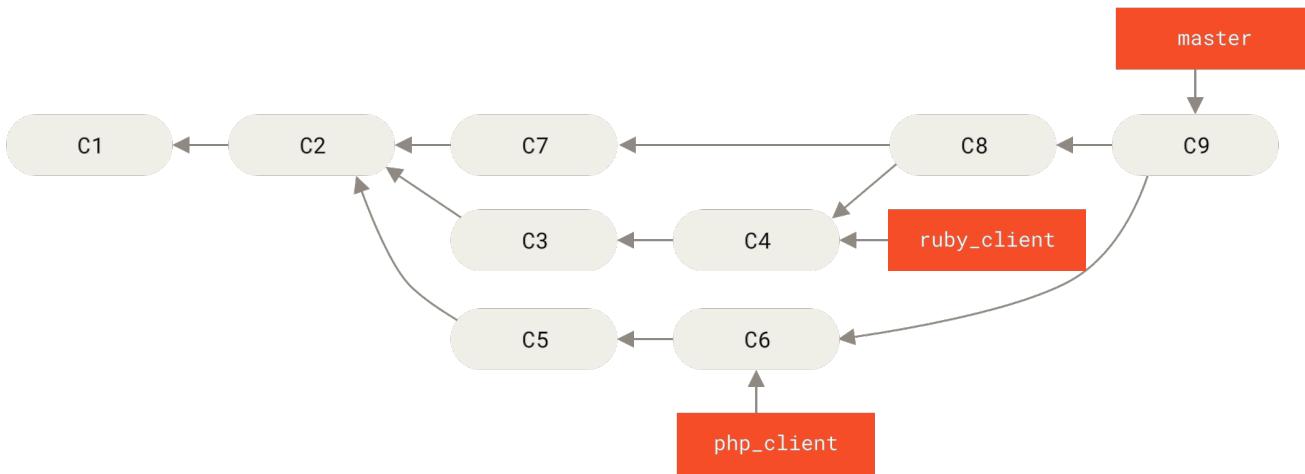
Процеси рада са спајањем

Један од основних процеса рада је да просто спојите сав рад директно у своју `master` грану. У овом сценарију, имате `master` грани која у суштини садржи стабилан код. Када имате рад у тематској грани који вам се чини завршеним, или рад који је неко други допринео а ви сте га проверили, спајате га у своју `master` грану, па бришете ту управо спојену тематску грану и понављате процес.

На пример, ако имамо репозиторијум са радом у две гране које се зову `ruby_client` и `php_client` који изгледа као [Историја са неколико тематских грана](#) и прво спојимо `ruby_client`, па онда `php_client`, историја коју ћете на крају имати изгледа као [После спајања тематске гране](#).



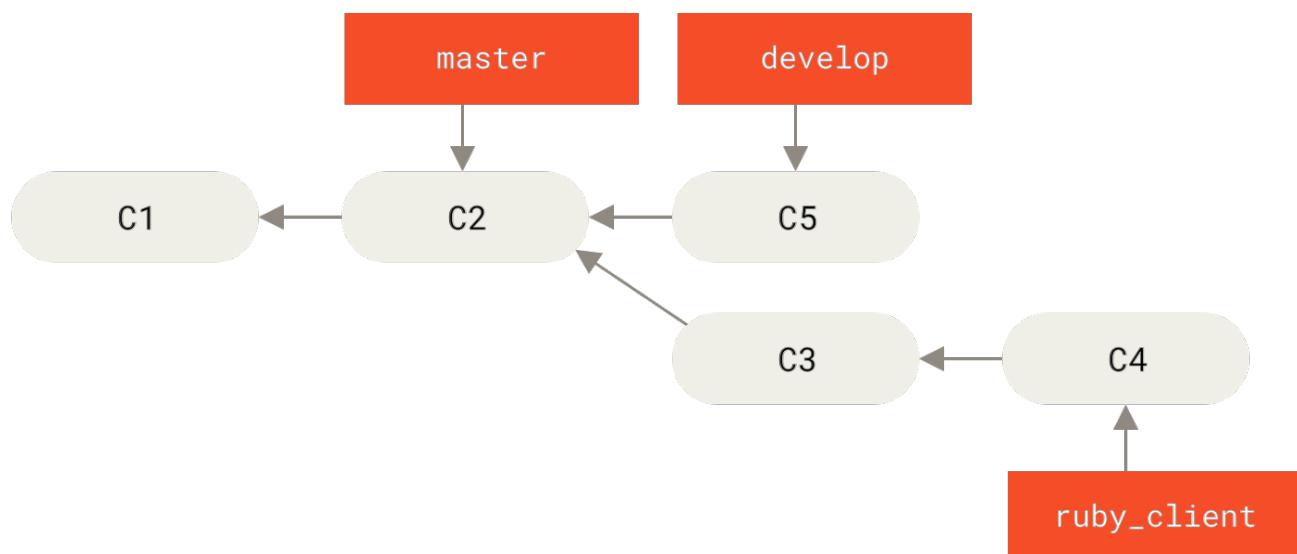
Слика 72. Историја са неколико тематских грана



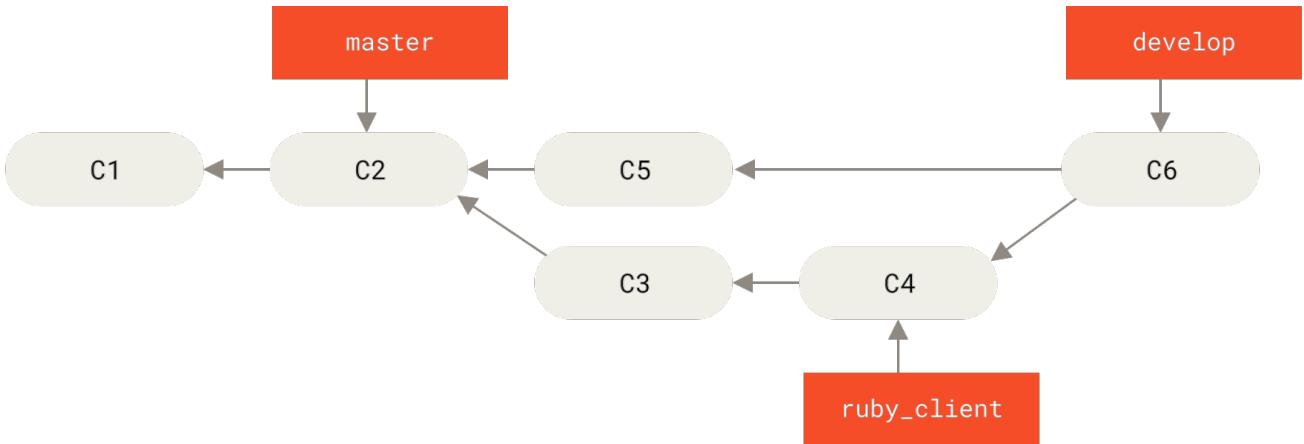
Слика 73. После спајања тематске гране

Ово је вероватно најједноставнији процес рада, али може да буде проблематичан ако радите на већим или стабилнијим пројектима где желите да буде јако обазриви око тога шта уводите.

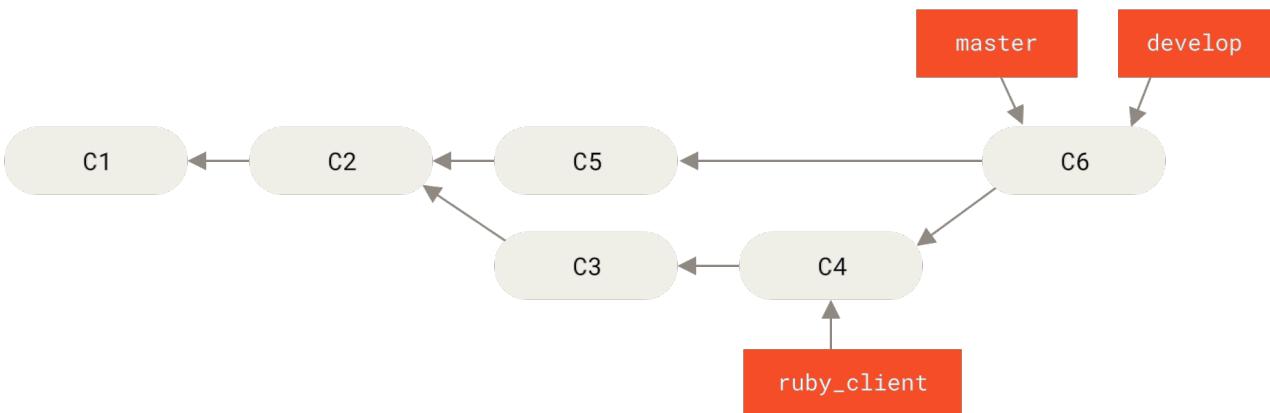
Ако имате важнији пројекат, можда ћете желети да користите двофазни циклус спајања. У овом сценарију имате две дуготрајне гране, `master` и `develop`, у којима одређујете да се `master` ажурира само када се издвоји врло стабилно издање и сав нови код се интегрише у `develop` грану. Обе ове гране редовно гурате у јавни репозиторијум. Сваки пут када имате нову тематску грану коју треба спојити ([Пре спајања тематске гране](#)), спајате је у `develop` ([Након спајања тематске гране](#)); затим, када означите издање, `master` грану премотате унапред на место на којем се налази сада стабилна `develop` грана ([Након издања пројекта](#)).



Слика 74. Пре спајања тематске гране



Слика 75. Након спајања тематске гране

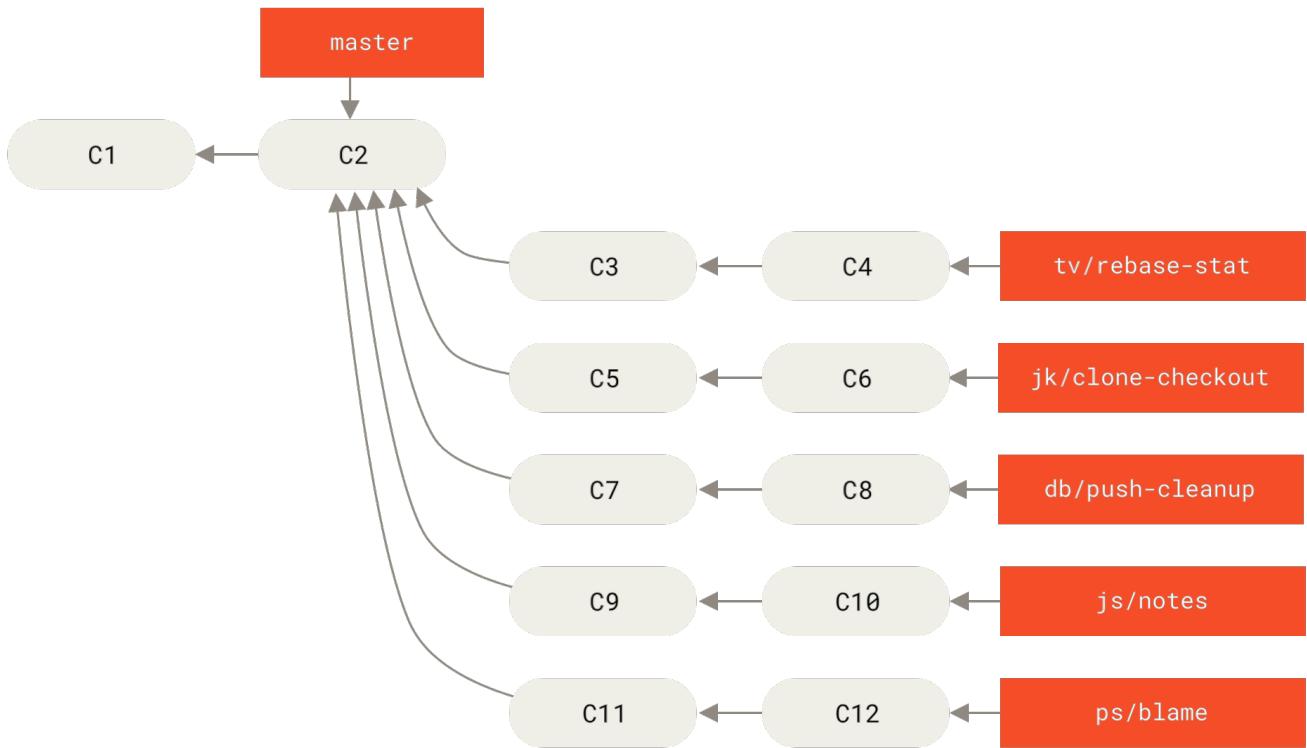


Слика 76. Након издања пројекта

Овако, када људи клонирају репозиторијум вашег пројекта, могу или да одјаве `master` да изграде последњу стабилну верзију и једноставно увек одржавају актуелну верзију, или могу да одјаве `develop`, која садржи најновији код. Овај концепт можете и да проширите тако да имате `integrate` грану у коју се сви рад спаја у једно. Онда, када код на тој грани постане стабилан и прође тестирање, спајате га у `develop` грани; и када се то на неко време покаже као стабилно, `master` грани премотате унапред.

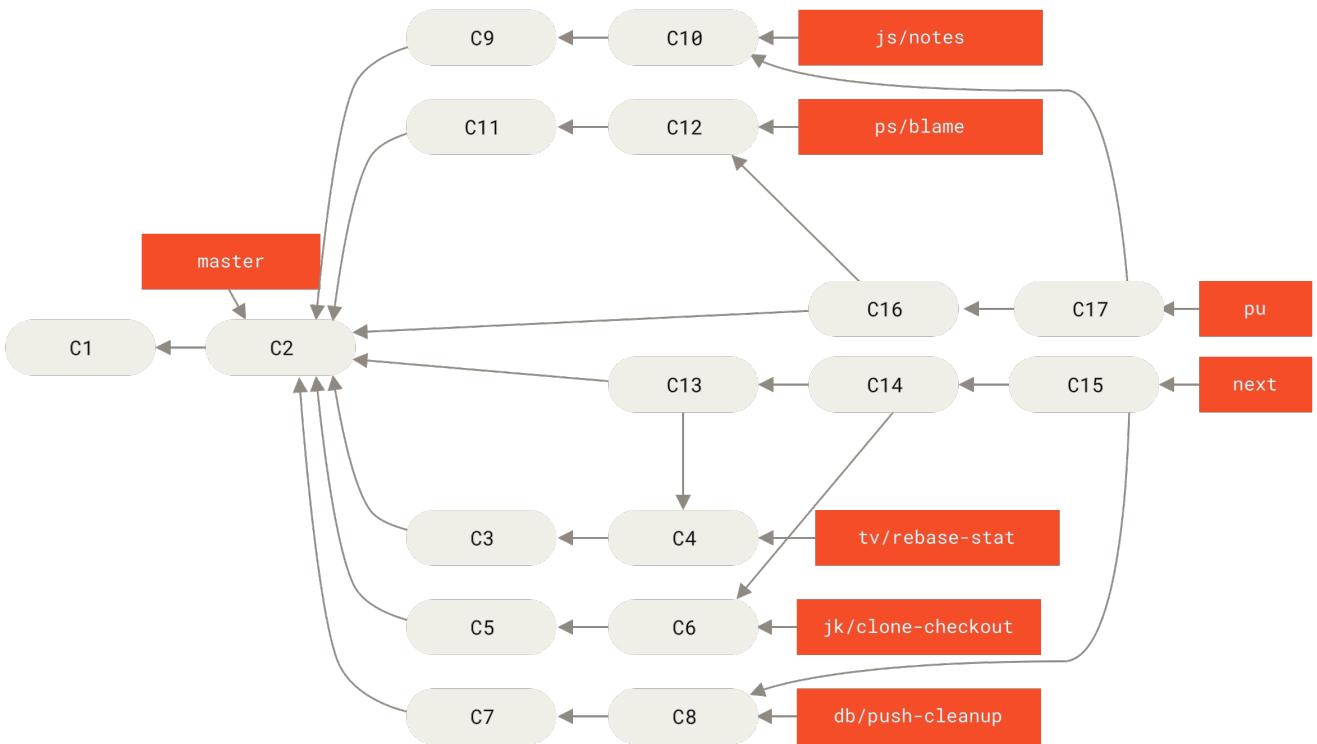
Процеси рада са спајањем великих грана

Гит пројекат има четири дуготрајне гране: `master`, `next` и `seen` (раније се звала `ru` — предложена ажурирања) за нови рад и `maint` за бекпортове одржавања. Када сарадници унесу нови рад, он се сакупља у тематске гране репозиторијума одржаваоца на начин сличан ономе што смо већ описали (погледајте [Управљање комплексним низом тематских грана на које се даје допринос](#)). У овом тренутку, теме се процењују да би се одредило да ли је безбедно да се употребе или још треба да се ради на њима. Ако су безбедне, спајају се у `next`, и та грана се туре како би сви могли да пробају интегрисане теме заједно.



Слика 77. Управљање комплексним низом тематских грана на које се даје допринос

Ако на темама још треба да се ради, спајање се обавља у `seen`. Када се одреди да су потпуно стабилне, теме се поново спајају у `master`. Затим се се поново изграђују гране `next` и `seen` од `master` гране. Ово значи да се `master` скоро увек креће унапред, `next` се с времена на време ребазира, а `seen` се још чешће ребазира.



Слика 78. Спајање тематских грана доприноса у дуготрајне гране за интеграцију

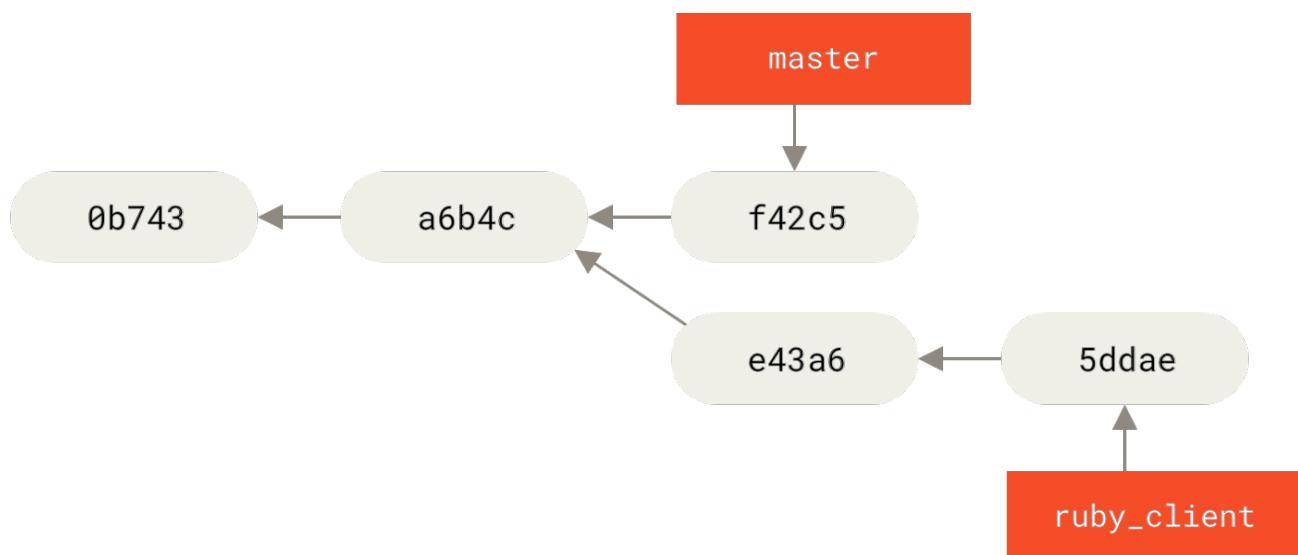
Када се тематска грана коначно споји у `master`, она се брише из репозиторијума. Гит пројекат такође има и `maint` грану која се рачва из последњег издања да би се обезбедиле

бекпортироване закрпе за случај да је потребно ново издање које само решава ситније проблеме а не укључује нови садржај (*maintenance release*). Зато, када клонирате Гит репозиторијум, имате четири гране које можете одјавити да сагледате пројекат у разним фазама развоја, зависно од тога колико стабилну верзију желите да имате, или како желите да дате допринос пројекту; а одржавалац има структурисан процес рада који им помаже око нових доприноса. Процес рада Гит пројекта је специјализован. Да бисте ово јасно разумели, погледајте [водич за Гит одржаваоца](#).

Процеси рада са ребазирањем и одабиром (*cherry-picking*)

Други одржаваоци више воле допринесени рад ребазирају или одаберу (*cherry-pick*— од допринесеног селективно изаберу оно са највише користи) допринесен рад са врха своје `master` гране уместо да га споје у њу, како би одржали углавном линеарну историју. Када имате рад у тематским гранама и одлучили сте да желите да га интегришете, померате се на ту грану и покрећете команду за ребазирање да бисте поново изградили промене на врх текуће `master` гране (или `develop` гране, и тако даље). Ако то прође како треба, можете да премотате унапред своју `master` грану и завршићете са линеарном историјом пројекта.

Други начин да преместите уведени рад са једне гране на другу је да га одaberete. Одабирање (*cherry-picking*) је у програму Гит као ребазирање за један комит. Узима закрпу која је уведена комитом и покушава да је поново примени на грану на којој се тренутно налазите. Ово је корисно ако имате већи број комитова на тематској грани, а желите да интегришете само један од њих, или имате само један комит на тематској грани и више вам одговара да селективно изаберете само један од њих (да га одaberete) уместо да обавите ребазирање. На пример, претпоставимо да имате пројекат који изгледа овако:

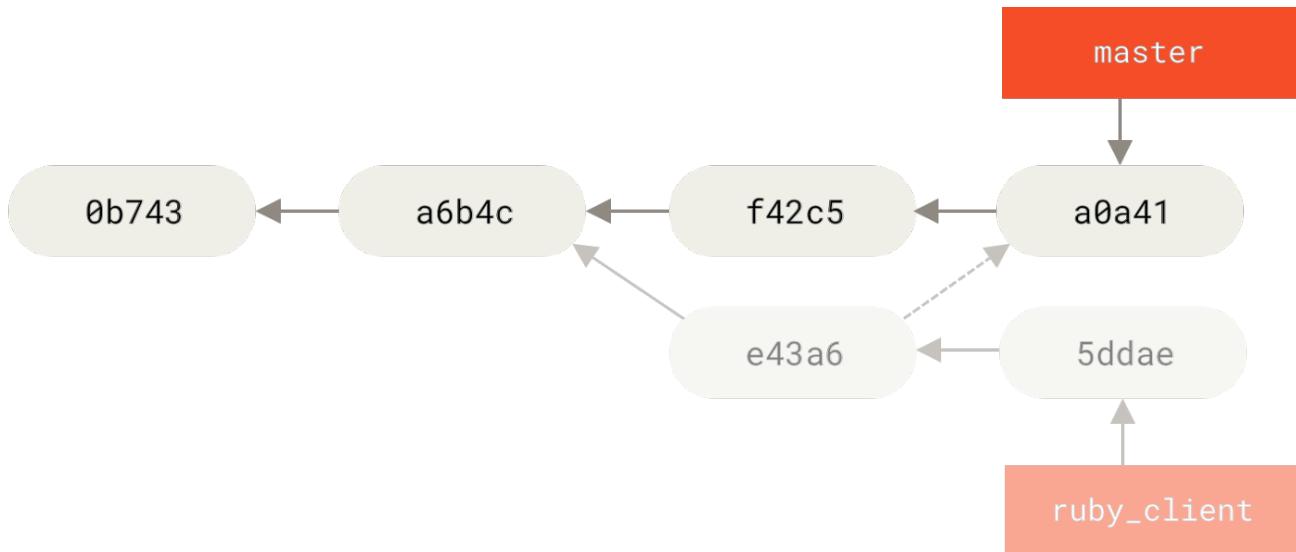


Слика 79. Пример историје пре одабирања

Ako želite da povučete комит `e43a6` у `master` грану, можете да извршите:

```
$ 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(-)
```

Ово повлачи исту промену која је уведена у `e43a6`, али добијате нову SHA-1 вредност за комит, јер је датум примењивања другачији. Сада ваша историја изгледа овако:



Слика 80. Историја након одабира комита на тематској грани

Сада можете да обришете тематску грани и одбаците комитове које нисте желели да повучете.

Rerere

Ако радите пуно спајања и ребазирања, или одржавате дуготрајну тематску грани, програм Гит вам пружа могућност под именом „rerere” која може да помогне.

Rerere је скраћеница од „*reuse recorded resolution*” (поново искористи забележено решење)—то је начин да се скрати ручно решавање конфликта. Када је *rerere* укључено, програм Гит ће чувати скуп пре- и пост-слика успешних спајања, па ако примети да постоји конфликт који потпуно личи на неки који сте већ разрешили, онда ће искористити то решење од прошлог пута и неће вас замарати тиме.

Ова могућност долази у два дела: конфигурационо подешавање и команда. Конфигурационо подешавање је `git rerere.enabled` и доволно је корисно да га ставите у глобална подешавања:

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

Одсада, кад год урадите спајање које разрешава конфликте, решење ће бити забележено у кешу за случај да поново затреба у будућности.

Ако буде било потребе, командом `git rerere` можете да имате интеракцију са *rerere* кешом.

Када се позове сама, програм Гит проверава своју базу података решења и покушава да пронађе подударање са било којим од тренутних конфликтата при спајању и реши их (мада се ово ради аутоматски ако је `gpgv.enabled` подешено на `true`). Постоје и подкоманде којима можете да видите шта ће бити забележено, да обришете одређено решење из кеша и да обришете цео кеш. Rerere ћемо детаљније описати у [Rerere](#).

Означавање издања

Када одлучите да направите пресек и објавите ново издање, обично је добра идеја да доделите ознаку тако да бисте могли поново да креирате то издање у било ком каснијем тренутку. Нову ознаку можете направити онако како смо објаснили у [Основе програма Гит](#). Ако одлучите да потпишете ознаку као одржавалац, означавање би могло да изгледа овако:

```
$ 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 потпишете своје ознаке, могли бисте имати проблем са дистрибуцијом јавног PGP кључа који се користи за потписивање ваших ознака. Одржавалац Гит пројекта решава овај проблем тако што укључује свој јавни кључ као блоб у репозиторијуму и онда додаје ознаку која директно показује на тај садржај. Да бисте урадили ово, можете одредити који кључ желите тако што ћете извршити `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]
```

Онда тај кључ можете директно да увезете у базу података програма Гит тако што ћете га извести и пајповати то кроз `git hash-object`, што пише нови блоб са тим садржајем у Гит и враћа вам SHA-1 блоба:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Сада када имате садржај свог кључа у програму Гит, можете да направите ознаку која показује директно на њега тако што ћете навести нову SHA-1 вредност коју вам је вратила команда `hash-object`:

```
$ git tag -a maintainer-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Ако извршите `git push --tags`, ознака `maintainer-gpg-pub` ће бити подељена са свима. Ако неко

жели да потврди ознаку, може директно да увезе ваш PGP кључ тако што ће повући блоб директно из базе података и увести га у GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

Тај кључ може да се искористи за потврђивање свих ваших потписаних ознака. Такође, ако уз поруку ознаке прикључите и инструкције, извршавање `git show <ознака>` ће вам омогућити да крајњем кориснику издате одређенија упутства у вези потврђивања ознака.

Генерисање броја изградње

Пошто програм Гит нема монотону растуће бројеве као 'v123' или нешто тако који ће ићи уз сваки комит, у случају да уз сваки комит желите и име читљиво за људе, можете да извршите `git describe` над тим комитом. Као одговор, програм Гит генерише стринг који се састоји од имена најближе ознаке раније од тог комита а, након којег следи број комита од те ознаке и на крају делимичну SHA-1 вредност комита који се описује (испред којег се наводи слово „g” са значењем Гит):

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

На овај начин можете да изvezете снимак или изградњу и да му дате име које људима има смисла. Заправо, ако изградите програм Гит из извornог кода који сте клонирали из Гит репозиторијума, `git --version` вам даје нешто што изгледа овако. Ако описујете комит који сте директно обележили, једноставно вам даје име ознаке.

Команда `git describe` подразумевано захтева прибележене ознаке (ознаке креиране са заставицом `-a` или `-s`); ако желите и да користите предности једноставних ознака (које нису прибележене), команди додајте и опцију `--tags`. Овај стринг можете да користите и као одредиште за `git checkout` или `git show` команду, мада се она ослања на скраћену SHA-1 вредност са краја, тако да можда неће важити довека. На пример, Линукс кернел је недавно скочио са 8 на 10 карактера да би се обезбедила јединственост међу SHA-1 објекта, тако да су стари излази команде `git describe` постали неважећи.

Припрема за издање

Сада желите да објавите изградњу. Једна од ствари коју ћете желети да урадите јесте да креирате архиву последњег снимка свог кода за оне јадне душе које не користе програм Гит. Команда за то је `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 неко отвори тај тарбол, добиће последњи снимак вашег пројекта у директоријуму `project`. Можете и да направите `zip` архиву на скоро исти начин, само што ћете команди `git`

`archive` проследити опцију `--format=zip`:

```
$ git archive master --prefix='project/' --format=zip > 'git describe master'.zip
```

Сада имате лепу тарбол и zip архиву издања свог пројекта коју можете поставити на свој веб сајт или послати имејлом другим људима.

Кратки лог

Време је да пошаљете имејл људима на вашој листи који желе да знају шта се дешава са вашим пројектом. Леп начин да брзо добијете неку врсту белешки промена (тзв. *changelog*) онога што је додато у пројекат од последњег издања или мејла је да употребите `git shortlog` команду. Она ће сумирати све комитове у опсегу који наведете; на пример, следећа команда ће вам дати кратак преглед свих комитова од последњег издања, ако се последње издање зове v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray 'puts'
    Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

Добијате чист резиме свих комитова од верзије v1.0.1, груписане по аутору, који можете послати као имејл на своју листу.

Резиме

Сада би требало да вам доприношење пројекту у програму Гит и одржавање сопственог пројекта или интеграција доприноса других корисника дође природно. Честитамо, постали сте ефикасан Гит програмер! У следећем поглављу ћете научити како да користите један од најпопуларнијих гит хостинг сервиса, GitHub.

GitHub

Сервис *GitHub* је највећи хост за Гит репозиторијуме и представља централну тачку сарадње за милионе програмера и пројекта. Велики проценат свих Гит репозиторијума се хостује на сервису *GitHub* и многи пројекти отвореног кода га користе за Гит хостинг, праћење тикета, преглед кода, и друге ствари. Зато, иако није непосредни део Гит пројекта отвореног кода, постоји велика вероватноћа да ћете у неком тренутку када будете почели професионално да користите програм Гит бити упућени на интеракцију са сервисом *GitHub*.

Ово поглавље објашњава како да ефикасно користите сервис *GitHub*. Покрићемо прављење и одржавање налога, прављење и коришћење Гит репозиторијумâ, уобичајене процесе рада по којима се дају доприноси пројектима и по којима се они прихватају, програмски интерфејс сервиса *GitHub* и много ситних савета за олакшање рада у општем случају.

Ако вас не занима коришћење сервиса *GitHub* за хостовање сопствених пројекта или сарадња са другим пројектима који су хостовани тамо, можете слободно да прескочите на [Гит алати](#).

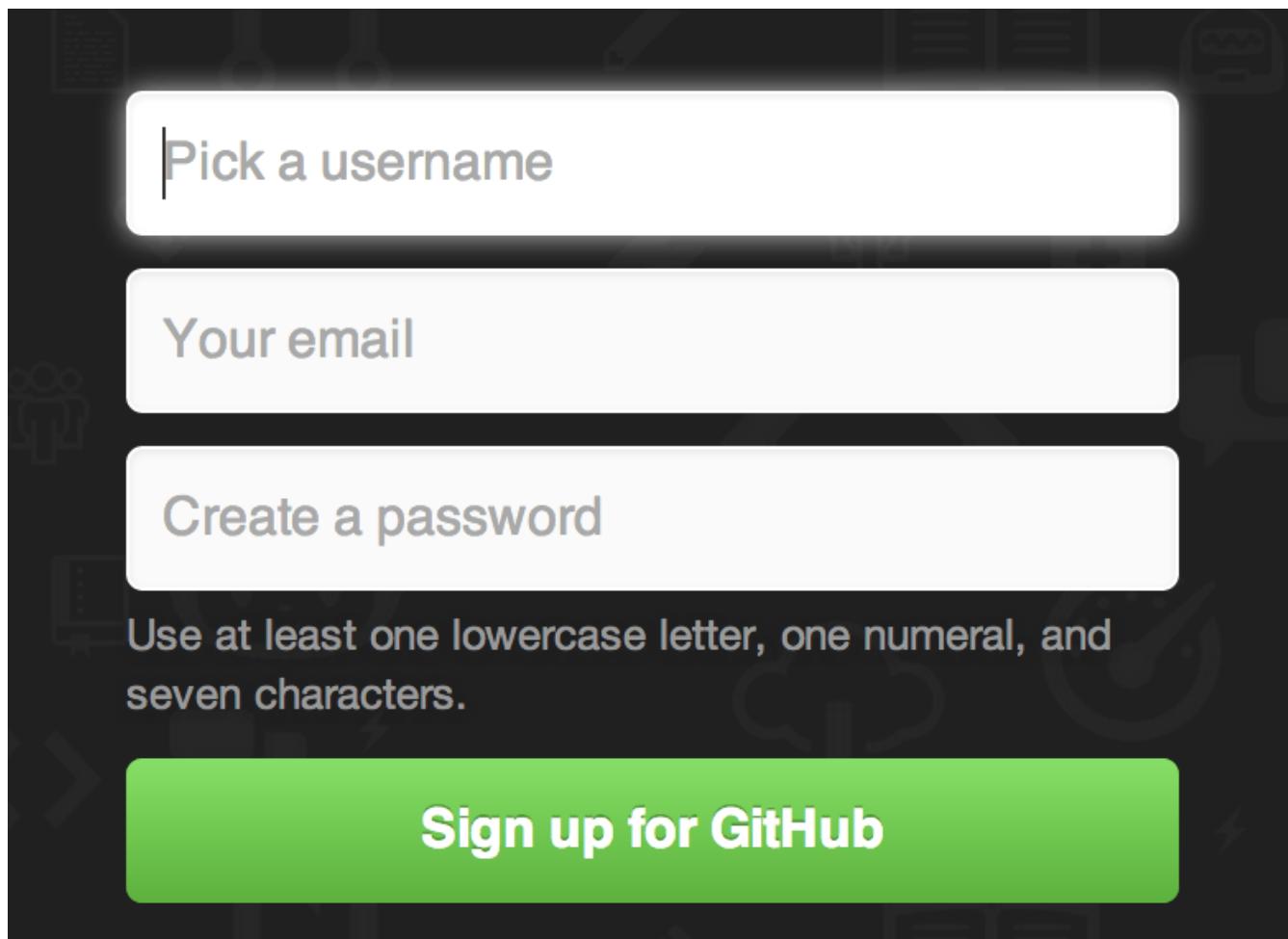
Промена интерфејса



Битно је напоменути да ће се снимци екрана елемената корисничког интерфејса дефинитивно мењати током времена, као што је то случај са многим активним веб сајтовима. Надамо се да ће генерална идеја о томе како се одређене ствари раде опстати, али ако желите нешто актуелније верзије снимака екрана, верзије ове књиге на мрежи ће можда имати новије снимке екрана.

Отварање налога и подешавања

Прва ствар коју треба да урадите је да отворите бесплатан кориснички налог. Једноставно посетите <https://github.com>, изаберите корисничко име које још увек није заузето, оставите своју имејл адресу и жељену лозинку, па кликните на велико зелено дугме „*Sign up for GitHub*”.



Слика 81. Форма за регистрацију на GitHub

Следећа ствар коју ћете видети је ценовник за напредније планове, али засад без проблема можете игнорисати ово. GitHub ће вам послати мејл којим ћете потврдити адресу коју сте унели. Учините ово одмах, јер је веома важно (као што ћемо видети касније).



GitHub бесплатним налозима нуди скоро сву своју функционалност, осим неких напредних могућности. GitHub комерцијални планови укључују напредне алате и могућности као и подигнуте границе за бесплатне сервисе, али њих нећемо обрадити у овој књизи. Ако желите више информација о доступним плановима и њихово поређење, посетите <https://github.com/pricing>.

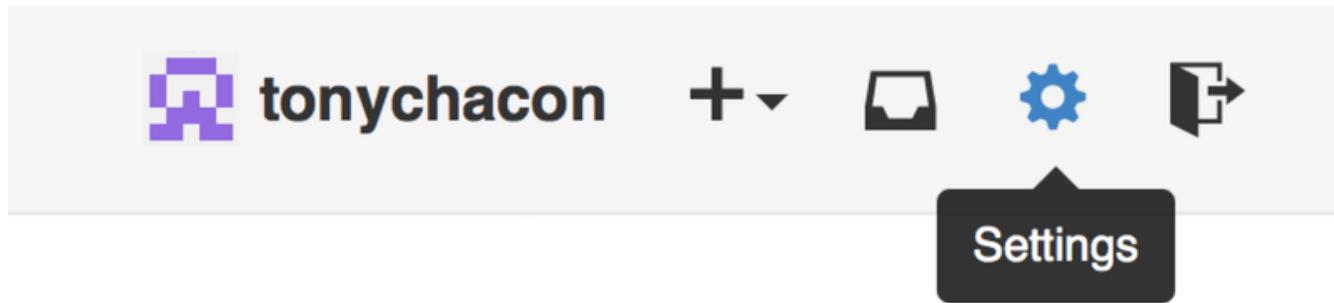
Ако кликнете на лого Октомачке у горњем левом углу екрана, одлазите на своју главну страну. Сада сте спремни да почнете са коришћењем сервиса GitHub.

SSH приступ

Сада сте у могућности да се у потпуности повежете са Гит репозиторијумима користећи <https://> протокол, потврђујете свој идентитет корисничким именом и лозинком које сте управо поставили. Међутим, ако желите једноставно да клонирате јавне пројекте, чак нема потребе ни да се региструјете — налог који смо креирали игра улогу када будемо рачвали пројекте и касније гурали рад на своје рачве.

Ако желите да користите SSH удаљене репозиторијуме, мораћете да конфигуришете јавни

кључ. Ако га још увек немате, погледајте [Генерисање јавног SSH кључа](#). Отворите подешавања свог налога користећи линк у горњем десном углу прозора:



Слика 82. Линк „Account settings”

Онда изаберите одељак „SSH keys” са леве стране.

A screenshot of the "Add an SSH Key" form on GitHub. The left sidebar shows the "SSH keys" section is selected. The main area has a heading "Add an SSH Key" and two input fields: "Title" and "Key". Below them is a green "Add key" button. Above the input fields, there's a note: "Need help? Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)".

Слика 83. SSH keys линк

Одавде, кликните на дугме „Add an SSH key”, дајте кључу име, па налепите садржај свог `~/.ssh/id_rsa.pub` (или како сте га већ назвали) фајла са јавним кључем у текст поље и кликните „Add key”.

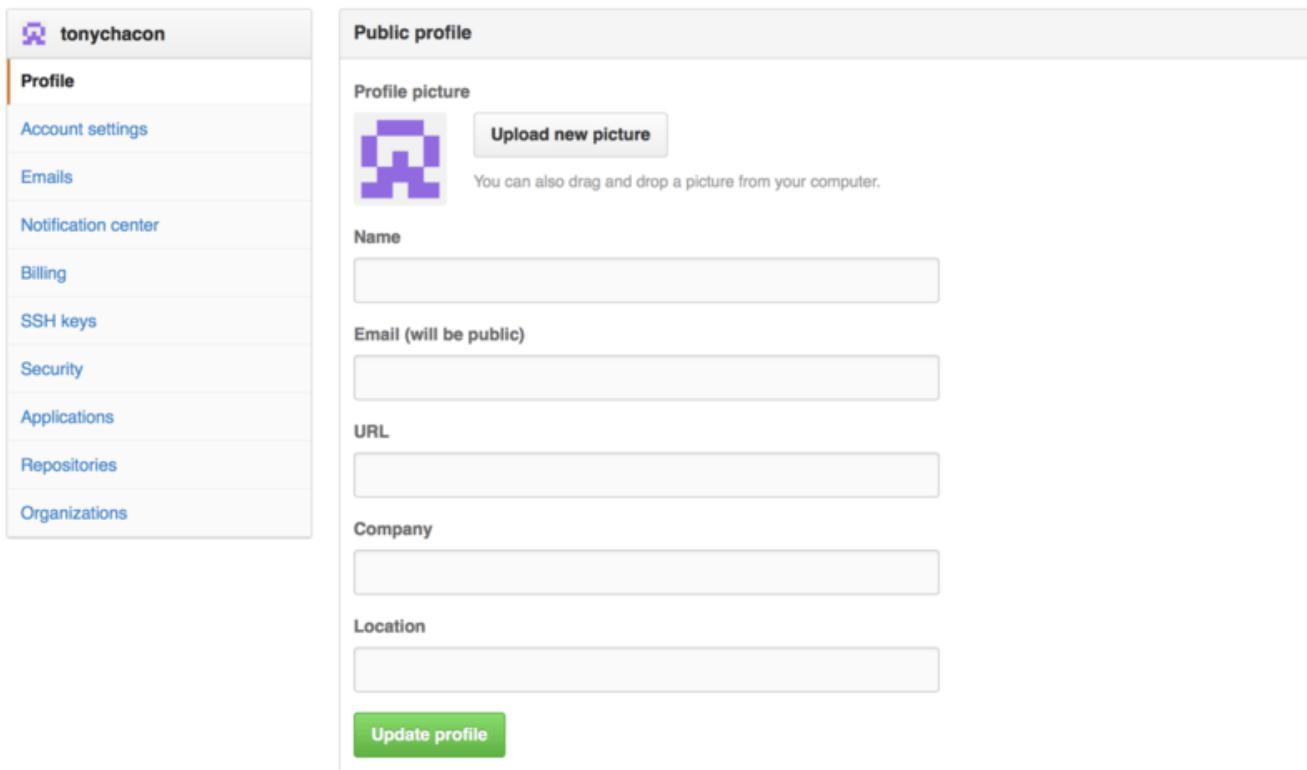


Обавезно дајте име свом SSH кључу тако да га лако запамтите. Сваком од својих кључева можете да дате име (нпр. „Мој лаптоп” или „Радни налог”) тако да ако касније морате да га опозовете, лако можете видети који кључ тражите.

Лични аватар

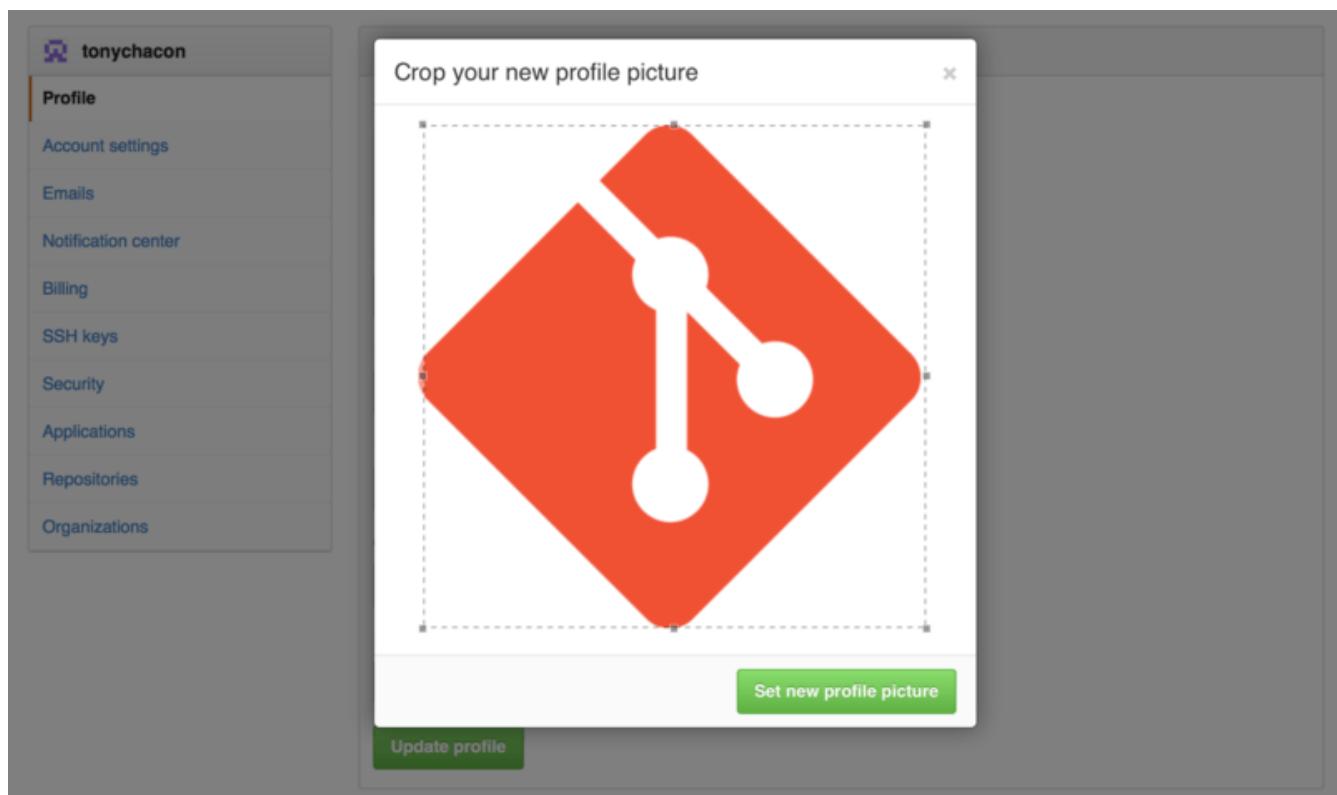
Затим, ако желите, генерисани аватар можете да промените slikom коју ви изаберете.

Најпре идите у картицу „Profile” (изнад картице „SSH Keys”) и кликните на „Upload new picture”.



Слика 84. Линк „Profile”

Изабраћемо копију Гит лога који имамо на хард диску; након тога добијамо прилику да га опсечемо.



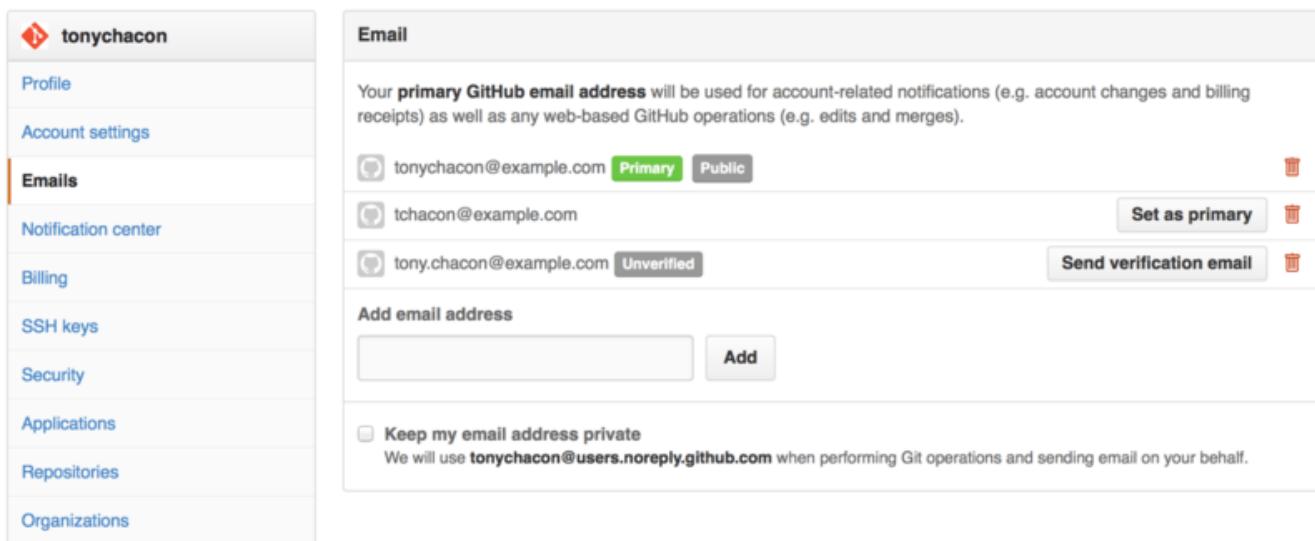
Слика 85. Oncećaње аватара

Одсада, кадгод вршите интеракцију негде на сајту, људи ће поред вашег корисничког имена видети ваш аватар.

Ако се деси да сте аватар окачили на популарни сервис Граватар (често се користи за Вордпрес налоге), тај аватар ће аутоматски користити и нема потребе да радите овај корак.

Ваше имејл адресе

Начин на који GitHub мапира ваше Гит комитове на ваш кориснички налог је помоћу имејл адресе. Ако у својим комитовима користите више имејл адреса и желите да их GitHub повеже како треба, треба да додате све имејл адресе које сте користили у *Emails* одељак admin одељка.



Слика 86. Додавање имејл адреса

У [Додавање имејл адреса](#) можемо видети неке од различитих могућих стања. Адреса на врху је потврђена и постављена као примарна адреса, што значи да ћете на њу добијати сва обавештења и рачуне. Друга адреса је потврђена и може да се подеси као примарна ако желите да је промените. Последња адреса није потврђена, што значи да њу не можете поставити као своју примарну адресу. Ако GitHub види било коју од ових у комит порукама у било ком репозиторијуму на сајту, одсада ће бити повезани са вашим налогом.

Двофакторска аутентификација

Најзад, за додатну сигурност, дефинитивно треба да подесите *Two-factor Authentication* тј. „2FA”. Двофакторска аутентификација је механизам проверу идентитета који у последње време постаје све популарнији за смањење ризика компромитовања вашег налога у случају да вам се некако докопа лозинке. Ако укључите ову опцију, GitHub ће вам тражити да потврдите идентитет употребом две различите методе; тако да ако једна од њих буде компромитована, нападач неће моћи да приступи вашем налогу.

Подешавање двофакторске аутентификације се налази у картици „*Security*” вашег „*Account settings*”.

The screenshot shows the GitHub user interface for security settings. On the left, a sidebar lists account management options: Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security (which is selected), Applications, Repositories, and Organizations. The main content area has two sections: 'Two-factor authentication' and 'Sessions'. In 'Two-factor authentication', the status is 'Off' with a red 'X', and a button 'Set up two-factor authentication' is visible. A note explains that 2FA provides another layer of security. In the 'Sessions' section, it lists devices logged into the account: 'Paris 85.168.227.34' (current session, Safari on OS X 10.9.4, signed in on September 30, 2014) and 'Safari on OS X 10.9.4' (signed in on September 30, 2014). There is also a link to 'GitHub Help'.

Слика 87. 2FA и Security картици

Ако кликнете на дугме „*Set up two-factor authentication*”, одлазите на конфигурациону страницу где можете изабрати да користите апликацију на телефону за генерисања секундарног кода („временски базирана једнократна лозинка”), или да вам GitHub шаље кôд преко у SMS поруци сваки пут када треба да се пријавите на систем.

Када изаберете жељену методу и испратите инструкције за подешавање 2FA, ваш налог ће онда бити мало сигурнији и мораћете да поред лозинке унесете и кôд сваки пут када се пријављујете на GitHub.

Како се даје допринос пројекту

Сада када вам је налог подешен, хајде да прођемо кроз неке детаље који би вам могли бити од користи када желите да дате допринос постојећем пројекту.

Рачвање пројеката

Ако желите да дате допринос постојећем пројекту за који немате дозволу гурања промена, можете да га „рачвате” (*fork*). GitHub ће направити копију пројекта која је потпуно ваша; пројекат сада живи у вашем простору имена и можете да гурате на њега.



Историјски, термин „fork” је имао помало негативну конотацију јер је описивао ситуацију када неко поведе пројекат отвореног кода у другом правцу, неретко стварајући супарнички пројекат и подели сараднике. У сервису GitHub, „fork” је једноставно исти пројекат у вашем простору имена, што вам дозвољава да јавно правите промене пројекта као начин да се допринос даје још отвореније.

На овај начин пројекти не морају да брину о додавању корисника као сарадника да би им дали дозволу за гурање промена. Људи могу да рачвају пројекат, гурају на рачву и дају свој

допринос својих промена назад оригиналном репозиторијуму тако што ће креирати нешто што се зове захтев за повлачење (*Pull Request*), што ћемо описати касније. Ово отвара тему за дискусију са прегледом кода, па онда власник или особа која даје допринос могу да разговарају о промени све док власник не буде задовољан; након тога власник може да споји промене.

Да бисте рачвали пројекат, посетите страницу пројекта и кликните на дугме „*Fork*” у горњем десном углу странице.



Слика 88. „*Fork*” дугме

Након неколико секунди, бићете одведени на страницу свог новог пројекта, са својом сопственом копијом кода по којој можете да пишете.

GitHub процес рада

GitHub је дизајниран према одређеном процесу рада за сарадњу који је фокусиран на захтеве за повлачење. Овај начин рада функционише било да сарађујете са добро увезаним тимом на једном дељеном репозиторијуму, или са глобално дистрибуираном компанијом или мрежом странаца који дају допринос пројекту преко бројних рачви. Базира се на [Тематске гране](#) процесу рада који смо описали у [Гранање у програму Гит](#).

Ево како ради у општем случају:

1. Рачвате пројекат.
2. Направите тематску грану из `master`.
3. Направите неке комитове да побољшате пројекат.
4. Гурнете ову грану на свој GitHub пројекат.
5. Отворите захтев за повлачење на GitHub сервису.
6. Дискутујете и можда наставите да комитујете.
7. Власник пројекта споји или затвори захтев за повлачење.
8. Синхронизујете ажурирану мастер грану назад у своју рачву.

У основи, ово је процес рада са руководиоцем интеграције који смо описали у [Процес рада са руководиоцем интеграције](#), али уместо да се користе имејлови за комуникацију и преглед промена, тимови користе GitHub веб алате.

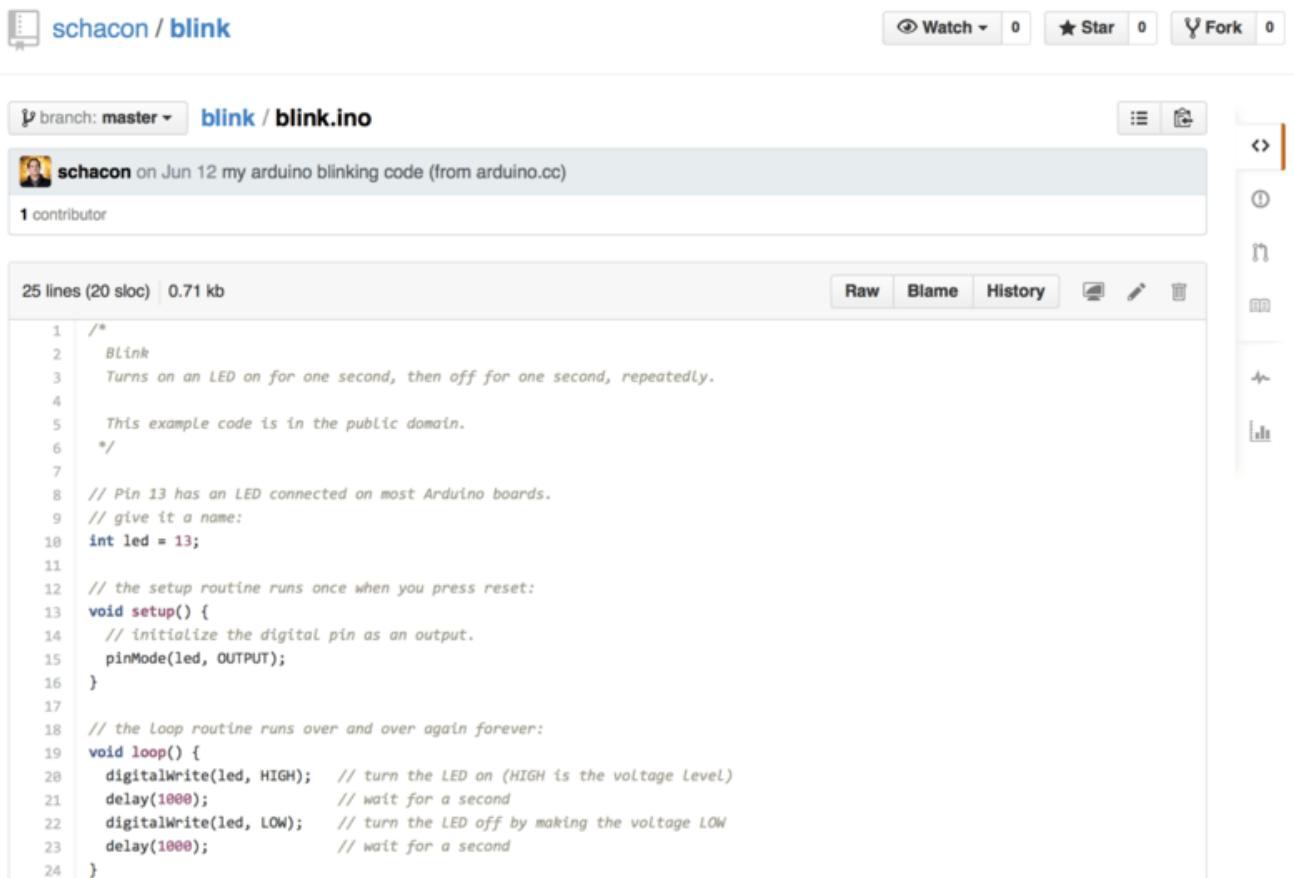
Хајде да прођемо кроз пример предлагања промена пројекту отвореног кода који је хостован на GitHub сервису користећи овај процес рада.



Уместо GitHub веб интерфејса, за већину ствари можете да користите званични **GitHub CLI** алат. Алат може да се користи на Виндоус, МекОС и Линукс системима. Посетите [GitHub CLI почетну страницу](#) за инструкције инсталације и упутство за коришћење.

Како се прави захтев за повлачење

Тони тражи код који хоће да извршава на свом Ардуину програмабилном микроконтролеру и нашао је одличан фајл програма на сервису GitHub на адреси <https://github.com/schacon/blink>.



The screenshot shows the GitHub repository page for the project 'schacon / blink'. At the top, there are buttons for 'Watch' (0), 'Star' (0), and 'Fork' (0). Below that, it says 'branch: master' and 'blink / blink.ino'. It shows a commit by 'schacon' from Jun 12: 'my arduino blinking code (from arduino.cc)'. There is one contributor listed. The code listing shows 25 lines (20 sloc) of 0.71 kb. The code itself is:

```
1 //*
2 * Blink
3 * Turns on an LED on for one second, then off for one second, repeatedly.
4 *
5 * This example code is in the public domain.
6 */
7
8 // Pin 13 has an LED connected on most Arduino boards.
9 // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14     // initialize the digital pin as an output.
15     pinMode(led, OUTPUT);
16 }
17
18 // the Loop routine runs over and over again forever:
19 void loop() {
20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
21     delay(1000);              // wait for a second
22     digitalWrite(led, LOW);   // turn the LED off by making the voltage LOW
23     delay(1000);              // wait for a second
24 }
```

Слика 89. Пројекат коме жељимо да дамо допринос

Једини проблем је што је брзина трептања сувише велика. Мислимо да би било много боље да се код сваке промене стања чека три секунде уместо једне. Зато хајде да унапредимо програм и да га пошаљемо назад на пројекат као предложену промену.

Најпре кликнемо на дугме 'Fork' које смо раније поменули да добијемо своју личну копију пројекта. Овде је наше корисничко име „tonychacon” тако да је наша копија овог пројекта на адреси <https://github.com/tonychacon/blink> и ту можемо да је уређујемо. Клонираћемо пројекат локално, направити тематску грану, променити код и коначно гурнути те промене назад на 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 Клонирамо нашу рачву пројекта локално.

② Направимо описну тематску грану.

③ Изменимо кôд.

④ Проверимо да ли је промена добра.

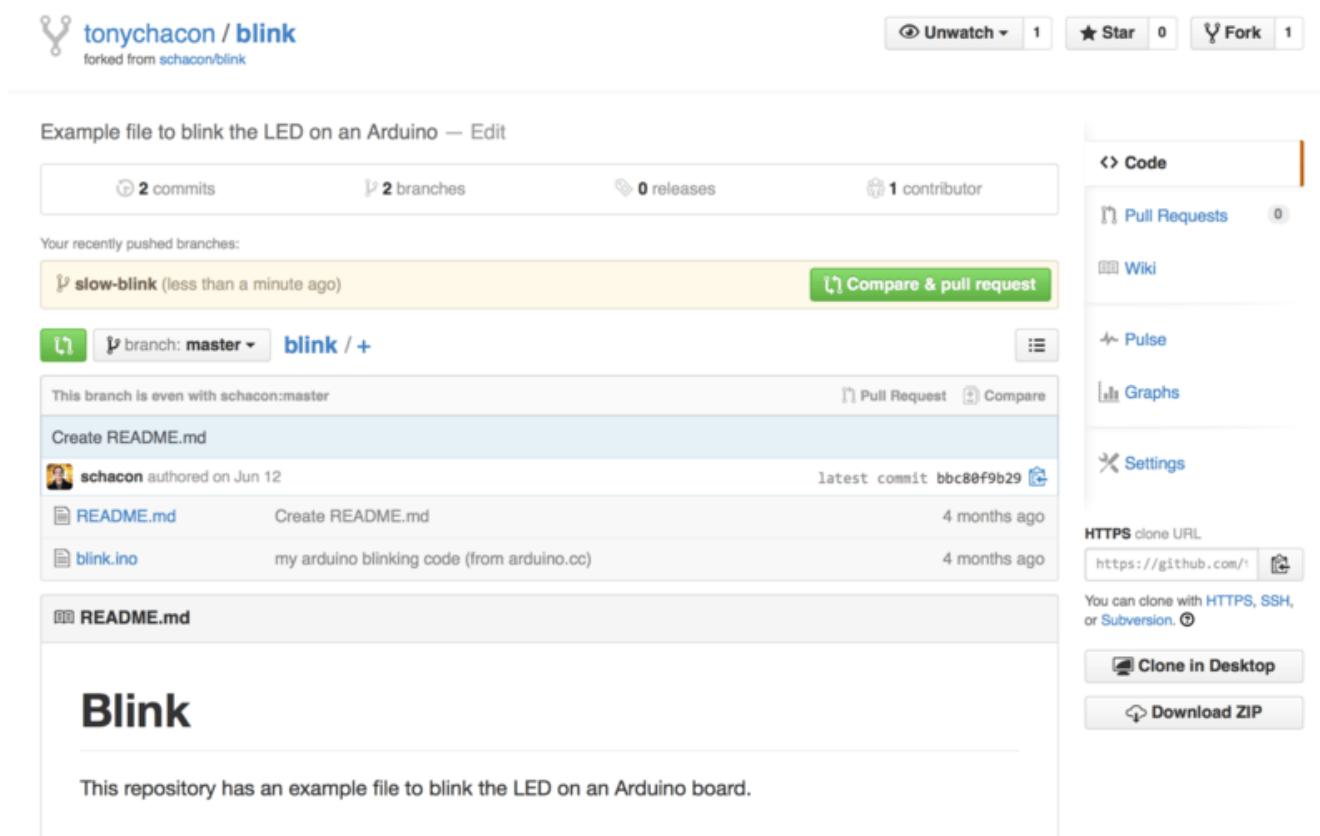
⑤ Комитујемо промену на тематску грану.

⑥ Гурнемо нову тематску грану назад на нашу GitHub рачву.

Ако се сада вратимо на нашу GitHub рачву, видећемо да је GitHub приметио да смо гурнули

нову тематску грани и приказује нам велико зелено дугме да одјавимо промене и отворимо захтев за повлачење ка првобитном пројекту.

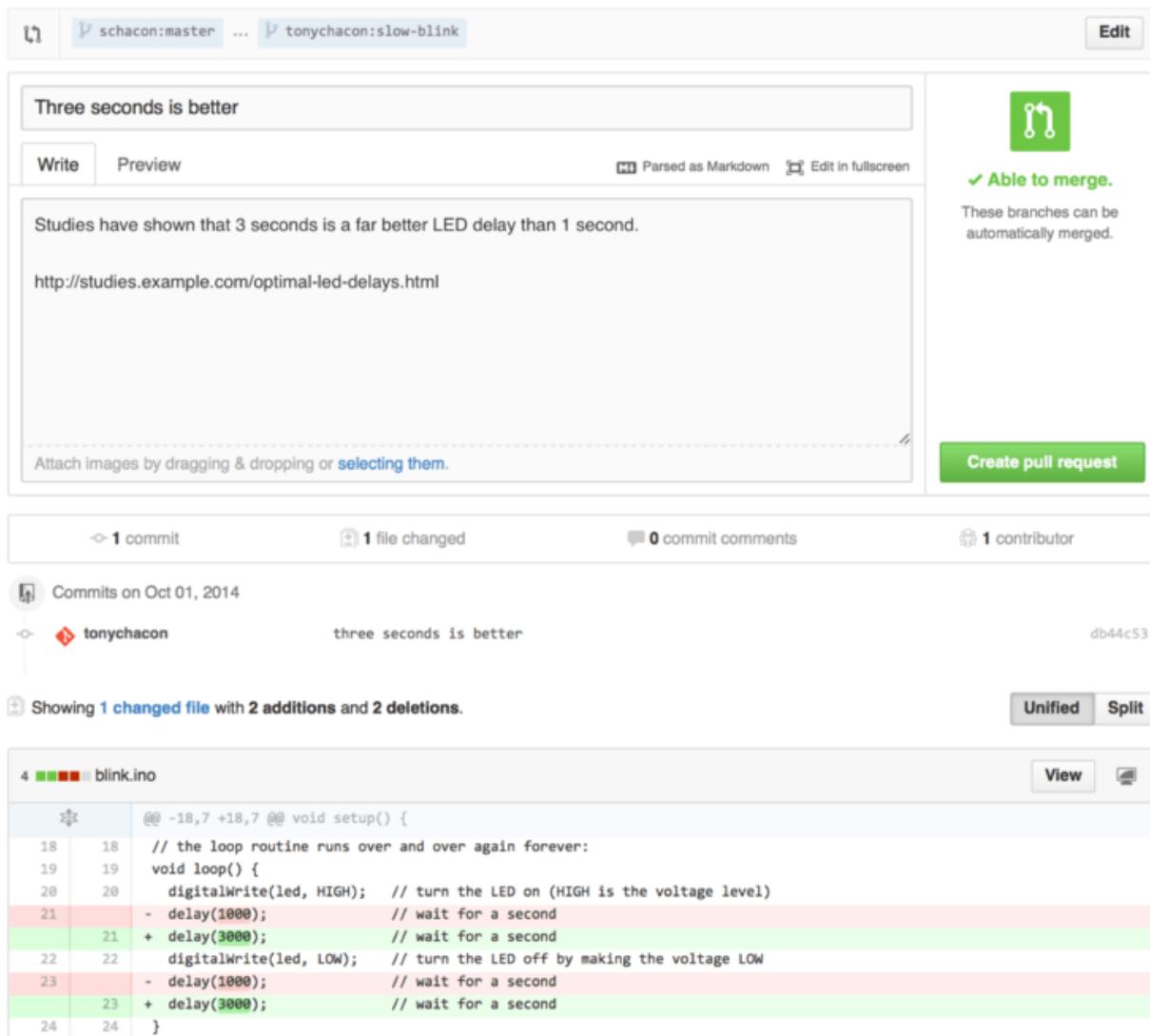
Други начин је да одете на страницу „*Branches*” на <https://github.com/<корисник>/<пројекат>/branches> да лоцирате своју грани и отворите нов захтев за повлачење одатле.



Слика 90. Дугме Pull Request

Ако кликнемо на то зелено дугме, видећемо екран који нас пита да нашем захтеву за повлачење дамо наслов и опис. Скоро увек вреди потрудити се мало за ово, пошто добар опис помаже власнику првобитног пројекта да схвати шта желите да урадите, да ли су ваше предложене промене исправне и да ли ће прихватање промена побољшати првобитни пројекат.

Можемо да видимо и листу комитова на нашој тематској грани који су „испред” **master** грани (у овом случају, само један) и уједињену разлику свих промена које ће бити направљене ако власник пројекта одлучи да споји ову грани.



The screenshot shows a GitHub pull request page. At the top, there's a header with the repository name 'tonychacon / blink' and a 'forked from schacon/blink' note. To the right are buttons for 'Unwatch', 'Star', and 'Fork'. Below the header, the pull request details are shown: 'schacon:master' is merged into 'tonychacon:slow-blink'. The title of the pull request is 'Three seconds is better'. The description states: 'Studies have shown that 3 seconds is a far better LED delay than 1 second.' and provides a link: 'http://studies.example.com/optimal-led-delays.html'. On the right side, there's a green icon indicating 'Able to merge' and a note: 'These branches can be automatically merged.' A 'Create pull request' button is also visible. Below the title, there are summary statistics: 1 commit, 1 file changed, 0 commit comments, and 1 contributor. The commit log shows a single commit from 'tonychacon' on Oct 01, 2014, with the message 'three seconds is better'. The commit hash is db44c53. At the bottom, it says 'Showing 1 changed file with 2 additions and 2 deletions.' and has 'Unified' and 'Split' view options. The diff view shows the changes in the 'blink.ino' file:

```

diff --git a/blink.ino b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
 18   18   // the loop routine runs over and over again forever:
 19   19   void loop() {
 20     20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
- 21       21       // wait for a second
+ 21       21       // wait for a second
 22     22     digitalWrite(led, LOW);   // turn the LED off by making the voltage LOW
- 23       23       // wait for a second
+ 23       23       // wait for a second
 24   24   }

```

Слика 91. Страница за прављење захтева за повлачење

Када на овом екрану кликнете на дугме '*Create pull request*', власник пројекта који сте рачвали ће добити обавештење да неко предлаже промену и добиће линк до странице која има све информације у вези тога.

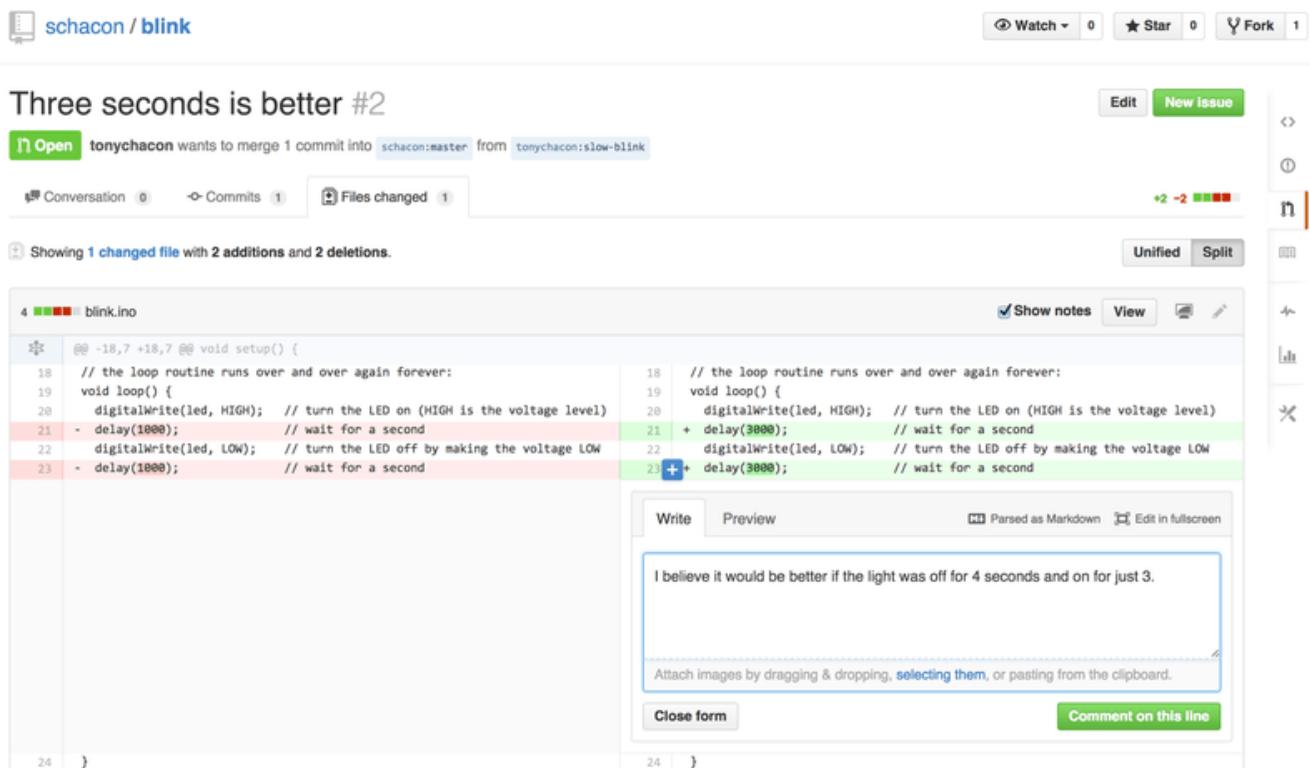


Мада се захтеви за повлачење најчешће користе за јавне пројекте као што је овај када сарадник има комплетну промену која је спремна да се примени, користе се често и код интерних пројекта *на почетку циклуса развоја*. Пошто можете да наставите да гурате на тематске гране чак и **након** што се отвори захтев за повлачење, он се често отвара рано и користи се као начин за тимску итерацију над радом у контексту, уместо да се отвори на самом крају процеса.

Итерација над захтевом за повлачење

Сада власник пројекта може да погледа предложену промену и да је споји, да је одбије, или да је коментарише. Рецимо да му се допада идеја, али му се више свиђа да светло буде искључено мало дуже него што је укључено.

Овакав разговор би се обављао имејловима у процесима рада који су представљени у [Дистрибуирани Гит](#), али се на GitHub сервису ово дешава онлајн. Власник пројекта може да прегледа уједињену разлику и остави коментар кликом на било коју линију.



Слика 92. Коментарисање одређене линије кода у захтеву за повлачење

Када одржавалац направи овај коментар, особа која је отворила захтев за повлачење (заправо, свако ко прати репозиторијум) добиће обавештење. Касније ћемо показати како променити ова подешавања, али ако су му била укључена имејл обавештења, Тони ће добити овакав имејл:



Слика 93. Коментари послати као имејл обавештења

Свако може да остави опште коментаре на захтев за повлачење. У [Страница за дискусију захтева за повлачење](#) видимо пример где власник пројекта коментарише линију кода, а онда оставља општи коментар у одељку за дискусију. Можете видети да се коментари о коду такође довлаче и у овај разговор.

Three seconds is better #2

Edit New issue

Open tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1 +2 -2

tonychacon commented 6 minutes ago
Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

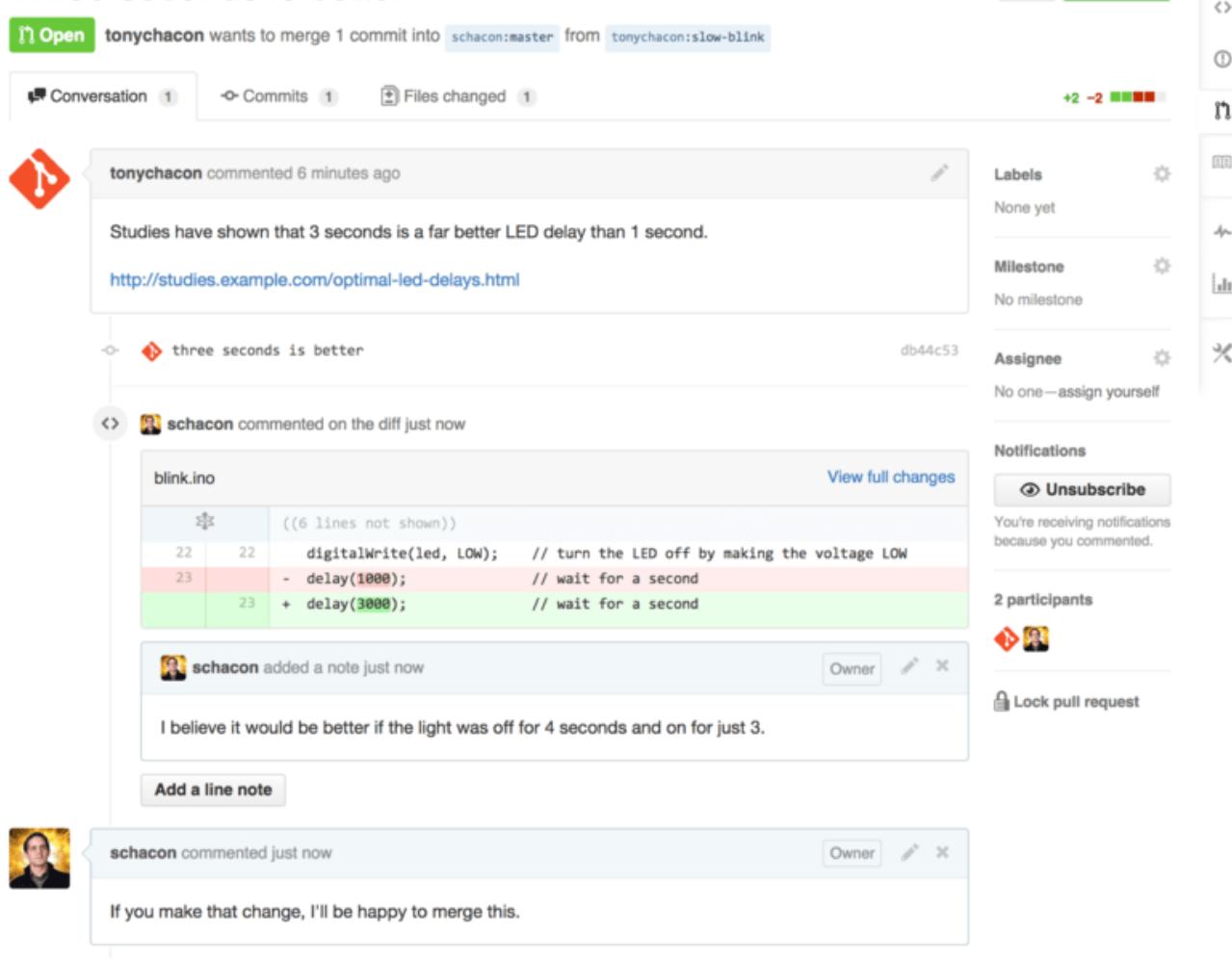
schacon commented on the diff just now
blink.ino View full changes
((6 lines not shown))
22 22 digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23 - delay(1000); // wait for a second
23 + delay(3000); // wait for a second

schacon added a note just now
I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

schacon commented just now
If you make that change, I'll be happy to merge this.

Labels None yet
Milestone No milestone
Assignee No one—assign yourself
Notifications Unsubscribe You're receiving notifications because you commented.
2 participants Lock pull request



Слика 94. Страница за дискусију захтева за повлачење

Сада сарадник може да види шта треба да уради да би се промена прихватила. Срећом, ово је веома једноставно. Док бисте преко имејлова морали да поново смотрате серију закрпи и пошаљете је поново на мејлинг листу, са GitHub сервисом можете једноставно да поново комитујете на тематској грани и да гурнете промене које ће аутоматски ажурирати захтев за повлачење. У [Коначни захтев за повлачење](#) такође можете да видите и то да је стари коментар на коду сакривен у ажурираном захтеву за повлачење, пошто је постављен на линији која се од тада изменила.

Додавање комитова на постојећи захтев за повлачење не окида обавештавање, тако да након што Тони гурне своје исправке, одлучује да остави коментар како би обавестио власника пројекта да је направио захтевану промену.

Three seconds is better #2

The screenshot shows a GitHub pull request interface. At the top, a green button says "Open" and the title is "tonychacon wants to merge 3 commits into schacon:master from tonychacon:slow-blink". Below this, there are tabs for "Conversation" (3), "Commits" (3), and "Files changed" (1). The main area shows a conversation between tonychacon and schacon. tonychacon commented 11 minutes ago: "Studies have shown that 3 seconds is a far better LED delay than 1 second. http://studies.example.com/optimal-led-delays.html". schacon commented on an outdated diff 5 minutes ago: "If you make that change, I'll be happy to merge this.". tonychacon added some commits 2 minutes ago: "longer off time" (commit db44c53) and "remove trailing whitespace" (commit ef4725c). tonychacon commented 10 seconds ago: "I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?". At the bottom right, there is a green button labeled "Merge pull request".

Слика 95. Коначни захтев за повлачење

Занимљива ствар коју можете приметити је да ако на овом захтеву за повлачење кликнете на картицу „Files Changed”, добићете „уједињену” разлику—другим речима, укупну нагомилану разлику која би била уведена у вашу главну грану ако би се ова тематска грана спојила са њом. Ако посматрамо `git diff`, ово вам у суштини автоматски покаже `git diff master…<грана>` за грану на којој је базиран захтев за повлачење. Погледајте [Како утврдити шта је уведено](#) за више информација о овој врсти разлике.

Друга ствар коју ћете приметити је то да GitHub проверава да ли се захтев за повлачење глатко спаја и нуди дугме које уместо вас ради спајање на серверу. Ово дугме се појављује само ако имате дозволу писања у репозиторијум и ако је могућ тривијални спој. Ако кликнете на њега, GitHub ће извршити „не премотавај унапред” спајање, што значи да чак и ако би спајање **могло** да буде премотавање унапред, ипак ће се направити нови комит спајања.

Ако вам више одговара, можете и једноставно да повучете грану и обавите спајање локално. Ако спојите ову грану у `master` грану и гурнете то на GitHub, захтев за повлачење ће се аутоматски затворити.

Ово је основни процес рада који користи већина GitHub пројектата. Праве се тематске гране, над њима се отварају се захтеви за повлачење, следи дискусија, евентуално се ради још мало на грани и на крају се захтев или затвори или споји.

Не само рачве



Важно је приметити да можете отворити и захтев за повлачење између две гране у истом репозиторијуму. Ако са неком особом радите на могућности и обоје имате дозволу за писање по пројекту, можете да гурнете тематску грану на репозиторијум и отворите захтев за повлачење у `master` грану тог истог пројекта да бисте покренули преглед кода и дискусију. Нема потребе за рачвањем.

Напредни захтеви за повлачење

Сада када смо показали основе давања доприноса пројекту на сервису GitHub, хајде да погледамо неколико занимљивих савета и трикова о захтевима за повлачење тако да можете бити ефикаснији док их користите.

Захтеви за повлачење као закрпе

Важно је разумети да многи пројекти не гледају на захтеве за повлачење као на низ савршених закрпа које чисто треба да се примене тим редом, као што већина пројектата базираних на мејлинг листи гледа на доприносе у низу закрпи. Већина GitHub пројектата гледа на гране за које се захтева повлачење као итеративне разговоре о предложеној промени, који кулминирају у јединствену разлику која се прихвата спајањем.

Ово је важна разлика, јер се у општем случају промена предлаже пре него што се сматра да је код савршен, што је много ређи случај код доприноса базираних на низу закрпа са мејлинг листе. Ово омогућава правовремени разговор са одржаваоцима тако да је долазак до доброг решења углавном плод тимског рада заједнице. Када се код предложи захтевом за повлачење и одржаваоци или заједница предложе промену, низ закрпа се у општем случају не смотава поново, већ се разлика гурне као нови комит на грани, водећи разговор даље тако да је контекст претходног рада недирнут.

На пример, ако се вратите назад и опет погледате [Коначни захтев за повлачење](#), приметићете да сарадник није ребазирао свој комит и послao још један захтев за повлачење. Уместо тога, додао је нове комитове и гурнуо их на постојећу грану. На овај начин, ако се касније вратите назад и погледате овај захтев за повлачење, моћи ћете лако да нађете контекст због кога су донете одлуке. Ако најавите кликните на дугме „Merge”, намерно се прави комит спајања који указује на захтев за повлачење тако да лако можете да се вратите назад и истражите оригиналну дискусију, ако то буде потребно.

Како одржати корак са узводним гранама

Ако ваш захтев за повлачење постане застарео или се из неког другог разлога не спаја глатко, то треба да поправите како би одржавалац могао лако да га споји. GitHub ће вам ово тестирати и обавестиће вас на дну сваког захтева за повлачење о томе да ли је спој тривијалан или не.



Слика 96. Захтев за повлачење се не спаја глатко

Ако видите нешто као [Захтев за повлачење се не спаја глатко](#), треба да поправите своју грану тако да постане зелена и да одржавалац не мора да ради додатни посао.

Ово можете урадити на два главна начина. Можете или да ребазирате своју грану на врх одредишне (обично је то `master` грана репозиторијума који сте рачвали), или да спојите одредишну грану у своју грану.

Већина програмера на сервису GitHub ће изабрати другу опцију, из истих разлога које смо прешли у претходном одељку. Оно што је важно је историја и коначно спајање, тако да вам ребазирање не даје много тога сем нешто чистије историје, а с друге стране је **много** компликованије и подложније грешкама.

Ако желите да спојите одредишну грану како би ваш захтев за повлачење могао да се споји, треба да додате првобитни репозиторијум као нови удаљени репозиторијум, преузмете (*fetch*) податке са њега, спојите главну грану тог репозиторијума у своју тематску грану, средите све евентуалне проблеме и на крају турнете то назад на исту грану за коју сте отворили захтев за повлачење.

На пример, рецимо да је у примеру „tinychacon” који смо користили малопре, првобитни аутор направи промену која ствара конфликт са захтевом за повлачење. Хајде да прођемо кроз те кораке.

```

$ 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 Додавање првобитног репозиторијума као удаљеног са именом `upstream`.

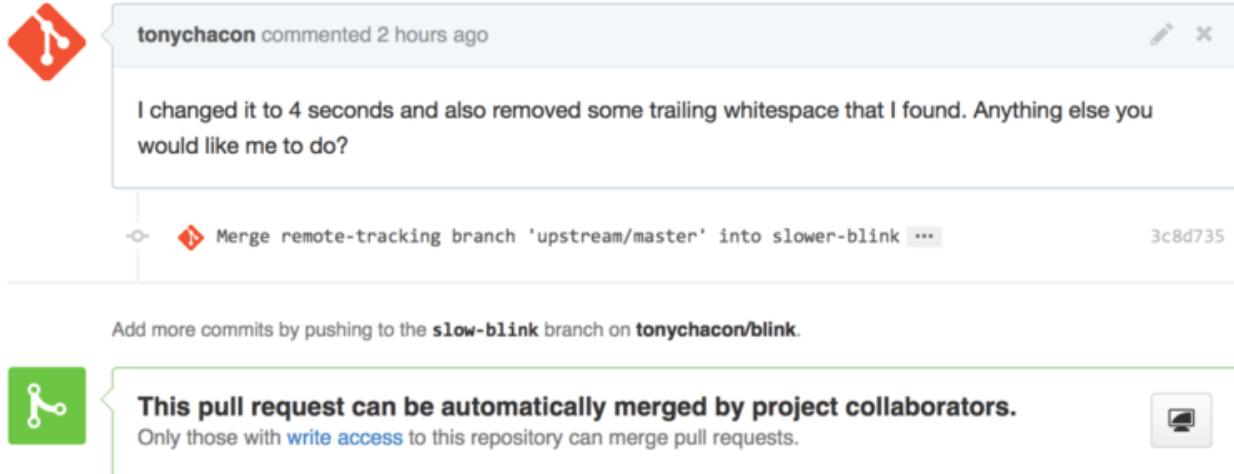
② Преузимање најновијег рада са тог удаљеног репозиторијума.

③ Спајање главне гране у вашу тематску грану.

④ Решавање конфликта који су настали.

⑤ Гурање назад на исту тематску грану.

Када то урадите, захтев за повлачење ће аутоматски бити ажуриран и поново проверен да би се установило да ли се глатко спаја.



Слика 97. Захтев за повлачење се сада спаја глатко

Једна од најбољих ствари у вези програма Гит је то што непрестано можете да радите овакве ствари. Ако имате пројекат који веома дуго траје, једноставно можете да спајате са одредишних грана изнова и изнова, а решавате само конфликте који настану од тренутка када сте последњи пут извршили спајање, што чини процес веома подесним за руковање.

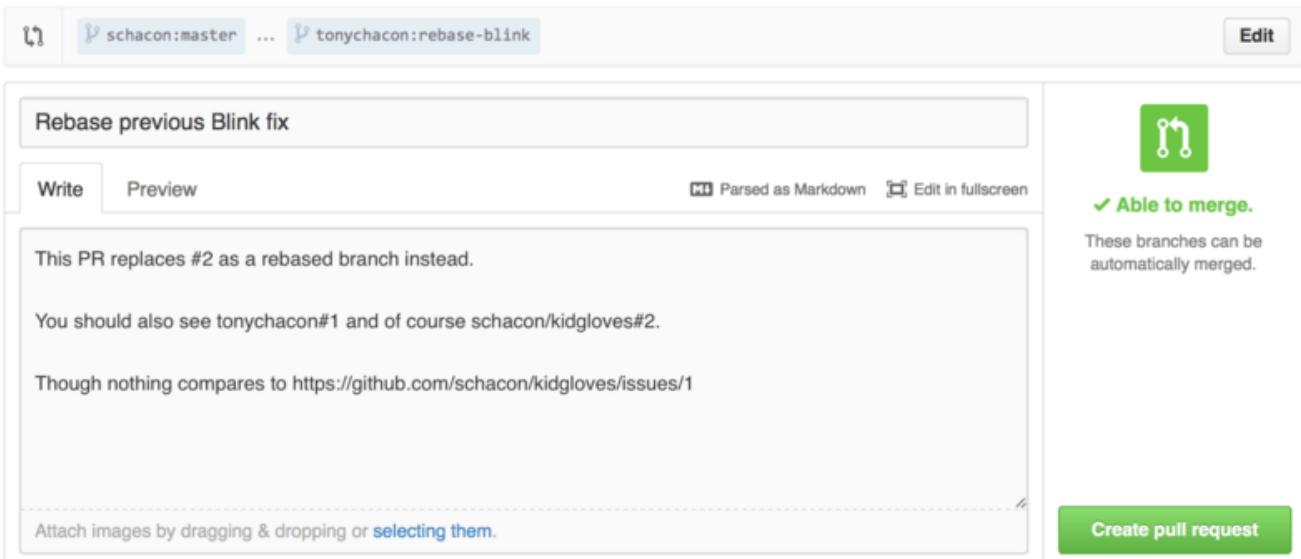
Ако дефинитивно желите да ребазирате грану како бисте је почистили, то наравно можете урадити, али се строго препоручује да не форсирате гурање преко гране на којој је захтев за повлачење већ отворен. Ако су је други људи повукли и радили нешто на њој, долазите до гомиле проблема који су описани у [Опасности ребазирања](#). Уместо тога, гурните ребазирану грану на нову грану на сервису GitHub и отворите потпуно нови захтев за спајањем који указује на стари, па онда затворите првобитни.

Референце

Ваше следеће питање би могло да буде „Како да укажем на стари захтев за повлачење?“. Испоставља се да постоји огроман број начина да укажете на скоро све што било где пишете на сервису GitHub.

Почнимо од тога како да унакрсно укажете на други захтев за повлачење или на тикет (*Issue*). Сви захтеви за повлачење и тикети имају своје бројеве који су јединствени за један пројекат. На пример, не можете имати захтев за повлачење #3 и тикет #3. Ако желите да укажете на захтев за повлачење или тикет са било ког другог, можете једноставно да ставите #<број> у било ком коментару или опису. Можете и да будете одређенији ако тикет или захтев за повлачење живи негде другде; пишите **корисничко-име#<број>** ако указујете на тикет или захтев за повлачење у рачви репозиторијума у којем се налазите, или **корисничко-име/репозиторијум#<број>** да укажете на нешто из неког другог репозиторијума.

Погледајмо пример. Рецимо да само ребазирали грану у претходном примеру, направили нови захтев за повлачење за њу, па сада желимо да укажемо на стари захтев за повлачење из новог. Такође желимо да укажемо на тикет у рачви репозиторијума и тикет у потпуно другом пројекту. Можемо да попунимо опис баш као што је то учињено у [Унакрсне референце у захтеву за повлачење](#).



Слика 98. Унакрсне референце у захтеву за повлачење

Када пошаљемо овај захтев за повлачење, видећемо да се све то приказује као [Приказане унакрсне референце у захтеву за повлачење](#).

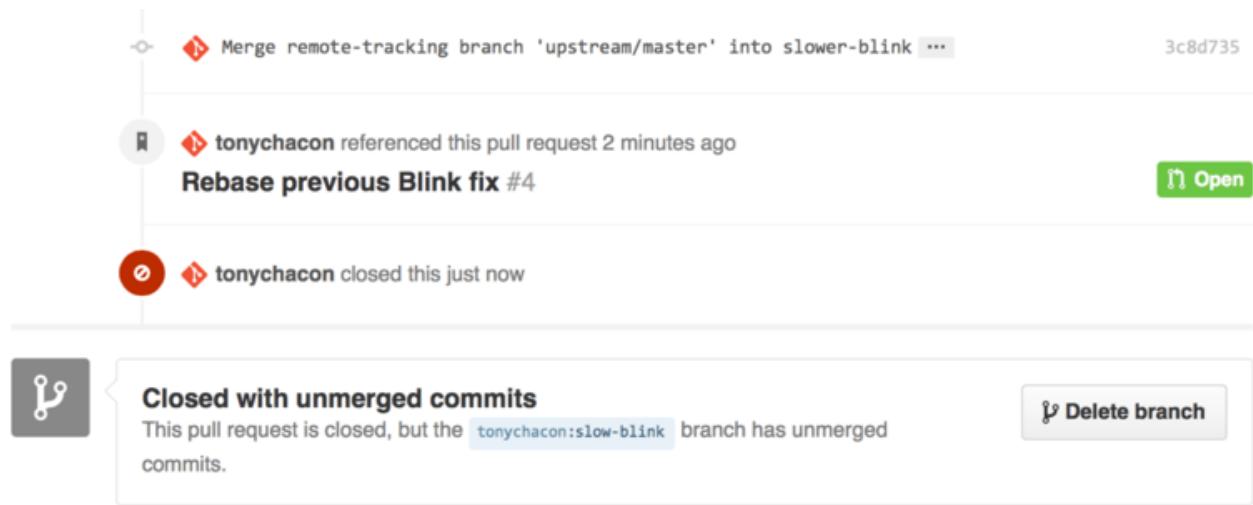
Rebase previous Blink fix #4

The screenshot shows the detailed view of the pull request. At the top, there is an 'Open' button, the author's name 'tonychacon', and the message 'wants to merge 2 commits into schacon:master from tonychacon:rebase-blink'. Below this, there are three navigation links: 'Conversation 0', 'Commits 2', and 'Files changed 1'. In the main content area, a comment from 'tonychacon' is shown: 'commented just now'. The comment text is identical to the one in the previous screenshot. Below the comment, there is a list of commits added by 'tonychacon': 'added some commits 4 hours ago', 'three seconds is better' (commit hash: afe904a), and 'remove trailing whitespace' (commit hash: a5a7751).

Слика 99. Приказане унакрсне референце у захтеву за повлачење

Приметите да је пун GitHub URL који смо унели скраћен само на неопходне информације.

Ако се сада Тони врати назад и затвори првобитни захтев за повлачење, то можемо да видимо његовим помињањем у новом, сервис GitHub аутоматски креира догађај праћења уназад (*trackback event*) у временској линији захтева за повлачење. Ово значи да ће свако ко посети овај захтев за повлачење и види да је затворен лако моћи да дође до оног којим је замењен. Линк ће изгледати отприлике као на [Линк уназад на нови захтев за повлачење у временској линији затвореног захтева за повлачење](#).



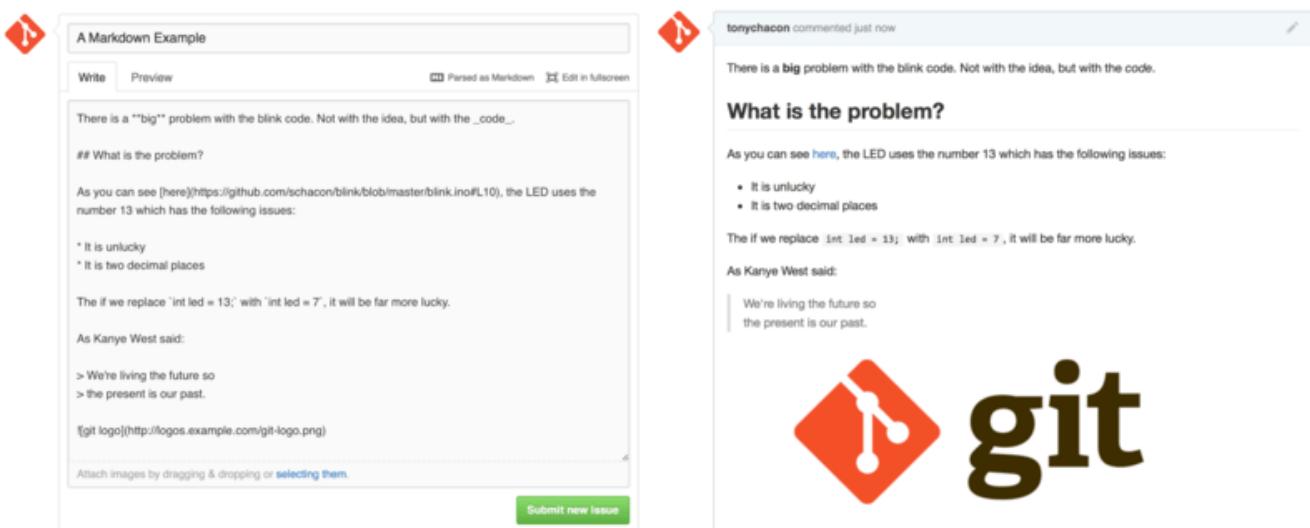
Слика 100. Линк уназад на нови захтев за повлачење у временској линији затвореног захтева за повлачење

Поред бројева тикета, на одређени комит можете указати са SHA-1. Морате да наведете комплетан SHA-1 са 40 карактера, али ако сервис GitHub то види у коментару, линковаће га директно на комит. Опет, можете да указујете на комитове у рачвама или другим репозиторијума на исти начин као са што се ради у тикетима.

Маркдаун са GitHub укусом

Линковање других тикета је само почетак занимљивих ствари које можете радити са скоро било којим текстуалним пољем на сервису GitHub. У описима тикета и захтева за повлачење, коментарима, коментарима у коду и на многим другим местима, можете да користите нешто што се зове Маркдаун са GitHub укусом (*GitHub Flavored Markdown*). Маркдаун је као писање обичног текста који се затим приказује са стиловима.

Погледајте [Пример Маркдауна са GitHub укусом како је написан и како се приказује](#) за пример како се коментари или текст могу написати а онда приказати користећи Маркдаун.



Слика 101. Пример Маркдауна са GitHub укусом како је написан и како се приказује

Маркдаун са GitHub укусом додаје још ствари које можете да урадите поред основне Маркдаун синтаксе. Све оне могу бити веома корисне када правите коментаре или описе за

захтеве за повлачење или тикете.

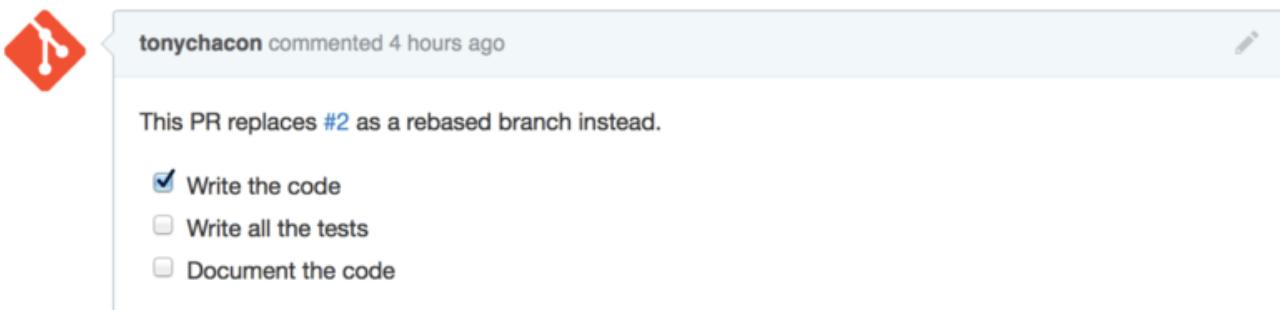
Листа задатака

Прва веома корисна одлика GitHub специфичног Маркдауна, поготово код захтева за повлачење, јесте листа задатака. Листа задатака је листа ствари са пољима за штиклирање које желите да буду урађене. Када их ставите у тикет или захтев за повлачење, то обично означава ствари које желите да се ураде пре него што ставка може да се сматра као завршена.

Листу задатака можете да направите овако:

- [X] Write the code
- [] Write all the tests
- [] Document the code

Ако ово укључимо у опис нашег захтеву за повлачење или у опис тикета, видећемо да се приказује као [Листа задатака приказана у Маркдаун коментару](#).



Слика 102. Листа задатака приказана у Маркдаун коментару

Ово се често користи у захтевима за повлачење ради указивања на све оно што бисте желели да се обави на грани пре него што захтев за повлачење буде спреман за спајање. Оно што је овде стварно супер је то што једноставним кликом на поља за штиклирање ажурирате коментар — не морате директно да уређујете Маркдаун да бисте штиклирали задатке.

Штавише, GitHub ће потражити листе задатака у вашим тикетима и захтевима за повлачење и приказаће их као метаподатке на страницама које их исписују. На пример, ако имате захтев за повлачење са задацима и баците поглед на страницу која представља преглед свих захтева за повлачење, видећете докле се стигло са радом. Ово помаже људима да разбију захтеве за повлачење у мање задатке и помаже другим људима да прате напредовање гране. Пример овога можете видети у [Сажетак листе задатака у листи захтева за повлачење](#).

Слика 103. Сажетак листе задатака у листи захтева за повлачење

Ово је невероватно корисно када рано отворите захтев за повлачење и користите га да пратите свој напредак имплементације могућности.

Исечци кода

Такође можете да додате и исечке кода у коментаре. Ово је посебно корисно када желите да представите нешто што би могли да пробате пре него што се стварно имплементира као комит на вашој грани. Ово се такође често користи да се дода пример кода који не ради или шта би овај захтев за повлачење могао да имплементира.

Да бисте додали исечак кода, треба да га „оградите“ краткоузлазним акцентима (`).

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```

```

Ako dodate ime језика као што smo gore učinili sa `java`, GitHub ће probati i da истакне sintaksu isečka. U slučaju gornjeg primjera, prikaz bi izgledao као [Приказан пример ограђеног исечка кода](#).

Слика 104. Приказан пример ограђеног исечка кода

Цитирање

Ако одговарате на мањи део дугачког коментара, можете селективно да цитирате коментар тако што ћете линије почети карактером >. Заправо, ово је толико корисно и толико често да постоји и пречица на тастатури. Ако у коментару обележите текст на који желите директно да одговорите и притисните тастер `r`, пречица ће у пољу коментара да цитира тај

текст уместо вас.

Цитати изгледају отприлике овако:

> 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?

Када се прикаже, коментар ће изгледа као [Приказ примера цитата](#).



schacon commented 2 minutes ago

Owner

That is the question—
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune,
Or to take Arms against a Sea of troubles,
And by opposing, end them? To die, to sleep—
No more; and by a sleep, to say we end
The Heart-ache, and the thousand Natural shocks
That Flesh is heir to?



tonychacon commented 10 seconds ago

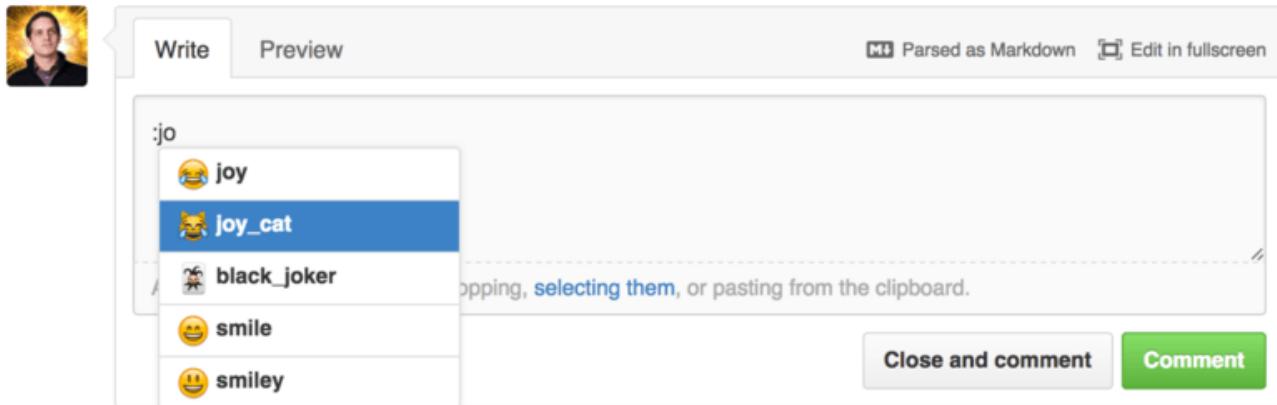
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?

Слика 105. Приказ примера цитата

Емођи

За крај, у коментарима можете да користите и емођије. Ово се заправо врло често користи у коментарима које видите на многим GitHub тикетима и захтевима за повлачење. Постоји чак и помоћник за емођије на сервису GitHub. Ако куцате коментар и почнете са карактером :, аутоматски довршавач ће вам помоћи да пронађете оно што тражите.

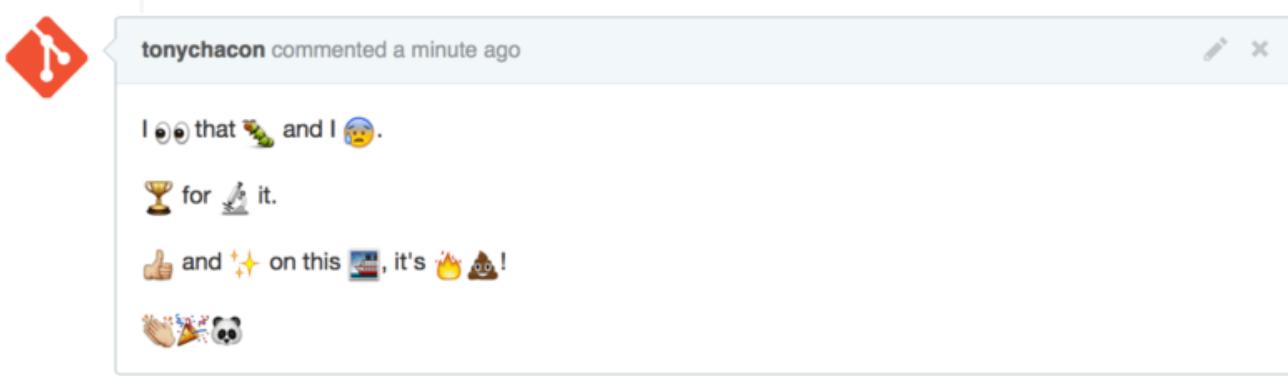


Слика 106. Аутоматски довршавач за емођије у акцији

Емођији имају облик :<име>: било где у коментару. На пример, можете да напишете нешто овако:

```
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:
```

Када се прикаже, изгледаће отприлике као [Коментар са пуно емођија](#).



Слика 107. Коментар са пуно емођија

Ово није нешто много корисно, али додаје елемент забаве и емоција медијуму којим се иначе тешко исказује емоције.

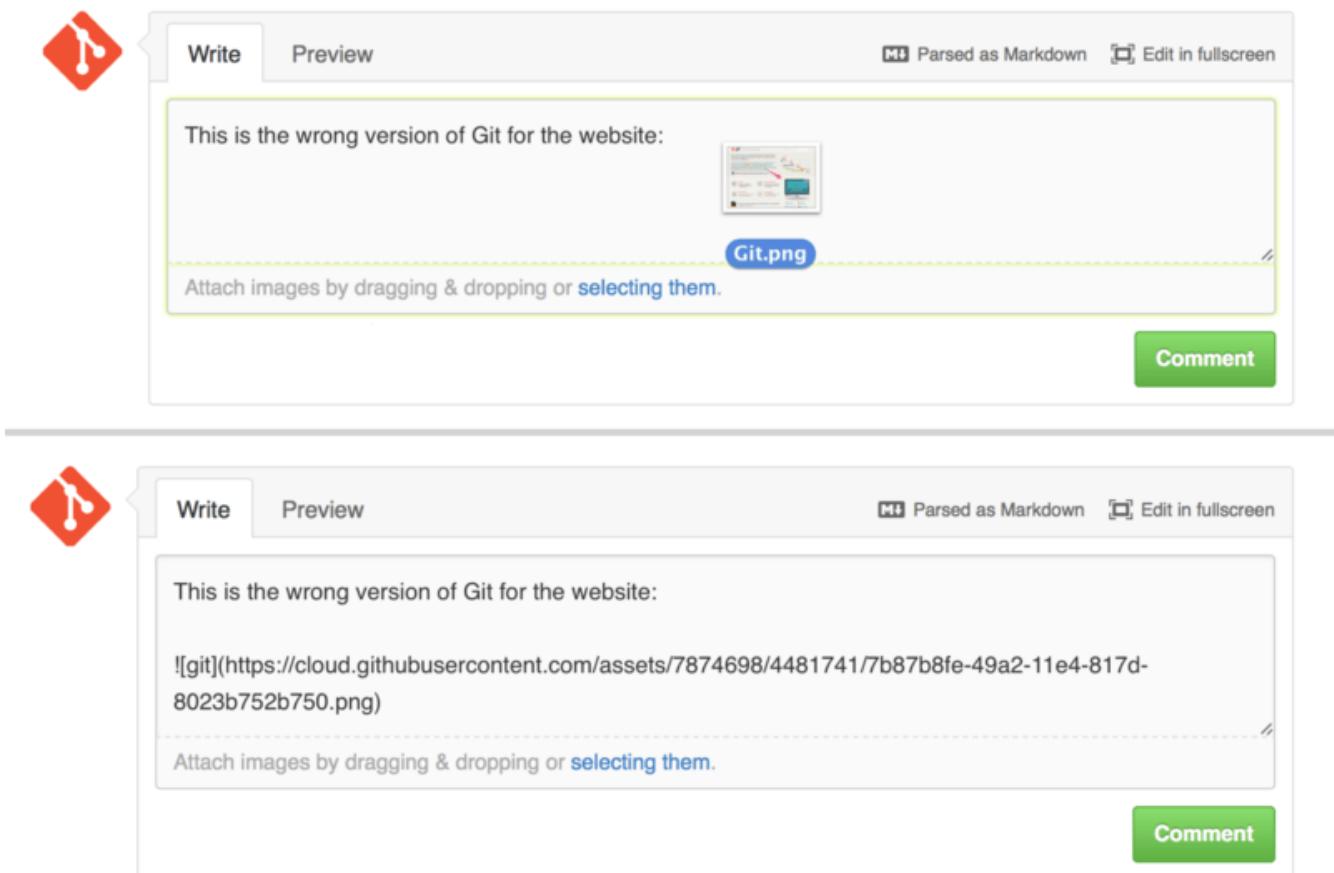


Ових дана заправо постоји велики број веб сервиса који користе емођи карактере. Одличан списак на који се можете позвати да нађете емођи који осликава оно што желите да кажете налази се на:

<https://www.webfx.com/tools/emoji-cheat-sheet/>

Слике

Технички ово није Маркдаун са GitHub укусом, али је невероватно корисно. Поред додавања линкова ка slikama у коментарима, што можете бити тешко за налажење и угађивање URL адреса, GitHub вам омогућава и да превучете и пустите слике у текст поља и да их тако уградите.



Слика 108. Превуците и пустите слике да бисте их пошаљете на сервер и аутоматски уградите

Ако погледате на [Превуците и пустите слике да бисте их пошаљете на сервер и аутоматски уградите](#), видећете мали савет „Parsed as Markdown” изнад текст поља. Клик на то ће вам приказати велики пано са свиме што можете урадити са Маркдауном на сервису GitHub.

Одржавање вашег јавног GitHub репозиторијума ажуруним

Једном када рачвате GitHub репозиторијум, ваш репозиторијум (ваша „рачва”) постоји независно од оригиналног. Тачније, када оригинални репозиторијум има нове комитове, GitHub вас обавештава поруком као што је:

This branch is 5 commits behind progit:master.

Али сервис GitHub никада неће аутоматски ажурирати ваш GitHub репозиторијум; то је нешто што морате сами да урадите. На срећу, врло је једноставно.

Једна могућност не захтева никакву конфигурацију. На пример, ако се рачвали из <https://github.com/progit/progit2.git>, своју `master` грану можете да одржавате ажуруном на

следећи начин:

```
$ git checkout master ①
$ git pull https://github.com/progit/progit2.git ②
$ git push origin master ③
```

① Ако сте били на некој другој грани, вратите се на `master`.

② Преузмите промене са <https://github.com/progit/progit2.git> и спојите их у `master`.

③ Гурните своју `master` грани на `origin`.

Ово функционише, али је помало незгодно што морате сваки пут да исписујете URL за преузимање. Тај посао можете да аутоматизирујете са мало конфигурисања:

```
$ git remote add progit https://github.com/progit/progit2.git ①
$ git fetch progit ②
$ git branch --set-upstream-to=progit/master master ③
$ git config --local remote.pushDefault origin ④
```

① Додајете изворни репозиторијум и дајете му име. Овде сам изабрао да га назовем `progit`.

② Преузимате референце на грани `progit` репозиторијума, тачније на `master`.

③ Постављате своју `master` грани да преузме из `progit` удаљеног.

④ Дефинишете да подразумевани репозиторијум у који се гура буде `origin`.

Једном када се ово уради, процес рада постаје доста простији:

```
$ git checkout master ①
$ git pull ②
$ git push ③
```

① Ако сте били на некој другој грани, враћате се на `master`.

② Преузимате промене са `progit` и спајате промене у `master`.

③ Гурате своју `master` грани на `origin`.

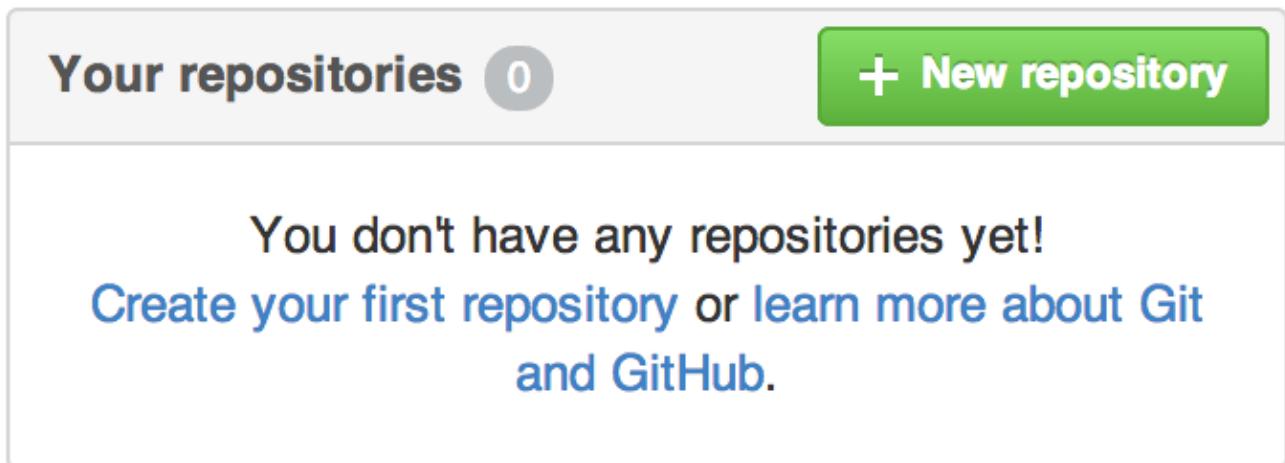
Овај приступ може бити користан, али има и лоших страна. Програм Git ће задовољно одрадити овај посао у тишини, али вас неће упозорити ако направите комит на `master`, повучете са `progit`, па затим гурнете на `origin`—све ове операције су исправне у овако постављеном систему. Тако да морате пазити да никада директно не комитујете на `master`, јер та грана у суштини припада узводном репозиторијуму.

Одржавање пројекта

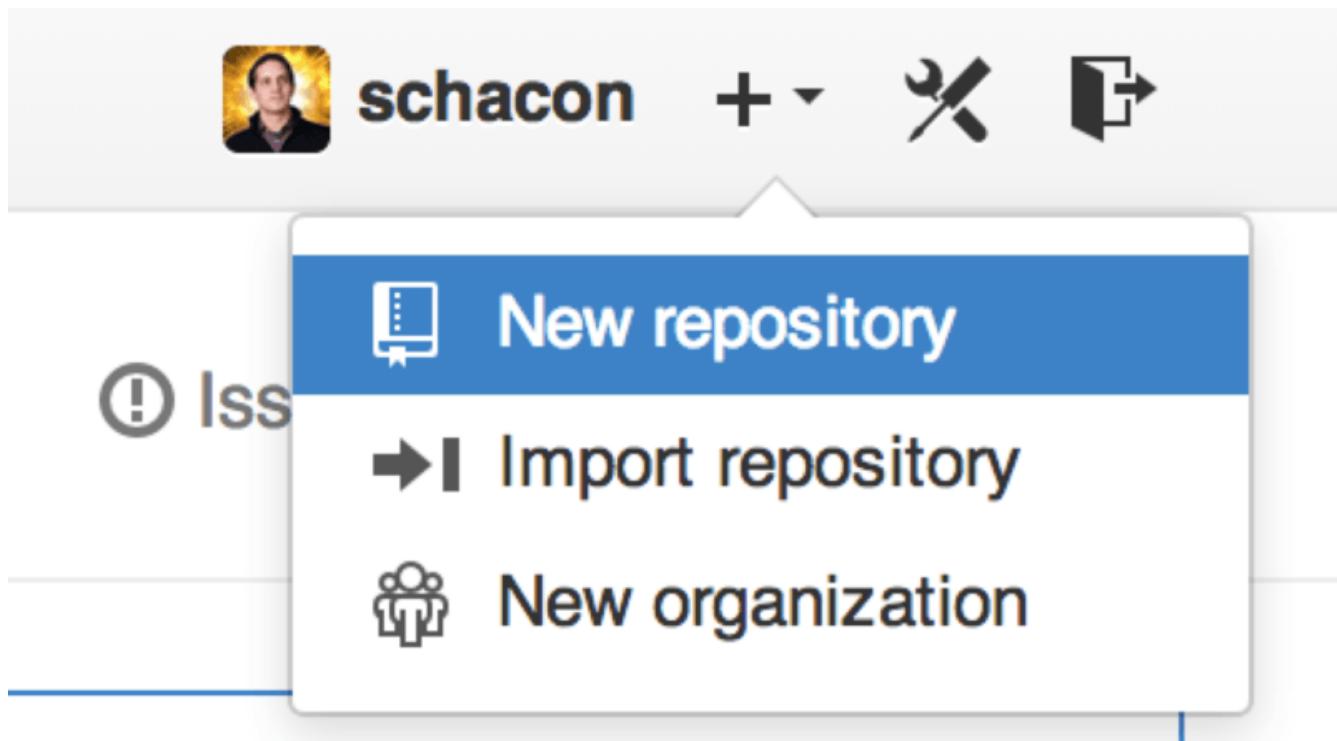
Сада када знамо како се даје допринос пројекту, хайде да погледамо другу страну: креирање, одржавање и администрирање сопственог пројекта.

Креирање новог репозиторијума

Креирајмо нови репозиторијум на коме ћемо делити код нашег пројекта. Почните кликом на дугме „New repository” у горњем десном углу командне табле, или на дугме у алатној траци поред вашег корисничког имена као што се види на [Падајући мени „New repository”](#).



Слика 109. Област „Your repositories”



Слика 110. Падајући мени „New repository”

Ово вас води на форму „new repository”:

Owner Repository name

PUBLIC  ben / iOSApp ✓

Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#).

Description (optional)

iOS project for our mobile group

 **Public**
Anyone can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** | ⓘ

Create repository

Слика 111. Форма new repository

Све што овде стварно треба да урадите је да дате име пројекту; остала поља уопште нису обавезна. Засад, само кликните на дугме „Create Repository”, и бум—имате нов репозиторијум на сервису GitHub, назван [**<корисник>/<име_пројекта>**](#).

Пошто тамо још увек немате никакав код, GitHub ће вам показати упутства како да направите потпуно нов Гит репозиторијум, или да га се повежете са постојећим Гит пројектом. Нећемо се враћати на ово овде; ако треба да се подсетите, баците око на [Основе програма Гит](#).

Сада када се ваш пројекат хостује на сервису GitHub, можете да дате URL сваком с ким желите да поделите пројекат. Сваком пројекту на сервису GitHub се може приступити прекор HTTPS као https://github.com/<корисник>/<име_пројекта>, и преко SSH као git@github.com:<корисник>/<име_пројекта>. Програм Гит може да преузима и да гура на обе ове URL адресе, али се контрола приступа врши на основу акредитације корисника који им приступа.



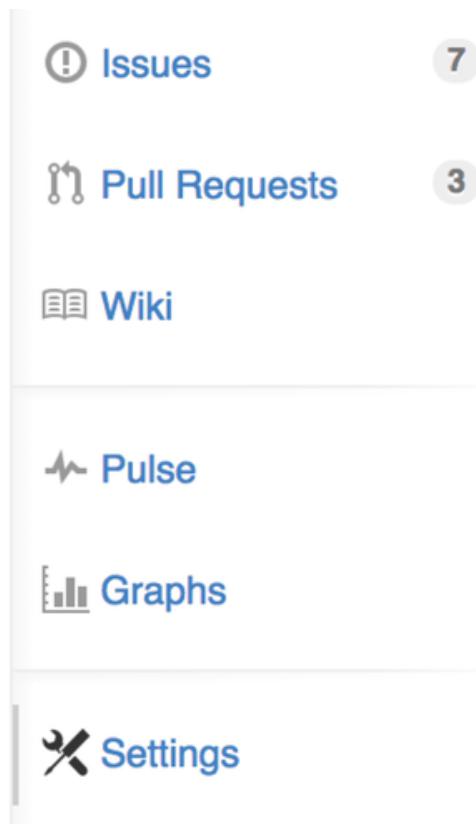
Често је боље да се дели HTTPS URL јавног пројекта, пошто корисник не мора да има GitHub налог да би му приступио у циљу клонирања. Корисници ће морати да имају налог и постављен SSH кључ да приступе вашем пројекту ако им дате SSH URL. HTTP URL је такође потпуно исти URL који би прекопирали у интернет прегледач да у њему погледају пројекат.

Додавање сарадника

Ако радите са другим људима којима желите омогућити да могу комитовати, треба да их додате као „сараднике”. Ако Бен, Џеф и Луиз сви отворе налог на сервису GitHub и желите да им дате приступ за гурање на ваш репозиторијум, можете да их додате на свој пројекат. То ће им дати приступ за „гурање”, што значи да ће моћи и да читају и пишу по пројекту и по

Гит репозиторијуму.

Кликните на линк „*Settings*” на дну траке са десне стране.



Слика 112. Линк за подешавање репозиторијума

Онда из менија са леве стране одаберите „*Collaborators*”. Затим само укуцајте корисничко име у поље за унос, и кликните „*Add collaborator*”. Ово можете да поновите онолико пута колико је потребно да бисте дали приступ свима коме желите. Ако треба да опозовете приступ, само кликните на „X” са десне стране одговорајућег реда.

| Options | Collaborators | Full access to the repository |
|---|--|-------------------------------|
| Collaborators | Ben Straub
ben | X |
| Webhooks & Services | Jeff King
peff | X |
| Deploy keys | Louise Corrigan
LouiseCorrigan | X |
| | Type a username | Add collaborator |

Слика 113. Сарадници репозиторијума

Управљање захтевима за повлачење

Сада када имате пројекат са неким кодом а можда чак и неколико сарадника који имају приступ гурању промена, хајде да видимо шта да радите када добијете захтев за повлачење.

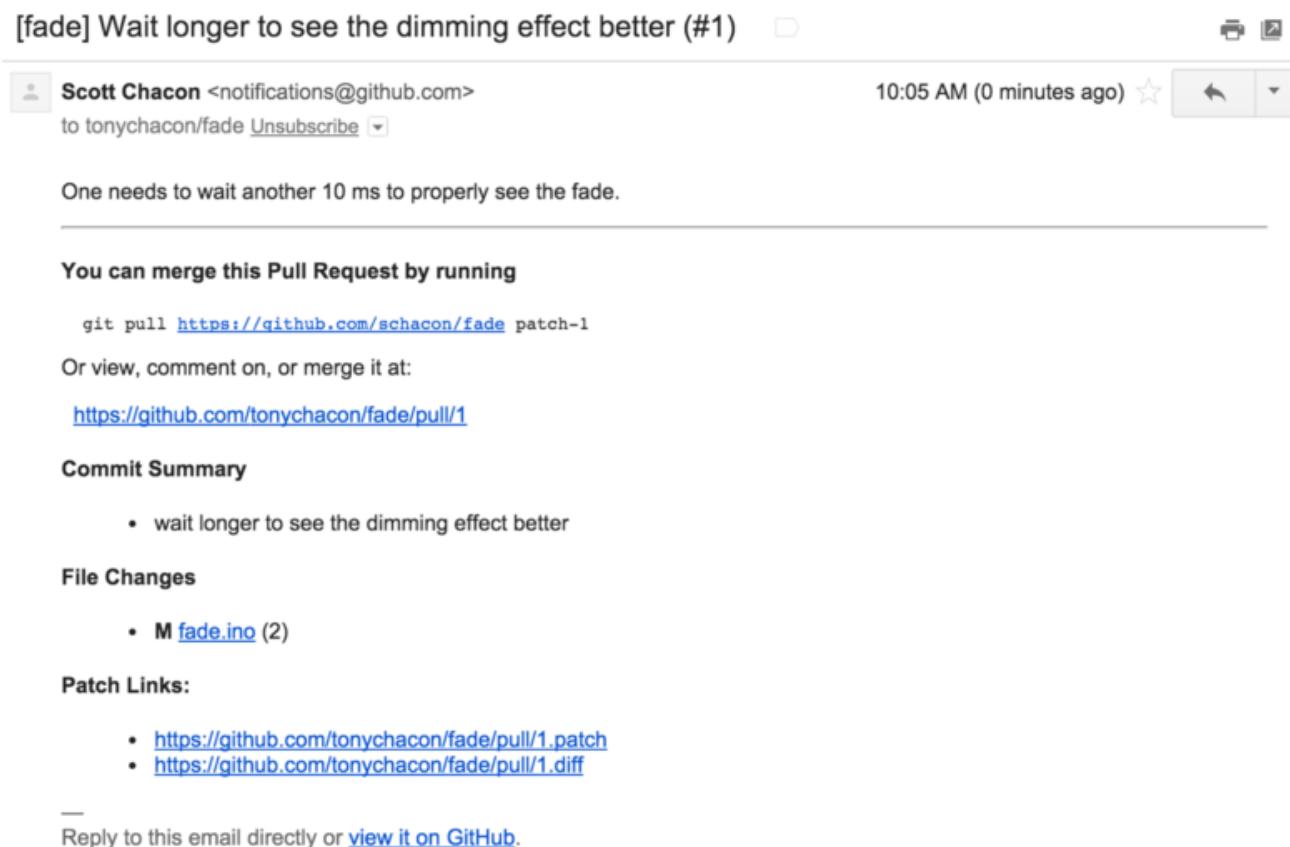
Захтеви за повлачење могу да дођу или са гране у рачви вашег репозиторијума, или са друге

гране из истог репозиторијума. Једина разлика је у томе што они израчве обично припадају другим људима где ви не можете да гурнете на њихову грану и они не могу на вашу, док код интерних захтева за повлачење у општем случају оба лица имају приступ грани.

За ове примере, претпоставићемо да сте ви „*tonychacon*” и да сте направили нови пројекат Ардуино кода који се зове „*fade*”.

Имејл обавештења

Неко дође, направи промене у вашем коду и пошаље вам захтев за повлачење. Требало да добијете имејл који вас обавештава о новом захтеву за повлачење и требало би да изгледа отприлике као [Имејл обавештење о новом захтеву за повлачење](#).



Слика 114. Имејл обавештење о новом захтеву за повлачење

Постоји неколико ствари у вези овог имејла које треба да приметите. Даће вам кратку статистику о разликама — листу фајлова који су се променили у захтеву за повлачење и за колико. Даје вам линк ка захтеву за повлачење на сервису GitHub. Такође вам даје и неколико URL адреса које можете да користите из командне линије.

Ако приметите линију која каже `git pull <url> patch-1`, ово је једноставан начин да се споји удаљена грана без потребе да ручно додајете удаљени репозиторијум. Брзо смо прешли ово у [Одјављивање удаљених грана](#). Ако желите, можете да креирате и скочите на тематску грану, па да онда извршите ову команду да спојите промене из захтева за повлачење.

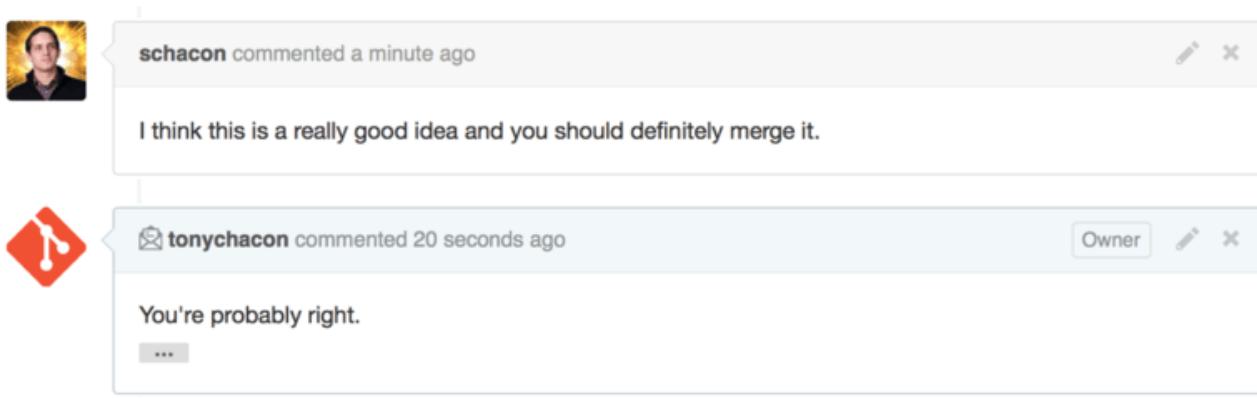
Друге занимљиве URL адресе су `.diff` и `.patch` адресе, које као што погађате, верзију захтева за повлачење као уједињену разлику и закрпу. Технички можете да спојите захтев за повлачење некако овако:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

Сарадња на захтеву за повлачење

Како смо објаснили у [GitHub процес рада](#), сада можете да поразговарате са особом која је отворила захтев за повлачење. Можете да коментаришете одређене линије кода, да коментаришете целе комитове или пак на читав захтев за повлачење, користећи Маркдаун синтаксу са GitHub укусом.

Сваки пут када неко други коментарише на захтев за повлачење, добићете имејл обавештење и тако знате да се дешава нека активност. Сваки од њих ће имати линк ка захтеву за повлачење где се активност догађа, а можете и директно у имејлу да одговорите и тако објавите свој коментар у низ захтева за повлачење.



Слика 115. Одговори на имејлове се укључују у низ

Онда када код буде сређен и када пожелите да га спојите, можете или да повучете код и спојите га локално, или да искористите `git pull <url> <branch>` синтаксу коју смо видели раније, или да рачву додате као удаљени репозиторијум па онда преузимате и спајате.

Ако је спајање тривијалано, можете и само да притиснете дугме „Merge” на GitHub веб сајту. Ово ће урадити „не премотавај унапред” спајање, што значи да ће направити комит спајања чак и ако је могуће спајање премотавањем унапред. Ово значи да ће без обзира на све, кадгод притиснете „Merge” дугме настати нови комит спајања. Као што видите у [Дугме Merge и упутство за ручно спајање захтева за повлачење](#), GitHub вам даје све ове информације ако кликнете на линк савета.

This pull request can be automatically merged.
You can also merge branches on the [command line](#).

Merging via command line
If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

HTTP Git Patch <https://github.com/schacon/fade.git>

Step 1: From your project repository, check out a new branch and test the changes.

```
git checkout -b schacon-patch-1 master  
git pull https://github.com/schacon/fade.git patch-1
```

Step 2: Merge the changes and update on GitHub.

```
git checkout master  
git merge --no-ff schacon-patch-1  
git push origin master
```

Слика 116. Дугме Merge и упутство за ручно спајање захтева за повлачење

Ако одлучите да не желите да га спојите, можете и да само затворите захтев за повлачење и особа која га је отворила ће бити обавештена.

Референце на захтев за повлачење

Ако баратате са **пуно** захтева за повлачење и не желите да додате гомилу удаљених репозиторијума или да сваки пут радите једнократно повлачење, GitHub вам омогућава да употребите један добар трик. Ово је делић напредног трика чије детаље ћемо обрадити у [Рефспек](#), али може да буде јако користан.

GitHub заправо оглашава гране које су захтев за повлачење репозиторијума као врсту псеудограна на серверу. Подразумевано их не добијате када клонирате, али оне су ту прикривене и можете да им приступите прилично једноставно.

Да бисмо ово показали, користићемо команду ниског нивоа (често се користи термин „цевоводна“ *plumbing* команда, о чему ћете више прочитати у [Водовод и порцелан](#)) која се зове [ls-remote](#). Ова команда се у општем случају не користи у свакодневним Гит операцијама али је корисна да нам покаже које референце постоје на серверу.

Ако извршимо ову команду над „blink“ репозиторијумом који смо раније користили, добићемо листу свих грана и ознака и осталих референци у репозиторијуму.

```
$ 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
```

Наравно, ако сте у свом репозиторијуму и извршићете `git ls-remote origin` или који год удаљени репозиторијум желите да проверите, показаће вам нешто слично овоме.

Ако се репозиторијум налази на GitHub сервису и имате отворене захтеве за повлачење, ове референце ћете добити са префиксом `refs/pull/`. Ово су у суштини гране, али пошто нису под `refs/heads/` обично их не добијате када клонирате или преузимате податке са сервера — процес преузимања их обично игнорише.

Постоје по две референце за сваки захтев за повлачење — она која се завршава са `/head` показује на потпуно исти комит који је и последњи комит у грани захтева за повлачење. Дакле, ако неко у нашем репозиторијуму отвори захтев за повлачење, његова грана се зове `bug-fix` и показује на комит `a5a775`, онда у **нашем** репозиторијуму нећемо имати грану `bug-fix` (пошто је она у његовој рапчи), али *имаћемо* `pull/<зп#>/head` која показује на `a5a775`. Ово значи да можемо дosta лако да повучемо сваку грану захтева за повлачење без потребе да додајемо гомилу удаљених репозиторијума.

Сада, можете да урадите нешто као директно преузимање референце.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch                  refs/pull/958/head -> FETCH_HEAD
```

Ово говори програму Гит, „Повежи се на `origin` удаљени репозиторијум и преузми референцу са именом `refs/pull/958/head`.“ Програм Гит вас са задовољством слуша и преузима све што вам треба да конструишете ту референцу и ставља показивач на комит који желите под `.git/FETCH_HEAD`. Након овога можете да извршићете `git merge FETCH_HEAD` у грани у којој желите да тестирате, али та порука комита спајања комита изгледа помало чудно. Такође, ако разматрате **много** захтева за повлачење, ово постаје заморно.

Постоји и начин да преузмете *све* захтеве за повлачење и да их одржите актуелним сваки пут када се повежете на удаљени репозиторијум. Отворите у свом омиљеном едитору `.git/config` и потражите `origin` удаљени репозиторијум. Требало би да изгледа некако овако:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Она линија која почиње са `fetch` = је такозвани рефспек. То је начин мапирања имена на удаљеном репозиторијуму у имена у вашем локалном `.git` директоријуму. Линија из примера говори програму Гит „ствари на удаљеном репозиторијуму које се налазе под `refs/heads` треба да иду у мој локални репозиторијум под ``refs/remotes/origin``. Можете изменити овај одељак тако да додате још један рефспек:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Последња линија каже програму Гит, „све референце који изгледају као `refs/pull/123/head` треба да се локално сместе као `refs/remotes/origin/pr/123`“. Сада, ако сачувате тај фајл и урадите `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
# ...
```

Сада се сви удаљени захтеви за повлачење локално представљају референцима које понашају слично као и пратеће гране; могуће их је само читати и ажурирају се када обавите преузимање. Овако је супер једноставно да испробате код из захтева за спајање 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'
```

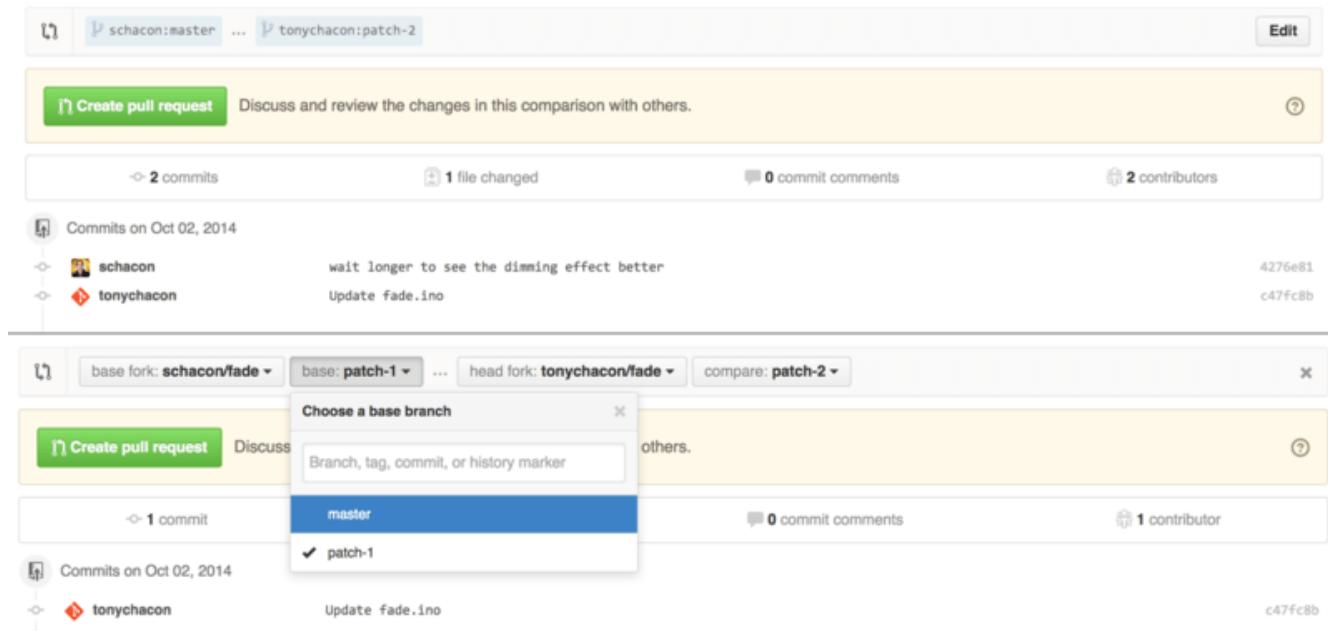
Они који имају око соколово приметиће да је `head` на kraju удаљеног дела рефспека. Постоји и `refs/pull/#/merge` референца на GitHub страни, што представља комит који би настао ако притиснете „*merge*“ дугме на сајту. Ово вам омогућава да тестирате спајање и пре него што уопште притиснете дугме.

Захтеви за повлачење на захтевима за повлачење

Не само да можете отварати захтеве за повлачење који као одредиште имају главну или `master` грану, већ можете и да отворите захтев за повлачење који као одредиште има било коју грану на мрежи. Заправо, одредиште може бити и други захтев за повлачење.

Ако видите захтев за повлачење који иде у добром правцу и имате идеју за промену која зависи од њега, или ниста сигурни да ли је то добра идеја, или једноставно немате приступ турању измена на одредишну грану, можете да отворите захтев за повлачење директно на њега.

Када отворите захтев за повлачење, појавиће се поље на врху стране које наводи ка којој грани желите да будете повучени и од које захтевате да повучете. Ако притиснете дугме „Edit” са десне стране тог поља, можете да промените не само гране већ и на којој рачви.



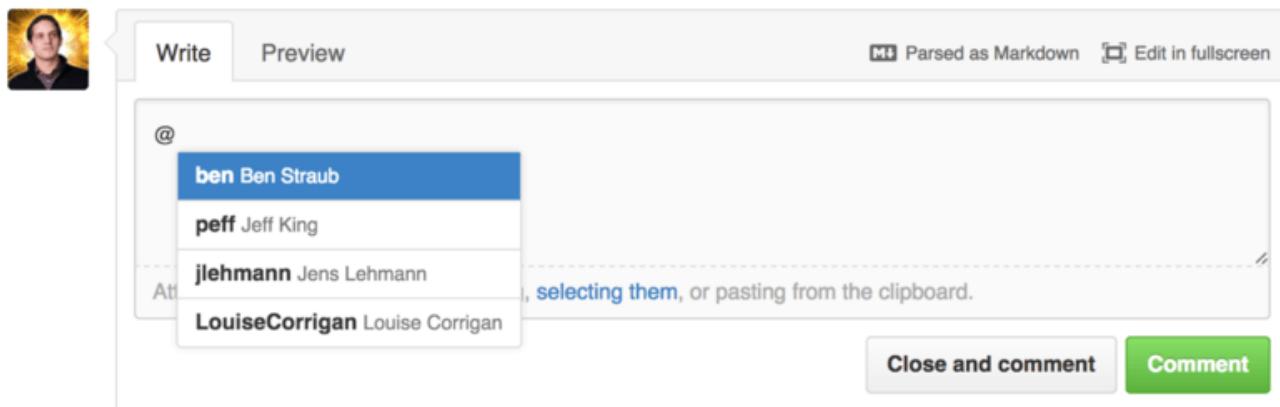
Слика 117. Ручна промена одредишне рачве и гране захтева за повлачење

Овде прилично лако можете навести да спајате своју нову грану у други захтев за повлачење или другу рачву пројекта.

Помињање и обавештења

GitHub такође има уграђен и прилично добар систем обавештења који је може бити од користи када имате питања или вам треба повратна информација од одређених особа или целих тимова.

У било ком коментару можете почети да куцате карактер @ и покренуће се ауто довршавање са именима и корисничким именима људи који су сарадници или дају допринос пројекту.



Слика 118. Помените да куцате @ да бисте поменули неког

Можете да поменете и корисника који није на тој падајућој листи, али ауто довршавач често то може да убрза.

Када објавите коментар у коме помињете корисника, он или она ће бити обавештени. То значи да ово може бити веома ефикасан начин позивања људи у разговоре, уместо да сами морају да питају. У GitHub захтевима за повлачење људи ће врло често позивати друге људе из свог тима или компаније да ураде рецензију захтева за повлачење или тикета.

Ако неко буде поменут на захтеву за повлачење или тикету, биће „претплаћени” на њега и наставиће да добијају обавештења сваки пут када се ту дододи нека активност. И ви ћете такође бити претплаћени на све што сами отворите, ако надгледате (*watch*) репозиторијум или ако коментаришете на нешто. Ако више не желите да добијате обавештења, на страници постоји дугме „*Unsubscribe*” на које можете кликнути и престаћете да добијате новости о њој.

Notifications



✖️ **Unsubscribe**

You're receiving notifications
because you commented.

Слика 119. Престанак праћења захтева за повлачење или тикета.

Страница са обавештењима

Када овде кажемо „обавештење” у контексту сервиса GitHub, мислимо на одређени начин којим сервис GitHub покушава да ступи у контакт са вама онда када се дододе неки догађаји. Постоји неколико различитих начина да конфигуришете обавештења. Ако одете на картицу „*Notification center*” странице за подешавања, видећете неке од опција које имате.

The screenshot shows the 'Notification center' section of the GitHub settings. On the left sidebar, under 'Notification center', the 'Watching' option is selected. The main content area is titled 'How you receive notifications'. It contains two sections: 'Participating' and 'Watching'. Under 'Participating', it says 'When you participate in a conversation or someone brings you in with an @mention.' with checkboxes for 'Email' and 'Web'. Under 'Watching', it says 'Updates to any repositories or threads you're watching.' with checkboxes for 'Email' and 'Web'. Below this is a 'Notification email' section with a 'Primary email address' input field containing 'tchacon@example.com' and a 'Save' button. At the bottom is a 'Custom routing' section with the note 'You can send notifications to different verified email addresses depending on the organization that owns the repository.'

Слика 120. Опције у центру за обавештавање

Постоје два избора: добијање обавештења путем имејла (*Email*) или путем веба (*Web*) и можете да изаберете једно од њих, ниједно, или оба у случају када активно учествујете у стварима и за активности на репозиторијумима које надгледате.

Веб обавештења

Веб обавештења постоје само сервису GitHub и можете да их погледате само на GitHub веб сајту. Ако вам је ова опција изабрана у подешавањима, па се окине обавештење за вам, видећете малу плаву тачку преко ваше иконице за обавештења у горњем делу екрана као што се види на [Центар за обавештења](#).

The screenshot shows the GitHub Notifications center. At the top, there are tabs for 'Notifications' (selected) and 'Watching'. A notification bar at the top right says 'You have unread notifications' with a 'Mark all as read' button. The main area is divided into sections by project: 'Unread' (4 notifications), 'Participating' (3 notifications), and 'All notifications'. The 'Unread' section lists 'mycorp/project1' with a notification from 'SF Corporate Housing Search' (posted an hour ago) and 'git/git-scm.com' with a notification from 'Front Page' (posted 3 hours ago). The 'Participating' section lists 'schacon/blink' with two notifications: 'To Be or Not To Be' (posted 5 days ago) and 'Three seconds is better' (posted 5 days ago).

Слика 121. Центар за обавештења

Ако кликнете на то, појавиће се листа свих ставки о којима сте обавештени, груписана по пројектима. Обавештења можете да филтрирате по одређеном пројекту тако што кликнете

на његово име у траци са леве стране. Можете и да потврдите обавештење кликом на иконицу штиклирања поред било ког обавештења, или да потврдите сва обавештења у пројекту тако што ћете кликнути на иконицу штиклирања на врху групе. Постоји и дугме за утишавање обавештења поред сваке иконице штиклирања на које можете да кликнете уколико више не желите обавештења у вези с том ставком.

Сви ови алати су веома корисни за руковање великим бројем обавештења. Многи напредни корисници сервиса GitHub ће једноставно искључити сва обавештења путем имејла и у потпуности управљати свим обавештењима преко овог екрана.

Имејл обавештења

Обавештења путем имејла су још један начин за руковање обавештењима на сервису GitHub. Ако вам је ова опција укључена, добијаћете имејл за свако обавештење. Видели смо примере овога у [Коментари послати као имејл обавештења](#) и [Имејл обавештење о новом захтеву за повлачење](#). Имејлови ће такође бити исправно смештени у тематске низове, што је лепо ако користите имејл клијент који подржава овај вид прегледа имејлова.

У заглавља имејлова које вам шаље сервис GitHub се утрађује и поприлична количина метаподатака, што може да буде веома корисно за постављање персонализованих филтера и правила.

На пример, ако погледамо имејл заглавља послата Тонију у имејлу који је приказан у [Имејл обавештење о новом захтеву за повлачење](#), видећемо следеће међу осталим информацијама:

```
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
```

Овде има неколико занимљивих ствари. Ако желите да истакнете или преусмерите имејлове за овај одређени пројекат или чак за захтев за повлачење, информација у [Message-ID](#) вам даје све податке у формату [`<корисник>/<пројекат>/<тип>/<ид>`](#). На пример, да је ово тикет, у пољу [`<тип>`](#) би уместо `pull` писало `issues`.

Поља [List-Pos](#) и [List-Unsubscribe](#) значе да ако имате клијент за мејл који разуме ово, можете лако да шаљете на листу или да укинете претплату са теме. То би у суштини било исто као да кликнете на дугме „*mute*” у веб верзији обавештења или на „*Unsubscribe*” на самој страници захтева за повлачење или тикета.

Такође вреди напоменути да ако су вам укључена и имејл и веб обавештења, па прочитате мејл верзију обавештења, веб верзија ће се такође означити као прочитана, ако су слике дозвољене у вашем имејл клијенту.

Посебни фајлови

Ако у вашем репозиторијуму постоји неколико посебних фајлова, сервис GitHub ће их приметити.

README

Први фајл је **README**, он може бити у скоро било ком формату који сервис GitHub препознаје као прозу. На пример, може да буде **README**, **README.md**, **README.asciidoc**, итд. Ако сервис GitHub у вашем извornом коду види **README** фајл, приказаће га на почетној страници пројекта.

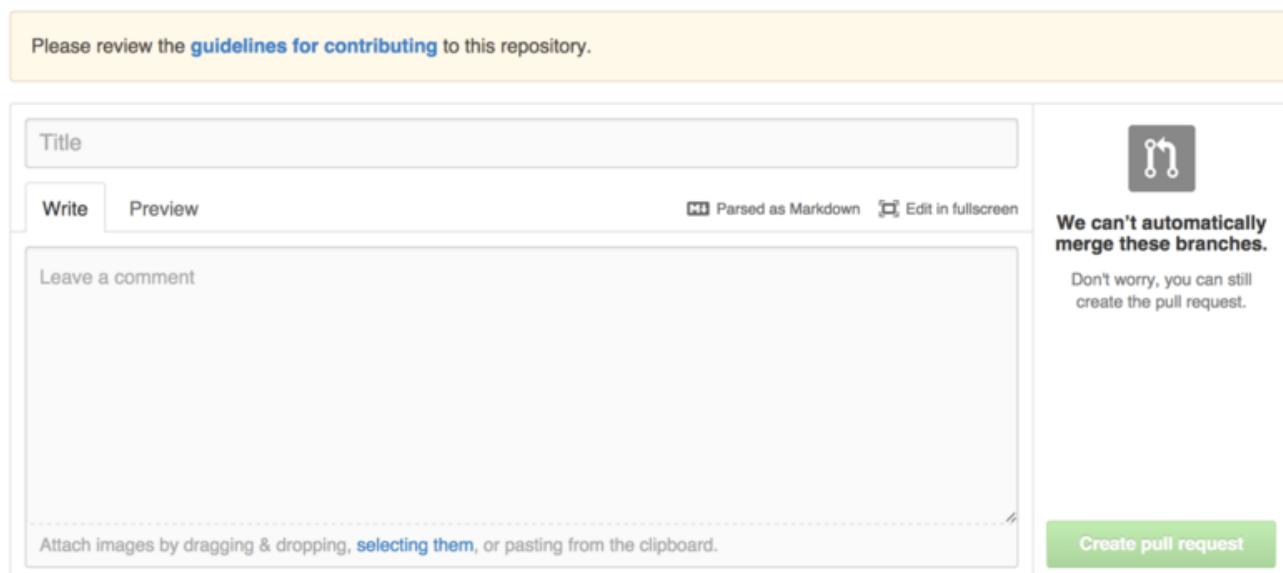
Многи тимови у овај фајл записују све релевантне информације о пројекту за неког ко је можда нов на репозиторијуму или пројекту. Обично се овде пишу следеће ствари:

- Шта је намена пројекта
- Како га конфигурисати и инсталацији
- Пример који показује како се користи или како да се покрене
- Лиценца под којом је пројекат доступан
- Како се даје допринос пројекту

Пошто ће сервис GitHub приказивати овај фајл, можете да уградите слике или линкове да би олакшали разумевање документа.

CONTRIBUTING

Други посебан фајл који сервис GitHub препознаје је фајл **CONTRIBUTING**. Ако имате фајл **CONTRIBUTING** са било којом екstenзијом, GitHub ће приказати **Отварање захтева за повлачење** када постоји фајл **CONTRIBUTING** кад год неко отвори захтев за повлачење.



Слика 122. Отварање захтева за повлачење када постоји фајл **CONTRIBUTING**

Идеја је да овде наведете одређене ствари које желите или не желите да буду део захтева за повлачење који се шаљу вашем пројекту. На овај начин људи ће заправо моћи да прочитају

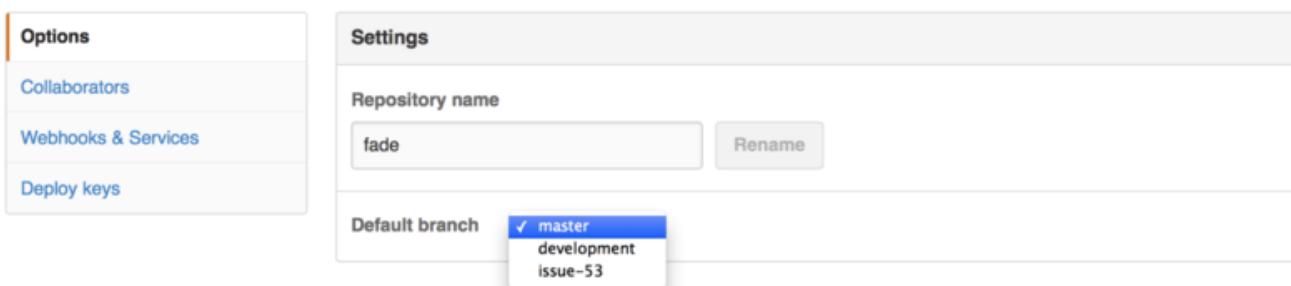
смернице и пре него што отворе захтев за повлачење.

Администрација проекта

У општем случају нема много административних ствари које можете да радите са само једним пројектом, али постоји неколико ставки које би могле бити интересантне.

Мењање подразумеване гране

Ако као подразумевану грану над којом желите да људи отварају захтеве за повлачење или да је подразумевано виде користите неку грану која није **master**, можете то да промените на страници подешавања репозиторијума у картици „*Options*”.

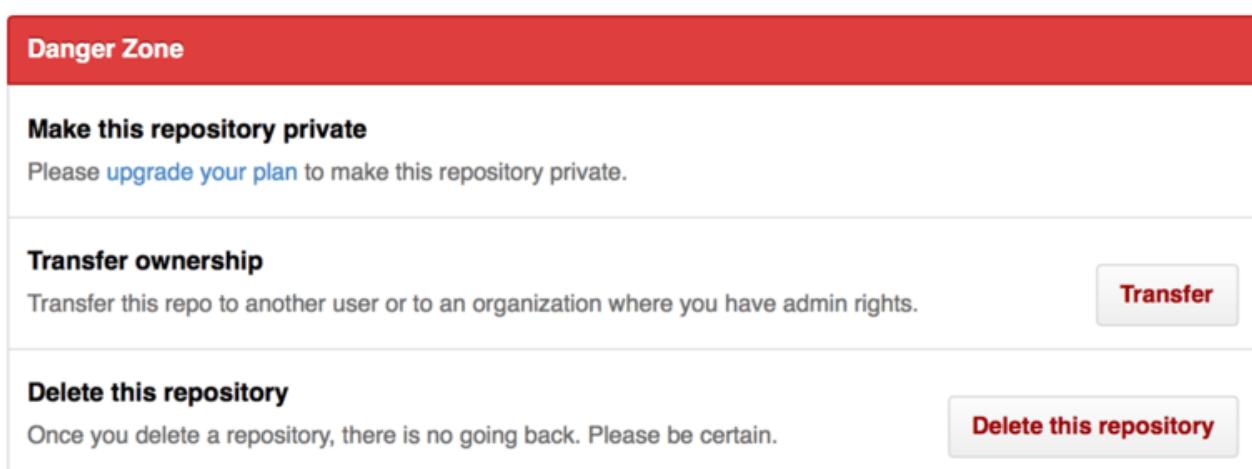


Слика 123. Промена подразумеване гране пројекта

Једноставно промените подразумевану грану у падајућој листи и то ће постати подразумевана грана за све главне операције од тада па надаље, укључујући и то која грана се подразумевано одјављује када неко клонира репозиторијум.

Пренос пројекта

Ако пожелите да свој пројекат пренесте другом кориснику или организацији на сервису GitHub, постоји опција „*Transfer ownership*” на дну исте „*Options*” картице на страници подешавања вашег репозиторијума која вам то омогућава.



Слика 124. Пренос пројекта другом кориснику сервиса GitHub или организацији

Ово је од помоћи ако напуштате пројекат и неко други жели да га преузме, или ако пројекат постаје све већи и желите да га преместите у организацију.

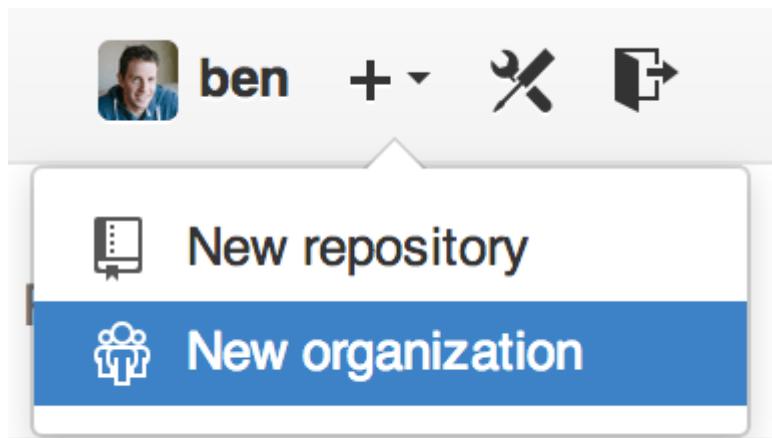
Ово не само да премешта репозиторијум заједно са свим пратиоцима и звездицама на друго место, већ подешава и преусмеравање са ваш УРЛ адресе на ново место. Преусмеравају се и клонирања и преузимања из програма Гит, не само веб захтеви.

Управљање организацијом

Поред налога за једног корисника, сервис GitHub има и нешто што се зове организације. Као и лични налози, организациони налози имају простор имена у ком постоје сви пројекти те организације, али многе друге ствари се разликују. Ови налози представљају групу људи са заједничким власништвом над пројектима и постоји много алата за управљање подгрупама тих људи. Обично се ови налози користе за групе отвореног кода (као што су „perl“ или „rails“) или компаније (као што су „google“ или „twitter“).

Основе о организацијама

Организација се прави веома једноставно; само кликните на иконицу у горњем десном углу било које странице сервиса GitHub и изаберите из менија „New organisation“.



Слика 125. „New organization“ ставка менија

Прво треба да дате име својој организацији и да оставите имејл адресу као главну тачку контакта са групом. Ако желите, онда можете позвати остале кориснике да постану сувласници налога.

Пратите наредне кораке и ускоро ћете постати власник потпуно нове организације. Као и лични налози, организације су бесплатне ако се слажете да све што планирате да чувате тамо буде отвореног кода.

Као власник организације, када рачвате репозиторијум, имајете избор да га рачвате у простор имена ваше организације. Када креирате нове репозиторијуме, можете да их креирате или под личним налогом или под било којом организацијом у којој сте један од власника. Можете и аутоматски да „надгледате“ било који нови репозиторијум који се направи под тим организацијама.

Баш као у [Лични аватар](#), можете поставити аватар за вашу организацију и тако је донекле персонализовали. Такође, баш као код личних налога, имате почетну страницу за организацију на којој се налази списак свих ваших репозиторијума и коју други људи могу

да виде.

Сада ћемо показати неке ствари које су мало другачије код организационих налога.

Тимови

Организације су повезане са појединцима преко тимова, који једноставно представљају групе појединачних корисничких налога и репозиторијума у оквиру организације заједно са врстом приступа који ти људи имају у тим репозиторијумима.

На пример, рецимо да ваша компанија има три репозиторијума: `frontend`, `backend` и `deployscripts`. Желећете да ваши *HTML/CSS/JavaScript* програмери имају приступ у `frontend` и можда у `backend`, а да људи из тима *Operations* имају приступ у `backend` и у `deployscripts`. Тимови ово чине једноставним, без потребе да управљате сарадницима за сваки појединачни репозиторијум.

Страница организације вам показује једноставну командну таблу са свим репозиторијумима, корисницима и тимовима који су под овом организацијом.

The screenshot shows the GitHub organization page for 'chaconcorp'. At the top, there's a search bar and a '+ New repository' button. Below it, three repositories are listed: 'deployscripts' (scripts for deployment), 'backend' (Backend Code), and 'frontend' (Frontend Code). To the right, there are two panels: 'People' (listing three members: dragonchacon, schacon, and tonychacon) and 'Teams' (listing three teams: Owners, Frontend Developers, and Ops).

| Team | Members | Repositories |
|---------------------|-----------|----------------|
| Owners | 1 member | 3 repositories |
| Frontend Developers | 2 members | 2 repositories |
| Ops | 3 members | 1 repository |

Слика 126. Страница организације

Да бисте управљали тимовима, можете да кликнете на „*Teams*” траку са десне стране у Страница организације.

Ово ће вас одвести на страницу на којој можете да додајете чланове у тим, да додајете репозиторијуме у тим или да управљате подешавањима и нивоима контроле приступа за

тим. Сваки тим може да има дозволу да само за читање, за читање и упис, или административни приступ репозиторијумима. Нивое можете да промените кликом на дугме -„*Settings*” у [Teams страница](#).

The screenshot shows the GitHub Teams interface for the 'Frontend Developers' team. At the top, there's a header with the organization logo 'chaconcorp', navigation links for 'People 3', 'Teams 3', and 'Audit log', and a search bar. Below the header, the team card for 'Frontend Developers' is displayed, showing it has no description, 2 members, and 2 repositories. It includes 'Leave' and 'Settings' buttons. The 'Members' tab is selected, showing two members: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon). Each member entry includes their profile picture, name, role ('tonychacon' is Admin), and a 'Remove' button. A button to 'Invite or add users to team' is also present. A note at the bottom of the team card states: 'This team grants **Admin** access: members can read from, push to, and add collaborators to the team's repositories.'

Слика 127. Teams страница

Када некога позовете у тим, он или она ће добити имејл у који их обавештава да су позвани.

Штавише, тимска [@помињања](#) (као што је [@astmecogr/frontend](#)) раде скоро исто као и са индивидуалним корисницима, осим што су онда **сви** чланови тима претплаћени на тему. Ово је корисно ако желите пажњу неког из тима, али нисте сигурни кога тачно треба да питате.

Корисник може да припада било ком броју тимова, зато немојте ограничавати себе само на тимове за контролисање приступа. Тимови одређених интересовања као што су [ux](#), [css](#) или [рефакторисање](#) су корисни за одређену врсту питања, а други као [legal](#) или [далтонисити](#) за потпуну другу врсту.

Ревизиони лог

Организације дају власницима и приступ свим информацијама о томе шта се догађа у организацији. Можете да одете на картицу *Audit Log* и да видите догађаје су се додали на нивоу организације, ко их је починио и где на свету су се одиграли.



| Recent events | | Filters ▾ | Search... |
|---|---|---|-----------------------|
|  | dragonchacon |  Yesterday's activity | member 32 minutes ago |
| | added themselves to the chaconcorp/ops team |  Organization membership | |
|  | schacon |  Team management | member 33 minutes ago |
| | added themselves to the chaconcorp/ops team |  Repository management | |
|  | tonychacon |  Billing updates | member 16 hours ago |
| | invited dragonchacon to the chaconcorp organization |  Hook activity | |
|  | tonychacon | France  | 16 hours ago |
| | invited schacon to the chaconcorp organization | | |
|  | tonychacon | France  | 16 hours ago |
| | gave chaconcorp/ops access to chaconcorp/backend | | |
|  | tonychacon | France  | 16 hours ago |
| | gave chaconcorp/frontend-developers access to chaconcorp/backend | | |
|  | tonychacon | France  | 16 hours ago |
| | gave chaconcorp/frontend-developers access to chaconcorp/frontend | | |
|  | tonychacon | France  | 16 hours ago |
| | created the repository chaconcorp/deployscripts | | |
|  | tonychacon | France  | 16 hours ago |
| | created the repository chaconcorp/backend | | |

Слика 128. Audit log

Можете и да филтрирате само одређене врсте догађаја, одређена места или одређене људе.

Писање скрипти за GitHub

До сада смо прешли све главније особине и процесе рада на сервису GitHub, али свака већа група или пројекат ће пожелети да се сервис GitHub на одређени начин прилагоди њиховим потребама, или ће пожелети да интегришу неке спољне сервисе.

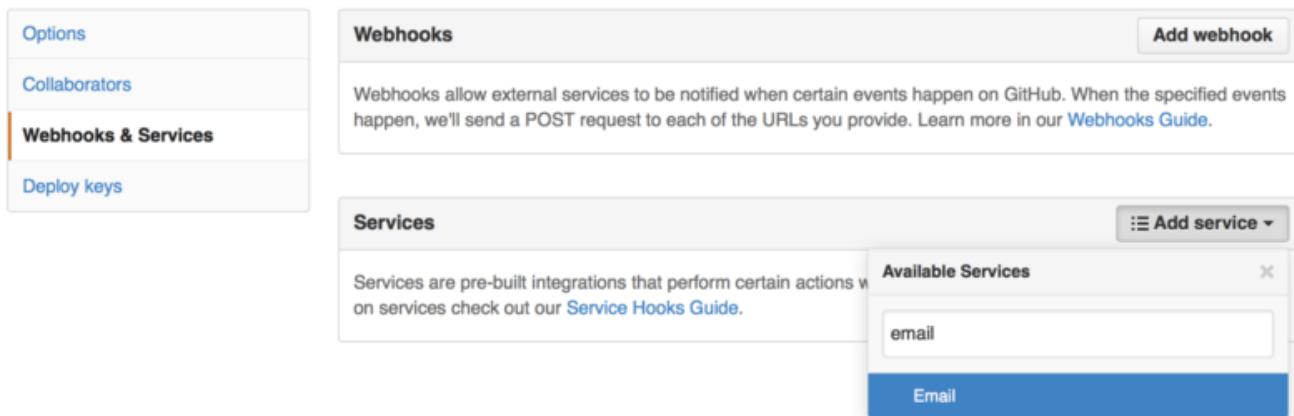
Срећом по нас, сервис GitHub је лако хаковати на многе начине. У овом одељку ћемо показати начине коришћења система GitHub кука и његовог API тако да натерамо сервис GitHub да ради онако како нама одговара.

Сервиси и Куке

Одељак административног дела GitHub репозиторијума „*Hooks and Services*” је најлакши начин за подешавање сервиса Github тако да врши интеракцију са спољним системима.

Сервиси

Прво ћемо погледати сервисе. Интеграције за куке и за сервисе се могу наћи у *Settings* одељку вашег репозиторијума, где смо раније ишли да додамо сараднике и мењамо подразумевану грану пројекта. Под картицом „*Webhooks and Services*” ћете видети нешто као [Конфигурациони одељак за сервисе и куке](#).



Слика 129. Конфигурациони одељак за сервисе и куке

Ту се налази на десетине сервиса које можете одабрати. То су углавном интеграције у друге комерцијалне системе, или системе отвореног кода. Већина њих служи за сервисе континуалне интеграције, а ту су и трекери за проблеме и багове, системи за ћаскање и системи за документацију. Проћи ћемо кроз подешавање једног врло једноставног, куке за имејл. Ако изпадајућег менија „*Add Service*” изаберете „*email*”, добићете конфигурациони екран као са слике [Конфигурација имејл сервиса](#).

The screenshot shows the GitHub 'Webhooks & Services' configuration interface. On the left, a sidebar lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main area is titled 'Services / Add Email' and contains the following sections:

- Install Notes**:
 - address whitespace separated email addresses (at most two)
 - secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
 - send_from_author uses the commit author email address in the From address of the email.
- Address**: A text input field containing 'tchacon@example.com'.
- Secret**: An empty text input field.
- Send from author**: A checkbox that is unchecked.
- Active**: A checked checkbox with the subtext 'We will run this service when an event is triggered.'
- Add service**: A green button at the bottom of the form.

Слика 130. Конфигурација имејл сервиса

У овом случају, ако притиснемо дугме „Add service”, на имејл адресу коју смо навели ћемо добити имејл сваки пут када неко гурне промене на репозиторијум. Сервиси могу да слушају пуно различитих врста догађаја, али већина слуша само догађаје гурања и онда раде нешто с тим подацима.

Ако постоји систем који користите и желите да га интегришете са сервисом GitHub, треба да проверите овде и погледате да ли већ постоји доступна интеграција за тај сервис. На пример, ако користите *Jenkins* да извршавате тестове над својом кодом, можете да укључите уграђену интеграцију за сервис *Jenkins* и да тако окинете тестирање сваки пут када неко гурне промене на репозиторијум.

Куке

Ако вам треба нешто одређеније или желите да се интегришете са сервисом или сајтом који се не налази на овом списку, можете да пробате општији систем кука. Куке за GitHub репозиторијум су прилично једноставне. Можете да наведете URL и GitHub ће слати HTTP товар ка тој URL адреси када се деси било који догађај који желите.

У општем случају, ово функционише тако да можете подесити мали веб сервис који слуша товар GitHub куке и онда ради нешто са подацима када их прими.

Да бисте омогућили куку, кликните на дугме „Add webhook” у [Конфигурациони одељак за сервисе и куке](#). То ће вас одвести на страницу која изгледа као [Конфигурација веб куке](#).

The screenshot shows the 'Webhooks & Services' section of the GitHub settings. On the left sidebar, 'Webhooks & Services' is selected. The main area is titled 'Webhooks / Add webhook'. It explains that a POST request will be sent to the specified URL with event details. It also links to developer documentation. The 'Payload URL' field contains 'https://example.com/postreceive'. The 'Content type' dropdown is set to 'application/json'. There is a 'Secret' input field which is empty. Below these, there's a section for selecting events: 'Just the push event.' (selected), 'Send me everything.', and 'Let me select individual events.'. A checkbox for 'Active' is checked, with a note that event details will be delivered when triggered. At the bottom is a green 'Add webhook' button.

Слика 131. Конфигурација веб куке

Конфигурација за веб куку је прилично једноставна. У већини случајева једноставно унесете УРЛ и тајни кључ, па онда кликнете на „*Add webhook*“. Постоји неколико опција за које догађаје желите да вам сервис GitHub шаље товар — подразумевано се товар прима само за **push** догађај, тј. када неко гурне нови код на било коју грану вашег репозиторијума.

Хајде да видимо мали пример веб сервиса који бисте могли да подесите за обраду веб куке. Користићемо Рубијев веб радни оквир Синатра пошто је дosta концизан и требало би да лако можете видети шта радимо.

Речимо да желите да добијете имејл када одређена особа гурне промене на одређену грану вашег пројекта која мења одређени фајл. То прилично лако да урадимо следећим кодом:

```

require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parsiraj JSON

  # prikupi podatke koje tražimo
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # uzmi listu svih fajlova koji su menjani
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # proveri naš uslov
  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   'Скот је променио фајл'
      body     "АЛАРМ"
    end
  end
end

```

Овде од сервиса GitHub добијамо товар и JSON формату и испитујемо ко је турнуо промене, на коју грану и који су фајлови су промењени сваким комит који је турнут. Затим проверимо да ли то задовољава наше услове и затим пошаљемо имејл ако задовољава.

Да бисте испрограмирали и тестирали нешто овако, можете да користите лепу конзолу за развој на истом екрану у коме сте подесили куку. Можете видети последњих неколико испорука које је сервис GitHub пробао да направи за ту веб куку. За сваку куку можете пронаћи податак када је достављена, да ли је достављање било успешно, као и тело и заглавља за захтев и за одговор. Због свега овога је невероватно лако да тестирате и отклањате грешке у својим кукама.

The screenshot shows the GitHub Webhooks interface. At the top, there's a section titled "Recent Deliveries" with three entries:

- An error message (red exclamation mark) for delivery ID 4aeae280-4e38-11e4-9bac-c130e992644b from 2014-10-07 17:40:41.
- A successful message (green checkmark) for delivery ID aff20880-4e37-11e4-9089-35319435e08b from 2014-10-07 17:36:21.
- A successful message (green checkmark) for delivery ID 90f37680-4e37-11e4-9508-227d13b2ccfc from 2014-10-07 17:35:29.

Below this, there are tabs for "Request" (selected), "Response" (status 200 green), and "Redeliver".

The "Headers" section shows the request details:

```
Request URL: https://hooks.example.com/payload
Request method: POST
Content-Type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push
```

The "Payload" section shows the JSON response body:

```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bfffaf827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bfffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
    }
  ]
}
```

Слика 132. Информације за отклањање грешака у веб куки

Још једна одлична могућност овога је то што можете поново да испоручите било који товар у циљу лакшег тестирања свог сервиса.

За више информација о томе како да пишете веб куке и о свим различитим догађајима које можете да ослушкујете, посетите *GitHub Developer* документацију која се налази на <https://developer.github.com/webhooks/>.

GitHub API

Сервиси и куке представљају начин да добијате обавештења у вези с догађајима који се догоде на вашим репозиторијумима, али шта ако вам треба више информација о овим догађајима? Шта ако треба да аутоматизујете нешто као додавање сарадника или додавање

ознака тикетима?

Овде се као користан показује GitHub API. GitHub има гомилу API крајњих тачака које вам омогућавају да радите скоро све што можете на веб сајту, али на аутоматизовани начин. У овом одељку ћемо научити како да се аутентификујете и повежете на API, како да оставите коментар на тикет и како да промените статус захтева за повлачење кроз API.

Основе коришћења

Најосновнија ствар коју можете да урадите је обичан GET захтев на крајњој тачки која не захтева аутентификацију. Ово може да буде корисничка информација или информација која може само да се прочита у вези с пројектом отвореног кода. На пример, ако желимо да сазнамо нешто више о кориснику који се зове „*schacon*”, извршићемо нешто овако:

```
$ 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"
}
```

Постоји гомила крајњих тачака као што је ова са којих можете да добијете информације о организацијама, пројектима, тикетима, комитовима — о скоро свему што јавно можете да видите на сервису GitHub. Можете чак и да користите API да прикажете произвољни Маркдаун или да нађете шаблон за [.gitignore](#).

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Коментарисање на тикету

Ипак, ако желите да нешто урадите на веб сајту, на пример да коментаришете на тикет или захтев за повлачење, или желите да погледате или да имате интеракцију са приватним садржајем, морате да се аутентификујете.

Постоји неколико начина за аутентификацију. Можете да користите основну аутентификацију само са својим корисничким именом и лозинком, али у општем случају је боља идеја да користите лични приступни токен. Можете да га генеришете из картице „*Applications*” на вашој страници за подешавања.

The screenshot shows the GitHub user profile for 'tonychacon'. On the left, there's a sidebar with links like Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications (which is selected), Repositories, and Organizations. The main area has a header 'Developer applications' with a 'Register new application' button. Below it, a message asks if you want to develop an application using the GitHub API. The 'Personal access tokens' section contains a 'Generate new token' button and a note about their use. The 'Authorized applications' section says there are none. The 'GitHub applications' section lists 'GitHub Team' with a 'Last used on Oct 6, 2014' timestamp and a 'Revoke' button.

Слика 133. Генеришите свој приступни токен са картице „Applications” на вашој страници за подешавања

Питаће вас да се одлучите за која подручја ће важити овај токен и за опис. Обавезно унесите добар опис како би вам било лакше да уклоните токен из скрипте или апликације када престанете да га користите.

GitHub ће вам показати токен само једном, зато га обавезно копирајте. Ово сада можете користити да потврдите свој идентитет у скрипти уместо да користите корисничко име и лозинку. Добро је што можете ограничiti подручје над којим желите да скрипта делује, а токен се може опозвати.

Поред тога, ово вам повећава границу учесталости захтева. Без аутентификације, ограничени сте на 60 захтева по сату. Са аутентификацијом, можете да направите до 5000 захтева по сату.

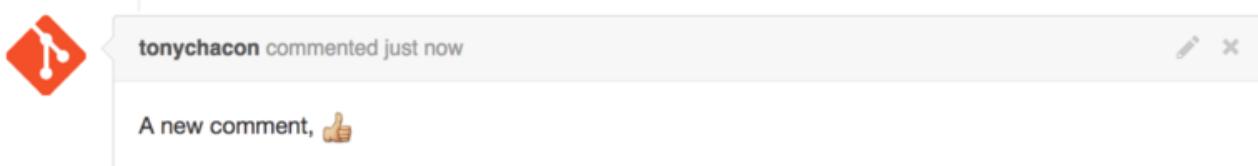
Хајде да искористимо токен тако да направимо коментар на једном од наших тикета. Рецимо да хоћемо оставити коментар на одређеном тикету, *Issue #6*. Да бисмо то урадили, морамо да пошаљемо HTTP POST захтев на [repos/<корисник>/<репозиторијум>/issues/<бр>/comments](#) користећи у заглављу „Authorization” токен који смо управо генерисали.

```

$ 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:"
}

```

Ако сада одете на тај тикет, видећете коментар који смо управо успешно послали као на [Коментар постављен употребом GitHub API](#).



Слика 134. Коментар постављен употребом GitHub API

API можете употребити да урадите скоро све што можете да урадите на сајту — да направите и подешавате прекретнице, да додељујете људе тикетима и захтевима за повлачење, да правите и мењате ознаке, да приступате подацима о комиту, да правите нове комитове и гране, да отварате, затварате и спајате захтеве за повлачење, да правите и мењате тимове, да коментаришете линије кода у захтеву за повлачење, да претражујете сајт и тако даље.

Промена статуса захтева за повлачење

Још један пример који ћемо погледати је рад са захтевима за повлачење, пошто то уме да буде јако корисно. Сваки комит може имати један или више статуса који су му придружени и постоји API који служи да додате статус и да вршите упит статуса.

Већина сервиса за континуалну интеграцију и тестирање користе овај API да реагују на гурање тако што тестирају код који је гурнут, па онда извештавају да ли је тај комит прошао све тестове. Ово можете користити и да проверите да ли је комит порука форматирана како ваља, да ли је подносилац захтева пратио све смернице за допринос, да ли је комит ваљано потписан — много тога.

Рецимо да на свом репозиторијуму подесите веб куку која гађа мали веб сервис који у комит поруци проверава стринг **Signed-off-by**.

```

require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parsiraj JSON
  repo_name = push['repository']['full_name']

  # pogledaj svaku komit poruku
  push["commits"].each do |commit|

    # potraži string "Signed-off-by"
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # pošalji status na 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

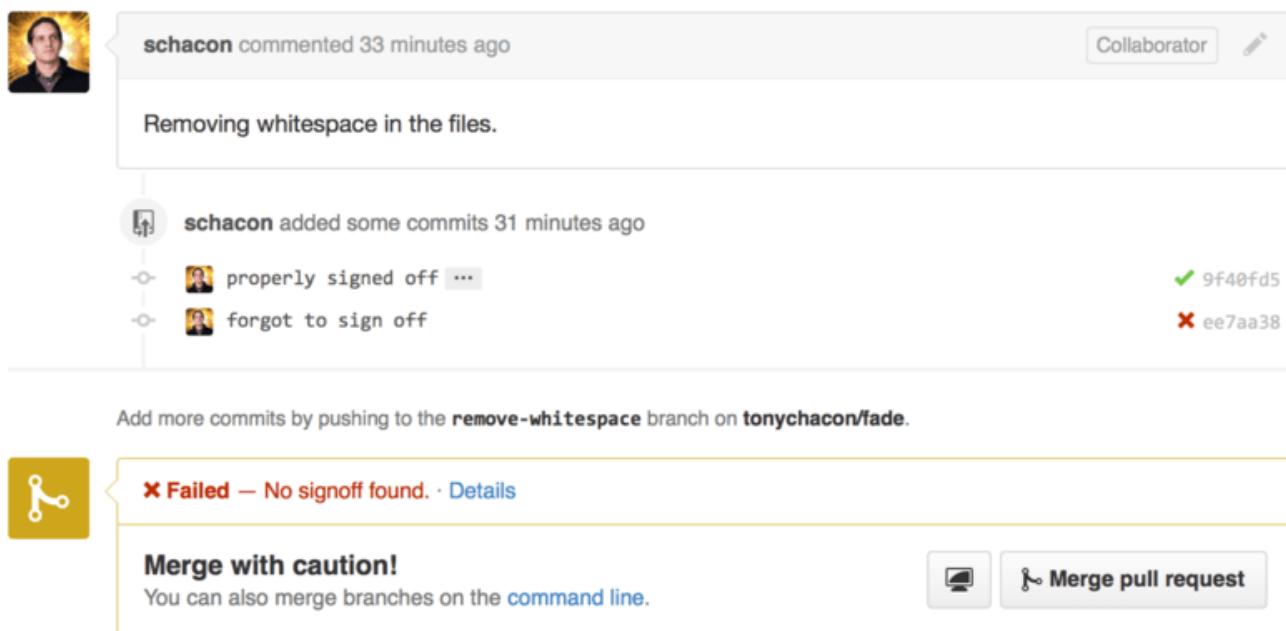
```

Надамо се да је ово лако испратити. У овом обрађивачу веб куке, испитује се сваки турнути комит, у комит поруци се тражи стринг **Signed-off-by** и коначно се ради POST преко HTTP протокола на [/repos/<корисник>/<репозиторијум>/statuses/<SHA-комита>](#) API крајњу тачку са статусом.

У овом случају можете да пошаљете стање ('success' тј. успех, 'failure' тј. неуспех, 'error' тј. грешка), опис онога што се десило, одредишни URL на који корисник може да оде за више информација и „context” у случају да има више статуса за један комит. На пример, сервис за тестирање може да јави статус, а и сервис за валидацију као што је овај може такође да јави

статус — разлика између њих се прави путем „context” поља.

Ако неко отвори нови захтев за повлачење на сервису GitHub и овај кука се окине, могли бисте видети нешто као [Статус комита преко API](#).



Слика 135. Статус комита преко API

Сада можете да видите мали зелени знак за штиклирање поред комита који у поруци има стринг `Signed-off-by` и црвени қрстић кроз онај где је аутор заборавио да се потпише. Можете и да видите да захтев за повлачење узима статус последњег комита са гране и упозорава вас ако је дошло до неуспеха. То је веома корисно ако користите овај API за резултате теста тако да случајно не спојите нешто чији последњи комит не пролази тестове.

Октокит

Иако смо скоро све радили кроз `curl` и једноставне HTTP захтеве у овим примерима, постоји неколико библиотека отвореног кода које овај API обезбеђују на природнији начин. У тренутку писања, подржани језици су Go, Објектни C, Руби и .NET. За више информација о овоме, погледајте <http://github.com/octokit>, јер они уместо вас рукују већином HTTP размене.

Надамо се да ће вам ови алати помоћи да прилагодите сервис GitHub својим потребама у складу са процесом рада који користите. За комплетну документацију и потпуни API, као и упутства за неке честе задатке, погледајте <https://developer.github.com>.

Резиме

Сада сте *GitHub* корисник. Знате како да направите налог, руководите организацијом, направите репозиторијум и турате промене на њега, дајете допринос пројектима других људи и прихватате доприносе других. У следећем поглављу ћете научити неке напредније алате и савете за решавање комплексних ситуација, који ће од вас заиста направити Гит мајстора.

Гит алати

До сада сте научили већину команди које се свакодневно користе, процесе рада који су вам потребни за управљање или одржавање Гит репозиторијума за контролу вашег извornог кода. Успешно сте завршили основне задатке праћења и комитовања фајлова и укротили сте снагу стејџа и једноставног тематског гранања и спајања.

Сада ћете истражити више веома моћних ствари које програм Гит може да уради, а које вероватно нећете свакодневно користити. Али у неком тренутку ће вам бити потребне.

Избор ревизија

Програм Гит вам омогућава да на неколико начина наведете одређене комитове или опсег комитова. Они нису обавезно очигледни, али је добро да се познају.

Просте ревизије

Један комит очигледно можете да наведете његовом пуном SHA-1 контролном сумом дужине 40 карактера, али такође постоје и начини навођења комитова који су лакши за људе. Овај одељак представља различите начине на које можете навести један комит.

Кратак SHA-1

Програм Гит је доволно паметан да одреди на који комит мислите када откуцате неколико првих карактера, све док је делимичан SHA-1 дужине барем четири карактера и није двосмислен – то јест, само један објекат у текућем репозиторијуму почиње тим делимичним SHA-1.

На пример, ако желите да погледате један комит, претпоставимо да извршите команду `git log` и пронађете комит у којем сте додали одређену функционалност:

```
$ 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
```

У овом случају, рецимо да се заинтересовани за комит чији хеш почиње са `1c002dd...`. Тада можете да испитате било којом од следећих варијација комаде `git show` (под претпоставком да су краће верзије недвосмислене):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Програм Гит може да одреди кратку, јединствену скраћеницу ваших SHA-1 вредности. Ако команди `git log` проследите `--abbrev-commit`, излаз ће приказати краће вредности, али ће остати јединствене; подразумевано користи седам карактера, али ако је потребно да се одржи недвосмисленост SHA-1 биће дужи:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial commit
```

У општем случају, осам до десет карактера је више негоово да буде јединствено у оквиру пројекта. На пример, у фебруару 2019, Линукс кернел (који је прилично стабилан пројекат) у својој бази података објекта име преко 875.000 комитова и скоро седам милиона објекта, а не постоје објекти чије су SHA-1 контролне суме идентичне у првих 12 карактера.

KRATKA NAPOMENA U BEZI SHA-1

Доста људи је у неком тренутку постало забринуто да ће, неким случајем, имати у свом репозиторијуму два различита објекта који имају исту SHA-1 контролну суму. Шта онда?

Ако се деси да комитујете објекат чија је SHA-1 вредност контролне суме иста као неког ранијег *различитог* објекта у вашем репозиторијуму, програм Гит ће видети претходни објекат који се већ налази у бази података, претпоставиће да је већ уписан и једноставно ће да поново искористити. Ако у неком тренутку покушате да одјавите так објекат, увек ћете добити податке из првог објекта.

Међутим, требало би да сте свесни колико је невероватно мала вероватноћа оваквог сценарија. The SHA-1 хеш је дужине 20 бајтова или 160 битова. Број насумично хешираних објеката који су потребни да се обезбеди 50% вероватноће једне једине колизије је око 2^{80} (формулa за одређивање вероватноће колизије је $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} је $1,2 \times 10^{24}$ или 1 милион милијарди милијади. То је 1.200 пута веће од укупног броја свих зрнаца песка на планети Земљи.



Ево примера који вам даје идеју шта би требало да се догоди па да дође до SHA-1 колизије. Ако би свих 6,5 милијарди људи на Земљи програмирали и сваке секунде, свако од њих генерише код еквивалентан комплетној историји Линукс кернела (3,6 милиона Гит објеката) и туре га на један огроман Гит репозиторијум, требало би да прође отприлике 2 године док тај репозиторијум буде садржао довољно објеката да има 50% вероватноће за једну једину SHA-1 колизију објеката. Дакле, природна SHA-1 колизија је мање вероватна од тога да у истој ноћи, у невезаним инцидентима сваког од чланова вашег програмерског тима нападну и убију вукови.

Ако томе посветите рачунарску снагу вредну неколико хиљада долара, могуће је да се синтетишу два фајла са истом контролном сумом, као што је у фебруару 2017. године доказано на <https://shattered.io/>. Програм Гит прелази на SHA256 као подразумевани алгоритам хеширања, јер је он много отпорнији на нападе колизијом, и садржи утрађен код којим се овај напад ублажава (мада није могуће да се потпуно елиминише).

Референце грана

Један једноставан начин да се наведе одређени комит може да се употреби ако је то комит на врху неке гране; тада у било којој Гит команди која очекује референцу на комит просто можете да наведете име гране. Не пример, ако желите да испитате последњи комит објекат на грани, следеће две команде су еквивалентне, под претпоставком да грана `topic1` показује на `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949  
$ git show topic1
```

Ако желите да видите на који тачно SHA-1 грана показује, или ако желите да видите на шта се своди било који од ових примера у смислу SHA-1 вредности, можете употребити Гит водоводни алат под називом `rev-parse`. За више информација у вези водоводних алата, можете погледати [Гит изнутра](#); у основи, `rev-parse` служи за операције ниског нивоа и није дизајнирана за свакодневне операције. Међутим, понекада може бити од помоћи када желите да видите шта се заиста догађа. Овде над својом граном можете да извршите `rev-parse`.

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```

RefLog кратка имена

Једна од ствари које програм Гит обавља у позадини док ви радите је да чува *reflog* – лог у којем чува где су се у последњих неколико месеци налазили ваш HEAD и референце грана.

Свој *reflog* можете погледати командом `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
```

Сваки пут када се из неког разлога ажурира врх ваше гране, програм Гит ту информацију чува у овој привременој историји. *Reflog* податке такође можете искористити и да наведете старије комитове. На пример, ако желите да видите пету претходну вредност показивача HEAD вашег репозиторијума, можете да искористите `@{5}` референцу коју видите у *reflog* излазу:

```
$ git show HEAD@{5}
```

Ову синтаксу можете и да употребите ако желите да видите где је пре неког одређеног времена била грана. Рецимо, да видите где је јуче била ваша `master` грана, откуцајте:

```
$ git show master@{yesterday}
```

То би вам приказало где је јуче био врх `master` гране. Ова техника функционише само за податке који се још увек налазе у вашем *reflog* дневнику, тако да је не можете употребити за комитове старије од неколико месеци.

Ако желите да се *reflog* информације форматирају као `git log` излаз, можете да извршите `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 је приметити да су *reflog* информације стриктно локалне – то је дневник онога што сте *ви* урадили у *свом* репозиторијуму. Референце неће бити исте у нечијој копији репозиторијума; а такође и непосредно након што иницијално клонирате репозиторијум, *reflog* ће бити празан јер се још увек није додела никаква активност у репозиторијуму. Извршавање `git show HEAD@{2.months.ago}` ће вам приказати одговарајући комит само ако сте клонирали пројекат барем пре два месеца – ако сте га клонирали раније од тога, видећете само свој први локални комит.



Посматрајте reflog као Гитову верзију историје љуске

Ако познајете ЈУНИКС или Линукс, *reflog* можете сматрати за Гитову верзију историје љуске, што јасно наглашава чињеницу да је оно што се тамо налази важно само за вас и за вашу „сесију” и не тиче се никог другог ко можда ради на истој машини.



Означавање заграда у PowerShell

Када се користи PowerShell, заграде као што су `{` и `}` представљају специјалне карактере и морају да се означе. Можете их означити краткоузлазним акцентом ``` или да их у комит референцима поставите унутар знакова навода:

```
$ git show HEAD@{0}      # НЕЋЕ радити
$ git show HEAD@`{0}`     # OK
$ git show "HEAD@{0}"     # OK
```

Референце на предаке

Још један главни начин за навођење комитова је путем његових предака. Ако на крај

референце поставите `^`, програм Гит то разрешава у значење родитеља наведеног комита. Рецимо да погледате у историју свог пројекта:

```
$ 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
```

Претходни комит можете видети тако што наведете `HEAD^`, што значи „родитељ од 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'
```

Означавање циркумфлекса на Виндоуз систему

У `cmd.exe` на Виндоуз систему, `^` је специјални карактер и мора да се третира другачије. Можете или да га удвојите, или да референцу на комит ставите унутар знакова навода:



```
$ git show HEAD^      # НЕЋЕ радити на Виндоуз систему
$ git show HEAD^^    # OK
$ git show "HEAD^"   # OK
```

Такође можете да наведете и број након `^` – на пример, `d921970^2` значи „други родитељ од `d921970`”. Ова синтакса је корисна само код комитова спајања који имају више од једног родитеља. Први родитељ је грана на којој сте били када сте покренули спајање, а други је комит на грани у коју се спајате:

```
$ 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

Још једна главна спецификација родитељства је `~`. Ово такође показује на првог родитеља, тако да су `HEAD~` и `HEAD^` исто. Разлика се јавља када наведете број. `HEAD~2` значи „први родитељ првог родитеља”, или „деда” – она пролази по првим родитељима онолико пута колико наведете. На пример, у историји наведеној раније, `HEAD~3` би било:

```
$ git show HEAD~3  
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d  
Author: Tom Preston-Werner <tom@mojombo.com>  
Date: Fri Nov 7 13:47:59 2008 -0500
```

Ignore *.gem

Ово такође може да се наведе и као `HEAD~~~`, што представља првог родитеља од првог родитеља од првог родитеља:

```
$ git show HEAD~~~  
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d  
Author: Tom Preston-Werner <tom@mojombo.com>  
Date: Fri Nov 7 13:47:59 2008 -0500
```

Ignore *.gem

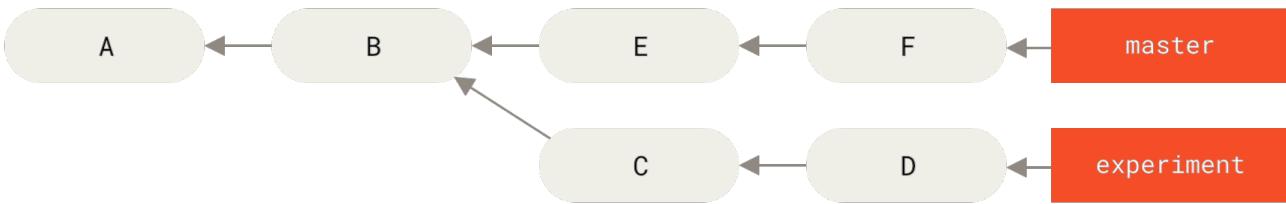
Ове синтаксе такође можете и да комбинујете – са `HEAD~3^2` можете добити другог родитеља претходне референце (под претпоставком да је то комит спајања) и тако даље.

Опсези комитова

Сада када можете да наведете појединачне комитове, хајде да видимо како се наводе опсези комитова. То је од посебне користи за управљање гранама – ако имате дosta грана, можете употребити навођења опсега за одговоре на питања као што је „који рад на овој грани још увек спојен у моју главну грану?”

Двострука тачка

Најчешћи начин задавања опсега је синтаксом са двоструком тачком. Ово у суштини тражи да програм Гит разреши опсег комитова који су доступни од једног, али нису доступни од другог комита. На пример, рецимо да имате историју комитова која изгледа као [Пример историје за избор опсега](#).



Слика 136. Пример историје за избор опсега

Желите да видите шта из ваше експерименталне гране још увек није спојено у вашу `master` грану. Можете затражити да вам програм Гит прикаже лог само тих комитова помоћу `master..experiment` – што значи „сви комитови до којих може да се дође из `experiment` или до којих не може да се дође од `master`“. У циљу јасноће и сажетости, у следећим примерима ће се уместо стварног лог излаза користити слова комит објекта са дијаграма у редоследу у којем би се приказали:

```
$ git log master..experiment
D
C
```

С друге стране, ако бисте желели да видите супротно – све комитове у `master` који нису у `experiment` – можете да замените места гранама. `experiment..master` вам приказује све у `master` до чега не може да се стигне из `experiment`:

```
$ git log experiment..master
F
E
```

Ово је корисно ако желите да `experiment` грану одржавате ажураном и да најпре погледате шта ћете то спојити. Још једна честа употреба ове синтаксе је се види шта ће те управо турнути на удаљени репозиторијум:

```
$ git log origin/master..HEAD
```

Ова команда вам приказује све комитове у вашој текућој грани који се не налазе у `master` грани `origin` удаљеног репозиторијума. Ако извршите `git push` и ваша текућа грана прати `origin/master`, комитови које прикаже `git log origin/master..HEAD` су комитови који ће се пренети на сервер. Једну страну синтаксе можете да изоставите, па програм Гит онда узима да то значи `HEAD`. На пример, исти резултат као у претходном примеру можете добити ако откуцате `git log origin/master..` – програм Гит замењује оно што недостаје на једној страни

са HEAD.

Вишеструке тачке

Синтакса двоструке тачке је корисна као скраћеница, али вероватно бисте желели да наведете више од две гране када желите да задате ревизију, као што је приказ комитова који се налазе у било којој од неколико грана и који се не налазе у грани на којо се тренутно налазите. Програм Гит вам омогућава да то урадите било карактером `^` било са `--not` испред оне референце за коју не желите да видите комитове коју су доступни из ње. Тако да су следеће три команде еквивалентне:

```
$ git log refA..refB  
$ git log ^refA refB  
$ git log refB --not refA
```

То је корисно, јер овом синтаксом у свом упиту можете да наведете више од две референце, што није могуће употребом синтаксе са двоструком тачком. На пример, ако желите да видите све комитове до којих може да се дође из `refA` или `refB`, али не може из `refC`, можете да откуцате било коју од следеће две команде:

```
$ git log refA refB ^refC  
$ git log refA refB --not refC
```

Ово чини врло моћан систем за упит ревизија који би требало да вам помогне у одређивању онога што се налази у вашим гранама.

Трострука тачка

Последња главна синтакса избора опсега је синтакса троструке тачке која наводи све комитове до којих може да се дође из било које од датих референци, али не из обе истовремено. Погледајте претходни пример историје комитова у [Пример историје за избор опсега](#). Ако желите да видите шта се налази у `master` или `experiment` али не и заједничке референце, можете да извршите:

```
$ git log master...experiment  
F  
E  
D  
C
```

Да поновимо, ово вам исписује обичан `log` излаз, али приказује само комит информације за ова четири комита, које се појављују у традиционалном редоследу по датуму комитовања.

У овом случају је уопицајено да се у команди `log` користи прекидач `--left-right` који вам приказује на којој страни опсега се налази сваки од комитова. Тако су подаци још кориснији:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Помоћу ових алата можете много лакше навести програму Гит комит или комитове које желите да испитате.

Интерактивно стејџовање

Програм Git долази са неколико скрипти које олакшавају неке задатке из команде линије. Овде ћете упознати неколико интерактивних комади које вам могу помоћи да лако састављате своје комитове тако да се у њих укључе само одређене комбинације и делови фајлова. Ови алати су од изванредне помоћи ако измените гомилу фајлова, па онда одлучите да је боље да се те измене организују у неколико фокусираних комитова, уместо у једном великом запетљаном комиту. На тај начин можете обезбедити да су ваши комитови логички груписани скупови измена које програмери који раде са вама једноставно могу да провере. Ако команду `git add` извршите са `-i` или `--interactive` опцијом, програм Гит прелази у режим интерактивне љуске и приказује нешто слично овоме:

```
$ 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>
```

Можете видети да вам ова команда приказује доста другачији поглед на ваш стејџ – у основи су то исте информације које добијате са `git status`, али мало сажетије и информативније. Она на левој страни приказује измене које сте поставили на стејџ, а на десној измене које нису на стејџу.

Након тога следи „*Commands*” одељак са командама који вам омогућава да урадите већи број ствари, као што је стављање и уклањање фајлова са стејџа, постављање делова фајлова на стејџ, додавање фајлова који се не прате и приказ разлике онога што је постављено на стејџ.

Постављање и уклањање фајлова са стејџа

Ако на `What now>` одзиву откуцате `u` или `2` (за *update* – ажурирање), скрипта тражи да наведете фајлове које желите да ставите на стејџ:

```
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>>
```

Ако желите да фајлове TODO и index.html поставите на стејџ, можете да откуцате бројеве:

```
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>>
```

* поред сваког фајла значи да је фајл изабран са стављање на стејџ. Ако на **Update>>** одзиву не откуцате ништа већ само притиснете ентер, програм Гит уместо вас узима све изабране фајлове и поставља их на стејџ:

```
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
```

Сада можете видети да се фајлови TODO и index.html налазе на стејџу и да се фајл simplegit.rb још увек не налази на стејџу. Ако у овом тренутку желите да фајл TODO уклоните са стејџа, можете употребити опцију **r** или **3** (за *revert* – враћање):

```
*** 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
```

Ако поново погледате на ваш Гит статус, видећете да сте фајл TODO уклонили са стејџа:

```
*** 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 желите да видите разлику онога што сте поставили на стејџ, употребите команду **d** или **6** (за *diff*). Она вам приказује списак фајлова који се налазе на стејџу, па можете да изаберете оне за које бисте желели да погледате разлику са стејџа. Ово веома личи на задавање **git diff --cached** на командној линији:

```

*** 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">

```

Овим основним командама можете користити режим интерактивног додавања који вам омогућава једноставнији рад са стејџом.

Постављање закрпа на стејџ

Постоји могућност да програм Гит на стејџ постави само одређене делове фајлова. На пример, ако направите две измене у фајлу simplegit.rb, а желите да само једну од њих поставите на стејџ, то ћете веома лако постићи програмом Гит. На интерактивном одзиву откуцајте **р** или **5** (за *patch* – закрпа). Програм Git ће вас питати да наведете фајлове које желите делимично да ставите на стејџ; затим ће за сваки одељак изабраних фајлова да прикаже блокове разлике фајла и питаће вас да ли желите да их поставите на стејџ, један по један:

```

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,?]?

```

У овом тренутку имате доста опција. Ако откуцате `?` приказаће вам се листа ствари које можете да урадите:

```
Stage this hunk [y,n,a,d/,j,J,g,e,?] ? 
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

У општем случају ћете откуцати `y` или `n` ако желите или нећете да поставите текући блок на стејџ, али помаже и да их за неке фајлове поставите све одједном или да одлуку за дати блок одложите за касније. Ако део неког фајла поставите на стејџ а остатак остане ван стејџа, имаћете следећи статус:

```
What now> 1
      staged      unstaged path
1:   unchanged      +0/-1 TODO
2:       +1/-1      nothing index.html
3:       +1/-1      +4/-0 lib/simplegit.rb
```

Статус `simplegit.rb` фајла је врло интересантан. Приказује вам да је неколико линија на стејџу, а неколико није. Овај фајл сте делимично поставили на стејџ. У овом тренутку можете да напустите скрипту за интерактивно додавање и извршите команду `git commit` којом ћете комитовати фајлове делимично налазе на стејџу.

Да бисте урадили делимично постављање фајла на стејџ, нема потребе да се налазите у режиму интерактивног додавања – исту скрипту можете да покренете употребом `git add -p` или `git add --patch` на командној линији.

Уз то, режим закрпе можете да користите и за делимично ресетовање фајлова `reset --patch` командом, за одјављивање делова фајлова `checkout --patch` командом и за сакривање делова фајлова `stash save --patch` командом. Како будемо прелазили на напреднију употребу ових команди, приказаћемо више детаља о свакој од њих.

Скривање и чишћење

Док радите на делу пројекта, ствари су често у нереду и пожелите да пређете на другу грану да мало радите на нечemu другом. Проблем настаје када не желите да комитујете напола

доворшен посао само да би касније могли да се вратите на ово место. Одговор на овај проблем је `git stash` команда.

Скривање узима неуређено стање вашег радног директоријума — то јест ваше измене на фајлове који се прате и стејдовани измене — и чува их на стек незавршених измена који касније у било које време можете поново да примените.



Прелазак на `git stash push`

Крајем октобра 2017, било је опшире дискусије на Гит мејлинг листи у вези тога да ли је команда `git stash save` превазиђена постојећом алтернативом `git stash push`. Главни разлог за ово је што команда `git stash push` уводи опцију скривања изабраних *pathspecs*, нешто што команда `git stash save` не подржава.

`git stash save` не иде никада у додгледно време, тако да не треба да бринете да ће одједном нестати. Али могли бисте да почнете прелаз на `push` алтернативу јер доноси нову функционалност.

Скривање вашег рада

Да бисмо показали како се ово ради, ући ћете у свој пројекат и почети да радите на неколико фајлова и можда ћете стејдовать једну од измена. Ако извршите `git status`, видећете своје неуређено стање:

```
$ 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
```

Сада желите да промените грану, али не желите да комитујете оно на чему још увек радите, тако да ћете сакрити измене. Да бисте ново скривање гурнули на стек, извршите `git stash` или `git stash save`:

```
$ 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")
```

Сада можете видети да је ваш радни директоријум чист:

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

У овом тренутку можете да промените грани и радите на другом месту; ваше измене су сачуване на стеку. Ако желите да видите која скривања имате сачувана, можете да употребите `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
```

У овом случају, раније су сачувана два скривања, тако да имате приступ трима скривеним радовима. Онај који сте управо сакрили можете поново да примените употребом команде која је приказана у испису помоћи оригиналне команде скривања: `git stash apply`. Ако желите да примените неко од старијих скривања, можете га навести по имениу, на следећи начин: `git stash apply stash@{2}`. Ако не наведете скривање, програм Гит претпоставља најскороје скривање и покушава да га примени:

```
$ 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")
```

Можете видети да програм Гит поново мења фајлове које сте чувањем скривања вратили на старо стање. У овом случају, имали сте чист радни директоријум када сте покушали да примените скривање и покушали сте да га примените на исту грани са које сте га и сачували. Није потребно имати чист радни директоријум и бити на истој грани да би се успешно применило скривање. Скривање можете да сачувате на једној грани, касније пређете на другу, па покушате да поново примените измене. Када примењујете скривање, у радном директоријуму такође можете имати и измене и некомитоване фајлове — програм Гит вам враћа конфликте при спајању ако постоји било шта што не може чисто да се примени.

Поново су примењене измене над вашим фајловима, али файл који сте раније ставили на стејџ није поново на стејџу. Да бисте то урадили, команду `git stash apply` морате извршити са `--index` опцијом чиме јој налажете да покуша поново да примени стејџоване измене. Да сте то извршили, вратили бисте се на своју оригиналну позицију:

```
$ 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
```

Опција `apply` покушава да примени само скривени рад—још увек га имате на стеку. Ако желите да га уклоните, извршите `git stash drop` са именом скривања који желите да уклоните:

```
$ 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)
```

Такође можете да извршите `git stash pop` чиме применујете скривање и непосредно након тога га уклањате са стека.

Креативно скривање

Постоји неколико варијанти скривања које такође могу бити корисне. Прва опција која је прилично популарна је `--keep-index` опција команде `stash save`. Она налаже програму Гит да у скривање укључи не само сви садржај на стејџу, већ и да га истовремено остави у индексу.

```
$ 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
```

Још једна уобичајена ствар коју можете да урадите скривањем је то да осим фајлова који се прате скријете и фајлове који се не прате. Подразумевано, команда `git stash` ће сачувати само измене и стејџоване фајлове који се прате. Ако наведете `--include-untracked` или `-u`,

програм Гит ће такође сакрити и све креиране фајлове који се не прате. Међутим, укључивање фајлова који се не прате у скривање ипак неће укључити и фајлове који се **експлицитно** игноришу; ако желите да се и они укључе у скривање, употребите **--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
$
```

На крају, ако наведете заставицу **--patch**, програм Гит неће сакрити две што је изменјено већ ће вас интерактивно питати које од промена желите да сакријете, а које хоћете да задржите у радном директоријуму.

```
$ 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,?]?
```

Stash this hunk [y,n,q,a,d,,e,?]?

```
 Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Креирање гране из скривања

Ако сакријете неки рад, оставите га тако неко време и наставите даље на грани са које сте сакрили рад, могуће је да ћете имати проблема када пожелите да поново да га примените. Ако примењивање покуша да измени фајл који сте ви изменили након скривања, имаћете конфликт спајања и мораћете покушати да га разрешите. Ако желите једноставнији начин да поново тестирате скривене измене, можете да извршите **git stash branch <име нове**

`гране`», која вам креира нову грану са задатим именом, одјављује комит на којем сте били када сте сакрили рад, тамо поново примењује ваш рад, па брише скривање у случају да се успешно применило:

```
$ 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)
```

Ово је фина скраћеница за једноставни опоравак скривеног рада и наставак посла на њему у новој грани.

Чишћење вашег радног директоријума

Конечно, можда нећете хтети да сакријете неки рад или фајлове из радног директоријума, већ једноставно хоћете да их се отарасите; команда `git clean` служи управо за то.

Неки уобичајени разлози за то би могли бити уклањање „крхотина” (*crufts*) које су генерисане спајањима или спољним алатима, или уклањање остатака након изградње како би се покренула чиста изградња.

Требало би да будете прилично опрезни са овом командом јер је дизајнирана да уклања фајлове из радног директоријума који се не прате. Ако се предомислите, често нема враћања садржаја ти фајлова. Безбеднија опција је да извршите `git stash --all` да уклоните све, али да га сакријете.

Ако желите из свог радног директоријума да уклоните фајлове крхотина, урадићете то помоћу команде `git clean`. Да из радног директоријума уклоните све фајлове који се не прате, извршите `git clean -f -d`, што ће уклонити све фајлове и поддиректоријуме који након уклањања фајлова остану празни. Опција `-f` значи *force* (принудно) или „уради како кажем” и неопходна је у случају до Гит конфигурациона променљива `clean.requireForce` није експлицитно постављена на `false`.

Ако некада пожелите да видите шта би команда урадила, покрените је са опцијом `-n` што значи „изврши пробу и реци ми шта би се уклонило”.

```
$ git clean -d -n  
Would remove test.o  
Would remove tmp/
```

Команда `git clean` ће подразумевано да уклони само непраћене фајлове који се не игноришу. Неће се уклонити било који фајл који се подудара са шаблоном из ваше `.gitignore` или неке друге листе фајлова који се игноришу. Ако и те фајлове желите да уклоните, као на пример све `.o` фајлове генерисане изградњом тако да можете покренути потпуно чисту изградњу, додајте `-x` комади чишћења.

```
$ 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 знате шта ће урадити команда `git clean`, увек је најпре покрените са `-n` и проверите резултат пре него што `-n` замените у `-f` и заиста извршите чишћење. Други начин на који можете бити опрезни у вези процеса је да га покренете са заставицом `-i` или „interactive”.

То покреће команду чишћења у интерактивном режиму.

```
$ 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>
```

На овај начин можете интерактивно да пролазите преко сваког фајла појединачно или да интерактивно наведете шаблоне за брисање.

Постоји чудна ситуација у којој може бити потребе да примените посебну принуду када од програма Гит тражите да очисти ваш радни директоријум. Рецимо да се налазите у радном директоријуму под којем сте копирали или клонирали друге Гит репозиторијуме (могуће као подмодуле), чак ће и команда `git clean -fd` одбити да обрише те директоријуме. У таквим случајевима чишћење морате да нагласите додавањем опције `-f`.



Потписивање вашег рада

Програм Гит је криптографски обезбеђен, али не штити од људских грешака. Ако преузимате рад других људи са интернета и желите да проверите да ли комитови заиста долазе од поузданог извора, програм Гит има неколико начина да потпише и провери рад помоћу програма GPG.

Увод у GPG

Најпре, да бисте било шта потписали, морате да конфигуришете програм GPG и да инсталirate свој лични кључ.

```
$ 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 немате инсталiran кључ, можете да га генеришете са `gpg --gen-key`.

```
gpg --gen-key
```

Једном када имате приватни кључ којим потписујете, можете конфигурисати програм Гит да га користи за потписивање ствари подешавањем `user.signingkey` конфигурационе поставке.

```
git config --global user.signingkey 0A46826A
```

Сада ће програм Гит подразумевано да користи ваш кључ да потпише ознаке и комитове ако то желите.

Потписивање ознака

Ако имате поставку GPG приватног кључа, сада можете да је користите за потписивање нових ознака. Све што треба да урадите је да уместо `-a` употребите `-S`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Ако над том ознаком извршите `git show`, видећете да је уз њу прикачен ваш GPG потпис:

```
$ 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

iQEcBAABAgAGBQJTZbQ1AAoJEF0+sviABDDrZbQH/09PfE51KPVP1anr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvcimnSDeSvzCpWAH17h8Wj6hhqePmLm91AYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXXVi
RUysgqjcpT8+iQM1PblGfHR4XAhu0qN5Fx06PSaFZhqvWFezJ28/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

Провера ознака

Да бисте проверили потписану ознаку, користите `git tag -v <име-ознаке>`. Ова команда користи програм GPG да провери потпис. Да би ово радило како треба, потребно је да у свом привеску кључева имате јавни кључ потписника:

```
$ 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
```

У случају да немате јавни кључ потписника, добијате нешто овако:

```
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'
```

Потписивање комитова

У новијим верзијама програма Гит (v1.7.9 и новијим), имате могућност да потписујете и појединачне комитове. Ако вас интересује директно потписивање комитова, а не само ознака, све што треба да урадите је да додате **-S** својој `git commit` команди.

```
$ 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
```

Да бисте видели и проверили ове потписе, постоји опција **--show-signature** команде `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

Уз то, команду `git log` можете подесити тако да проверава све потписе које пронађе и да их приказује у свом излазу у `%G?` формату.

```
$ 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
```

Овде можемо видети да је само последњи комит потписан и важећи, а да претходни комитови нису.

У програму Гит верзије 1.8.3 и каснијим, опцијом `--verify-signatures` командама `git merge` и `git pull` може да се наложи да приликом спајања испитају и одбију спајање комита који нема у себи GPG потпис којем се верује.

Ако употребите ову опцију када спајате грану која садржи комитове који нису потписани и важећи, спајање не успева.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Ако спајање садржи само исправне потписане комитове, команда спајања ће вам приказати све потписе које је проверила, па ће наставити даље са спајањем.

```
$ 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(+)
```

Опцију `-S` такође можете користити и са командом `git merge` да би се потписао и сам комит спајања. Следећи пример истовремено проверава да је сваки комит у грани која се спаја потписан и потписује крајњи комит спајања.

```
$ 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(+)
```

Свако мора да потпише

Потписивање ознака и комитова је одлично, али ако одлучите да то користите у вашем уобичајеном процесу рада, мораћете обезбедити да свако у тиму разуме како се то ради. Ако то не урадите, провешћете пуно времена помажући људима око поновног исписивања својих комитова потписаним верзијама. Будите сигурни да разумете GPG и предности потписивања ствари пре него што то усвојите као део свог стандардног процеса рада.

Претрага

Без обзира на величину пројекта, често ћете имати потребу да пронађете место на којем је функција дефинисана или на којем се позива, или да пронађете историју неке методе. Програм Гит вам обезбеђује неколико корисних алата за брзу и лаку претрагу кода и комитова који су сачувани у његовој бази података. Приказаћемо неколико тих команди.

Git Grep

Програм Гит се испоручује са командом под називом `grep` која вам омогућава да једноставно претражујете било које комитовано стабло или радни директоријум на стринг или регуларни израз. У наредним примерима ћемо претраживати код самог програма Гит.

Команда `git grep` ће подразумевано да претражује све фајлове у вашем радном директоријуму. Као прву варијацију можете јој проследити `-n` тако да исписује бројеве линија у којима је програм Гит пронашао подударања.

```
$ 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
```

Уз основну претрагу која је приказана изнад, команда `git grep` нуди и велики број осталих интересантних опција.

На пример, уместо претходног позива, програму Гит опцијом `--count` можете наложити да испис сажме тако да вам прикаже само у којим фајловима постоје подударања и колико подударања је пронашао у сваком од фајлова:

```
$ 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 вас интересује контекст траженог стринга, методу или функцију која га обухвата можете за сваки подударени стринг добити опцијом `-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 */
```

Као што видите, рутина `gmtime_r` се позива и у `match_multi_number` и у `match_digit` функцији у фајлу `date.c` (треће приказано подударање представља само стринг који се јавља у коментару).

Такође можете да тражите и сложене комбинације стрингова заставицом `--and` која обезбеђује да се вишеструка подударања налазе у истој линији. На пример, хајде да потражимо било које линије које дефинишу константу са једним од подстрингова „LINK” или „BUF_MAX” у одређеној старијој верзији Гит кода коју представља ознака `v1.8.0`

(укључићемо и опције `--break` и `--heading` које помажу да се излаз издели у читљивији 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_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

Команда `git grep` има неколико предности у односу на обичне команде претраге као што су `grep` и `ack`. Прва је да је то да је заиста веома брза, а друга да можете вршити претрагу кроз било које Гит стабло, не само по радном директоријуму. Као што смо видели у горњем примеру, појмове смо тражили у старијој верзији извornог кода програма Гит, а не у верзији која је тренутно одјављена.

Git Log претрага

Можда не тражите место на којем се налази појам, већ време* у којем је постојао или када је уведен. Команда `git log` има више моћних алата за проналажење одређених комитова према садржају њихових комит порука или чак према садржају разлике коју уводе.

На пример, ако желимо да пронађемо када је по први пут уведена константа `ZLIB_BUF_MAX`, програму Гит опцијом `-S` (колоквијално названо „пијук” тј. *pickaxe* програма Гит) можемо наложити да нам прикаже само комитове који су изменили број појављивања тог стринга.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Ако погледамо разлике тих комитова, можемо видети да је у `ef49a7a` уведена константа и да је у `e01503b` изменјена.

Ако је потребно да будете још одређенији, опцијом `-G` можете задати регуларни израз по којем ће се вршити претрага.

Линијска претрага лога

Још једна прилично напредна претрага лога је невероватно корисна линијска претрага лога. Једноставно покрените `git log` са опцијом `-L` и приказаће вам се историја функције или линије кода у вашем пројекту.

На пример, ако желимо да видимо сваку измену функције `git_deflate_bound` у фајлу `zlib.c`, извршили бисмо `git log -L :git_deflate_bound:zlib.c`. То ће покушати да одреди шта су границе те функције, па ће онда претражити историју и приказаће нам сваку измену која је учињена над функцијом као низ закрпа почевши од тренутка када је функција креирана.

```
$ 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 програм Гит за ваш програмски језик не може да одреди како да подудари функцију или

методу, можете му помоћи тако што ћете навести регуларни израз. На пример, следеће би урадило исту ствар као и горњи пример: `git log -L '/unsigned long git_deflate_bound/,/^}:/zlib.c`. Такође можете да јој проследите и опсег линија или један број линије и вратиће вам исту врсту приказа.

Поновно исписивање историје

У многим ситуацијама, док радите са програмом Гит, можете пожелети да из неког разлога ревидирате своју историју комитова. Једна од одличних ствари увези програма Гит је што вам омогућава да доносите одлуке у последњем могућем тренутку. Стјом можете одлучити који фајлови иду у које комитове непосредно пре него што направите комит, можете одлучити да још увек нисте желели да радите на нечему командом `git stash` и можете поново исписати историју комитова који су се већ догодили тако да изгледају као да су се догађали на неки другачији начин. Ово може значити измену редоследа комитова, измену порука или мењање фајлова у комиту, гњечење или растављање комитова, или потпуно уклањање комитова – све то пре него што рад поделите са другима.

У овом одељку ћете сазнати како се завршавају ови веома корисни послови тако да вашу историју комитова можете учинити да изгледа тачно онако како желите пре него што је поделите са другима.

Не гурајте свој рад док нисте задовољни њиме

Пошто се заиста доста посла ради локално унутар вашег клона, једно од кључних правила програма Гит је да имате доста слободе код локалног преписивања своје историје. Међутим, једном када гурнете свој рад, прича је потпуно другачија и имајте на уму да би рад требало да гурнете као довршен, осим ако немате заиста добар разлог да га измените. Укратко, требало би да избегавате гурање свог рада све док не будете задовољни њиме и спремни да га делите са остатком света.



Измена последњег комита

Измена вашег последњег комита је поновно исписивање историје које ћете вероватно најчешће радити. Често ћете имати потребу да урадите две основне ствари вашем последњем комиту: измена комит поруке, или измена снимка који сте управо сачували тако што ћете додавати, мењати и уклањати фајлове.

Ако само желите да измените вашу последњу комит поруку, ствар је проста:

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

Ова комада учитава претходну комит поруку у сесију едитора у којем поруку можете да измените, сачувате, па напустите едитор. Када измене сачувате и напустите едитор, он уписује нови комит који садржи ажурирану комит поруку и поставља га као нови последњи комит.

С друге стране, ако желите да измените *садржај* последњег комита, процес тече у

основи на исти начин — најпре направите измене које мислите да сте заборавили, ставите их на стејц, па `git commit --amend` замењује тај последњи комит вашим новим, унапређеним комитом.

Морате бити опрезни са овом техником јер мењање значи и измену SHA-1 вредности комита. То је као веома мало ребазирање — не мењајте свој последњи комит ако сте га већ гурнули.

Измењени комит може (али и не мора) захтевати измену комит поруке

Када мењате комит, имате могућност да промените и комит поруку и садржај самог комита. Ако је измена комита значајна, скоро обавезно би требало и да ажурирате комит поруку тако да обухвати и тај изменjeni садржај.



С друге стране, ако су ваше измене погодно тривијалне (исправка смешне грешке у куцању или додавање фајла који сте заборавили да ставите на стејц) тако да ранија комит порука у потпуности важи, можете једноставно да направите измене, поставите их на стејц и избегнете потпуно непотребну сесију едитора са:

```
$ git commit --amend --no-edit
```

Измена више комит порука одједном

Ако желите да измените комит који се налази даље у вашој историји, морате да употребите сложеније алате. Програм Гит нема алат за измену историје, али можете употребити `rebase` да ребазирате низ комитова на HEAD на којем су оригинално били базирани umesto да их премештате на неки други. Интерактивним `rebase` алатом затим можете да се зауставите након сваком комита који желите да измените и промените му поруку, додате фајлове или урадите штагод да желите. Ребазирање се извршава интерактивно додавањем опције `-i` команди `git rebase`. Морате навести колико далеко уназад желите да поново испишете комитове тако што команди задате на који комит да започне ребазирање.

На пример, ако желите измените последње три комит поруке, или било коју од порука из те групе, као аргумент команди `git rebase -i` наводите родитеља последњег комита који желите да уредите, што је `HEAD~2^` или `HEAD~3..~3` је вероватно лакше за памћење, јер покушавате да уредите последња три комита, али имајте на уму да заправо наводите четврти претходни комит, тј. родитеља последњег комита четврти желите да уредите:

```
$ git rebase -i HEAD~3
```

Још једном, упamtите да је ово команда ребазирања — поново ће се исписати сваки комит из опсега `HEAD~3..HEAD`, без обзира да ли измените поруку или не. Немојте навести било који комит који је већ гурнут на сервер — збунићете остале програмере тиме што достављате алтернативне верзије једне те исте измене.

Извршавањем ове команде ћете у свом текст едитору добити листу комитова која личи на следећу:

```
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 је приметити да се ови комитови наводе у супротном редоследу у односу на уобичајен приказ командом `log`. Ако извршите `log`, видећете нешто слично овоме:

```
$ 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
```

Уочите обрнути редослед. Интерактивно ребазирање вам приказује скрипту коју ће извршити. Почекеће од комита који наведете у командној линији (`HEAD~3`) и поново ће проћи кроз измене које су уведене сваким од ових комитова, од врха ка дну. Уместо најновијег, на врху приказује најстарији комит јер је то први кроз који ће поново да прође.

Потребно је да уредите скрипту тако да стане на комиту који желите да уредите. Да бисте то урадили, измените реч 'pick' у 'edit' за сваки од комитова на којем желите да скрипта стане.

На пример, ако желите да измените само трећу комит поруку, фајл треба да измените тако да изгледа овако:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Када сачувате и напустите едитор, програм Гит вас враћа назад на последњи комит у тој листи и приказује вам командну линију са следећом поруком:

```
$ 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
```

Ова упутства вам прецизно кажу шта да урадите. Откуцајте:

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

Измените комит поруку и напустите едитор. Затим извршите:

```
$ git rebase --continue
```

Ова команда ће аутоматски применити наредна два комита и завршили сте. Ако на више линија измените *pick* у *edit*, ове кораке можете да поновите за сваки комит који сте променили у *edit*. Програм Гит ће се зауставити сваки пут, омогућити вам да измените комит и наставиће даље када то урадите.

Промена редоследа комитова

Интерактивна ребазирања можете употребити и за промену редоследа или потпуно уклањање комитова. Ако желите да уклоните „Add cat-file” комит и измените редослед у којем се уводе остала два комита, скрипту ребазирања можете променити из:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

у:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

Када то сачувате и напистите едитор, програм Гит премотава уназад вашу грани на родитеља ових комитова, примењује **310154e** па затим **f7f3f6d** и онда се зауставља. Ефективно сте изменили редослед ових комитова и потпуно уклонили „Added cat-file” комит.

Сажимање комитова

Алатом за интерактивно ребазирање је могуће и да узмете низ комитова па да их згужвате у један. Скрипта у поруку ребазирања поставља корисна упутства:

```
#
# 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
```

Ако уместо „pick” или „edit” наведете „squash”, програм Гит примењује и ту и измену непосредно испред ње и даје вам да спојите комит поруке у једну. Дакле, ако желите да из ова три комита направите само један, преправите скрипту тако да изгледа овако:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

Када скрипту сачувате и напустите едитор, програм Гит примењује све три измене, па вас

затим враћа у едитор да спојите три комит поруке у једну:

```
# 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
```

Када то сачувате, имате један комит који је увео све измене из претходна три комита.

Подела комита

Подела комита поништава комит, па онда делимично ставља на стејџ и комитује онолико пута са колико комитова желите да завршите. На пример, рецимо да желите поделити средњи комит од ваша три. Уместо „Update README formatting and add blame” желите да га поделите на два комита: „Update README formatting” као први и „Add blame” као други. То можете урадити у `rebase -i` скрипти тако што ћете изменити упутство на комиту који желите да поделите у „edit”:

```
pick f7f3f6d Change my name a bit  
edit 310154e Update README formatting and add blame  
pick a5f4a0d Add cat-file
```

Затим када вас скрипта постави у командну линију, ресетујете тај комит, узмете ресетоване измене, па од њих направите више комитова. Када сачувате и напустите едитор, програм Гит премотава уназад на родитеља првог комита у вашој листи, примењује први комит (`f7f3f6d`), примењује други комит (`310154e`), па вас враћа у конзолу. Ту можете одрадити мешани ресет тог комита са `git reset HEAD^`, чиме се тај комит ефективно поништава и оставља измене фајлове ван стејџа. Сада можете да стејџујете и комитујете фајлове све док не будете имали неколико комитова, па када завршите, извршите `git rebase --continue`:

```
$ 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
```

Програм Гит примењује последњи комит у скрипти (`a5f4a0d`) и ваша историја изгледа овако:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

Да поновимо, ово мења све SHA-1 вредности свих комитова у вашој листи, па будите сигурни да на листи није ниједан комит који сте већ раније турнули на дељени репозиторијум. Приметите да је последњи комит на листи (**f7f3f6d**) остао неизмењен. Мада се овај комит појављује у скрипти, он је био обележен као „pick” и примењен је пре било каквих измена које уводи ребазирање, па га програм Гит не мења.

Брисање комита

Ако желите да се решите комита, можете га обрисати `rebase -i` скриптом. У листи комитова ставите реч „drop” испред комита који желите да обришете (или једноставно обришите ту линију из скрипте ребазирања):

```
pick 461cb2a This commit is OK
drop 5aecc10 This commit is broken
```

Услед начина на који програм Гит изграђује комит објекте, брисање или измена комита ће изазвати поновно исписивање свих комитова који следе након њега. Што даље идете уназад кроз историју свог репозиторијума, више комитова ће морати поново да се креира. Ово може да изазове много конфликта при спајању ако касније у низу имате доста комитова који зависе од онога који сте управо обрисали.

Ако дођете на пола пута кроз овакво ребазирање и одлучите да то и није тако добра идеја, увек можете да се зауставите. Откуцајте `git rebase --abort`, и ваш репозиторијум се враћа на стање у којем је био пре него што сте покренули ребазирање.

Ако завршите ребазирање и схватите да то није оно што желите, можете употребите `git reflog` да вратите назад старију верзију своје гране. За више информација о команди `reflog`, погледајте [Опоравак података](#).



Дру Деволт је направила згодан практични водич са вежбама који помаже да научите како се користи `git rebase`. Можете га пронаћи на адреси: <https://git-rebase.io/>

Нуклеарна опција: filter-branch

Постоји још једна опција за поновно исписивање историје коју можете користити када је потребно да поново испишете велики број комитова на неки начин који може да се одради скриптом — на пример, глобална измена ваше имејл адресе или уклањање фајла из сваког комита. Команда `filter-branch` може поново да испише огромне откосе ваше историје, тако да вероватно не треба да је користите осим у случају када ваш пројекат још увек није јавни

и неко други није свој рад базирао на комитовима које ћете управо да препишете. Ипак, може бити веома корисна. Научићете неколико уобичајених употреба тако да стекнете идеју о неколико од многих ствари које ова команда може да уради.

Уклањање фајла из сваког комита

Ово се јавља прилично често. Неко случајно комитује огроман бинарни фајл са непромишљеним `git add .`, па желите да га уклоните свуда. Можда сте случају комитовали фајл који је садржао лозинку, а желите да отворите кôд свог пројекта. `filter-branch` је алат који највероватније желите употребити да прочешљате своју комплетну историју. Ако из целе своје историје желите да уклоните фајл под именом `passwords.txt`, употребите `--tree-filter` опцију команде `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Опција `--tree-filter` извршава наведену команду након сваког одјављивања пројекта, па затим поново комитује резултате. У овом случају, из сваког снимка уклањате фајл под именом `passwords.txt`, без обзира на то да ли он постоји у снимку или не. Ако желите да уклоните све случајно комитоване резервне фајлове едитора, можете да извршите нешто као што је `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Видећете како програм Гит поново исписује стабла и комитове па на крају помера показивач грани. У оштем случају је добра идеја да ово урадите у грани за тестирање, па да затим одрадите „hard-reset” ваше `master` грани када одредите да је исход заиста оно што желите. Ако `filter-branch` желите да извршите над свим вашим гранама, проследите јој `--all`.

Постављање поддиректоријума као новог корена

Речимо да сте одрадили увоз из неког другог система за контролу извornог кода и имате поддиректоријуме који немају смисла (`trunk`, `tags`, и тако даље). Ако желите да `trunk` поддиректоријум постане нови корен пројекта за сваки комит, `filter-branch` вам такође долази у помоћ:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cd8e8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Сада је нови корен пројекта оно што се сваки пут налазило у `trunk` поддиректоријуму. Програм Гит ће такође да уклони све комитове који нису утицали на поддиректоријум.

Глобална измена имејл адресе

Још један уобичајени случај је да сте пре почетка рада заборавили да извршите `git config` и поставите своје име и имејл адресу, или да можда желите отворити кôд свог пројекта и

промените своју пословну имејл адресу у приватну. У сваком случају, имејл адресе такође можете одједном променити у више комитова употребом команде `filter-branch`. Морате пазити да измените само своје имејл адресе, тако да користите `--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
```

Ово пролази кроз све комитове и поново исписује сваки комит тако да има вашу нову имејл адресу. Пошто комитови садрже SHA-1 вредности својих родитеља, ова команда мења SHA-1 вредности свих комитова у вашој историји, а не само оних у којима се јавља ваша адреса коју замењујете.

Демистификовани ресет

Пре него што пређемо на више специјализоване алате, хајде да представимо `reset` и `checkout`. Ове команде су два дела програма Гит који уводе највише забуне када се први пут сртнете са њима. Оне обављају доста ствари, тако да њихово потпуно разумевање и правилна употреба изгледа безнадежно. Зато предлажемо једноставну метафору.

Три стабла

Једноставнији начин размишљања о командама `reset` и `checkout` је да себи у мислима представите програм Гит као менаџера садржаја три различита стабла. Овде под „стабло“ уствари мислимо на „колекцију фајлова“, а не на одређену структуру података. Постоји неколико случајева у којима се индекс не понаша баш као стабло, али за наше потребе је једноставније да се за сада овако посматрају ствари.

Програм Гит у свом уобичајеном раду као систем управља и манипулише са три стабла:

| Стабло | Улога |
|--------------------|--|
| HEAD | Снимак последњег комита, наредни родитељ |
| Индекс | Предложени снимак наредног комита |
| Радни директоријум | Изоловано окружење |

HEAD

HEAD је показивач на референцу текуће грану, која је затим показивач на последњи комит направљен на тој грани. Ово значи да ће HEAD бити родитељ наредног комита који се креира. Обично је најједноставније да мислите о HEAD као о снимку **вашег последњег комита**.

Уствари, прилично је једноставно да видите како изгледа тај снимак. Ево примера исписа садржаја актуелног директоријума и SHA-1 контролних сума сваког фајла у HEAD снимку:

```
$ 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
```

Команде `cat-file` и `ls-tree` су „водоводне” команде које се користе за ствари ниског нивоа и обично се не користе у свакодневном раду, али нам помажу да видимо шта се овде дешава.

Индекс

Индекс је ваш **предложени наредни комит**. Овај концепт смо такође звали и „стејџ” (позорницу) програма Гит, јер је то оно шта програм Гит посматра када извршите `git commit`.

Програм Git попуњава овај индекс листом садржаја свих фајлова који су последњи пут били одјављени у ваш радни директоријум и како су изгледали у тренутку одјављивања. Затим неке од тих фајлови замените новим верзијама, и `git commit` то конвертује у стабло за нови комит.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Овде поново користимо `ls-files`, што је више позадинска команда која приказује како тренутно изгледа ваш индекс.

Технички, индекс није структура стабла – уствари је имплементиран као спљоштени манифест – али је за наше потребе доволно близу томе.

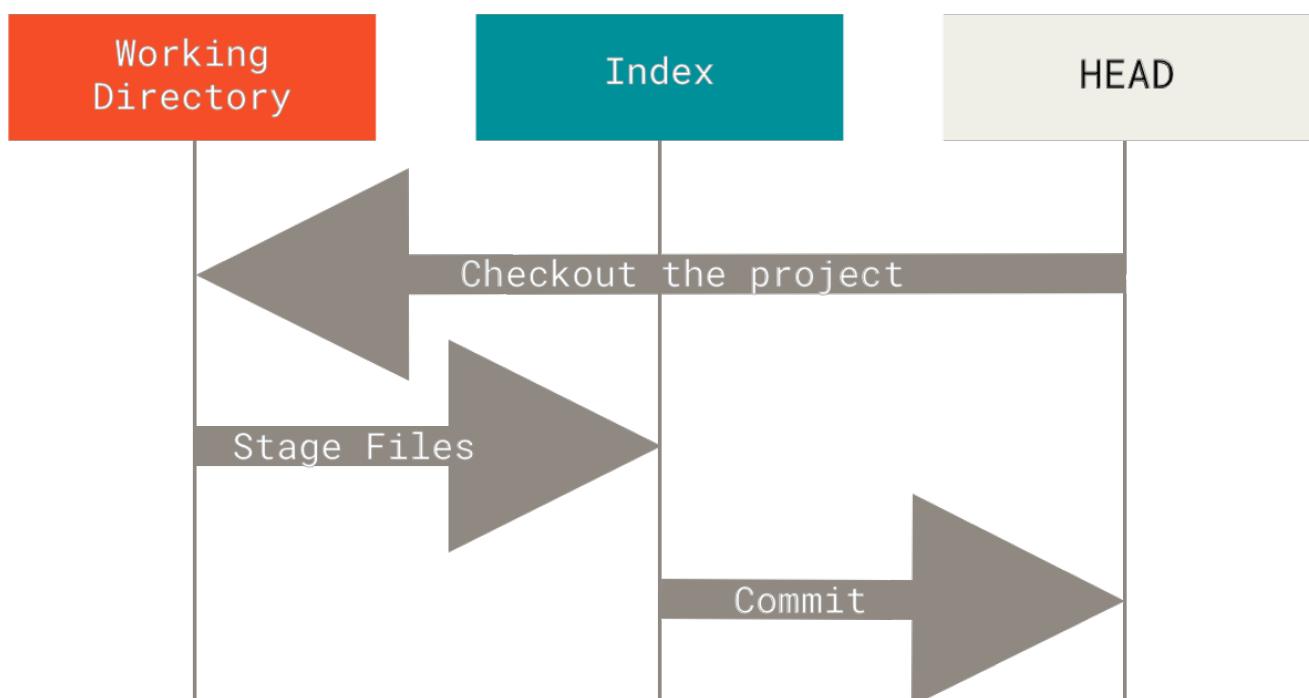
Радни директоријум

Коначно, имате свој радни директоријум. Друга два стабла чувају свој садржај на ефикасан или незгодан начин, унутар `.git` директоријума. Радни директоријум их распакује у стварне фајлове, тако да вам је много лакше да их уређујете. Посматрајте радни директоријум као **изоловано окружење** (*sandbox*), у којем измене можете да испробате пре него што их комитујете у стејџ (индекс), па затим у историју.

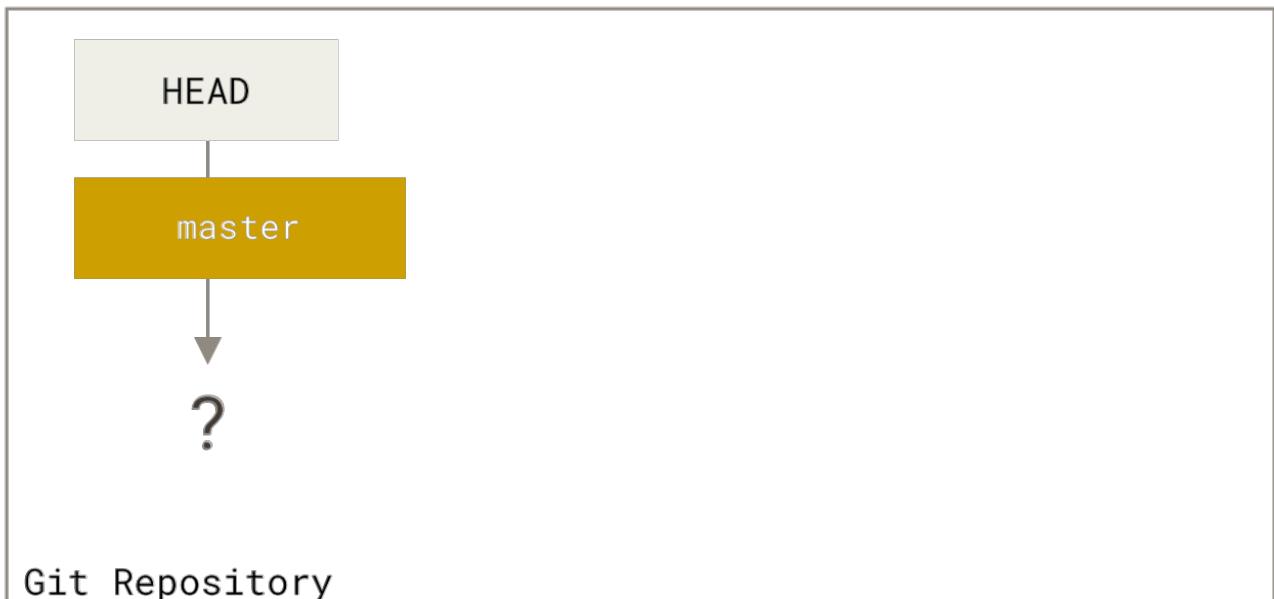
```
$ tree  
.  
├── README  
├── Rakefile  
└── lib  
    └── simplegit.rb  
  
1 directory, 3 files
```

Процес рада

Главна сврха програма Гит је да бележи снимке вашег пројекта у сукцесивно бољим стањима, манипулацијом ова три стабла.

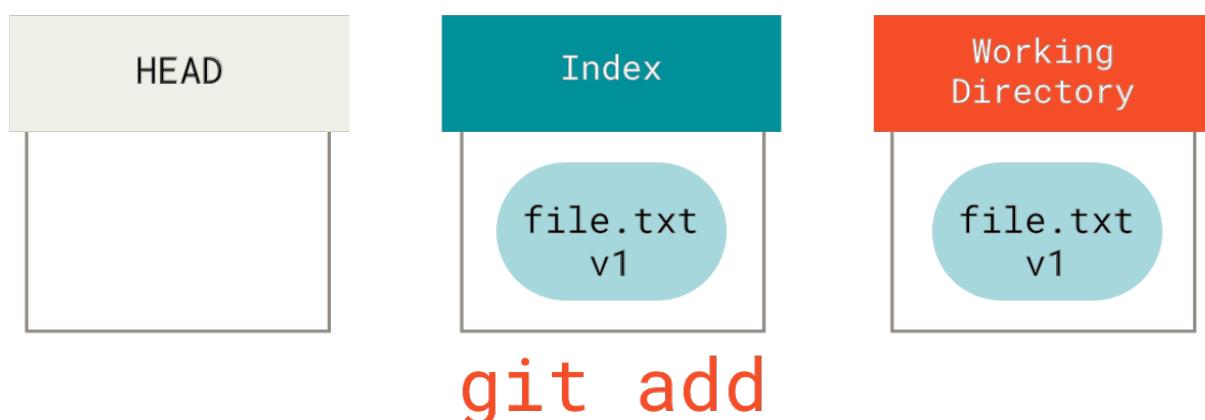
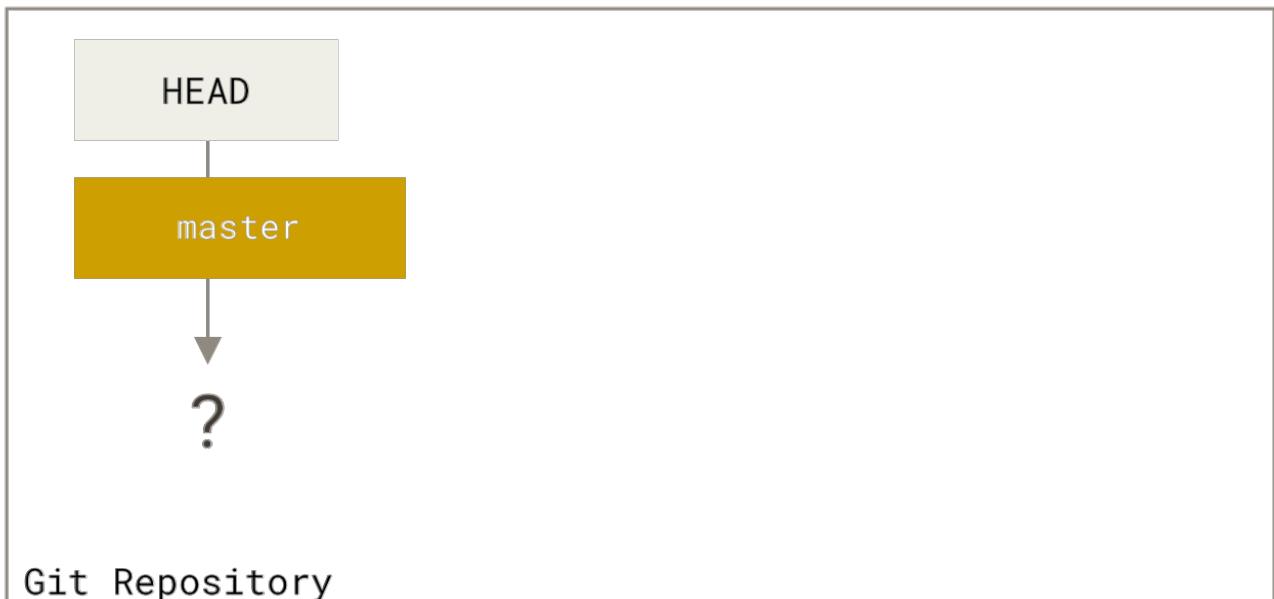


Хајде да визуелизујемо овај процес: рецимо да одете у директоријум који садржи само један фајл. То ћемо звати **v1** фајла, и означаваћемо га плавом бојом. Сада извршимо `git init`, што креира Гит репозиторијум са HEAD референцом која показује на још увек нерођену `master` грану.

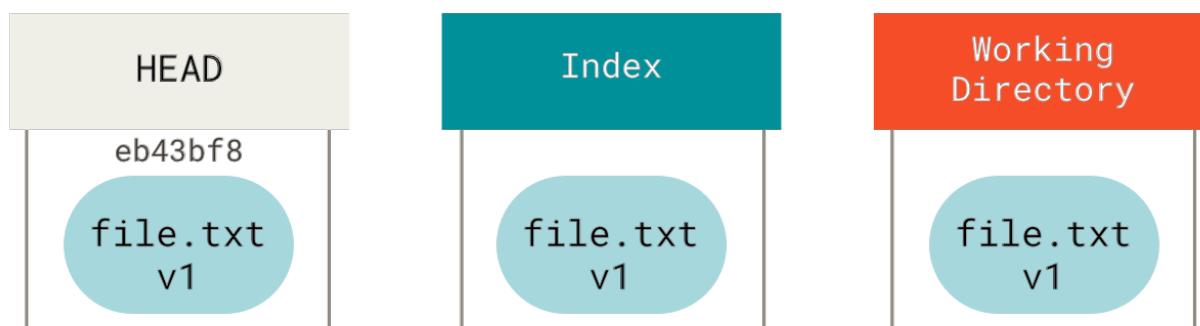
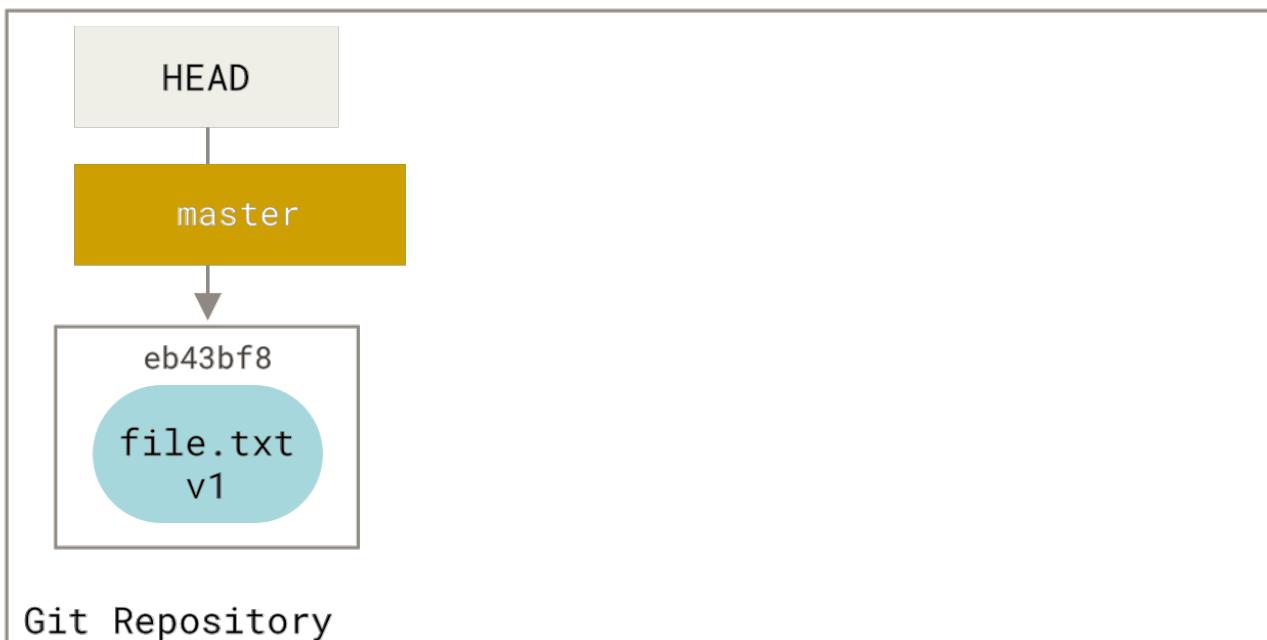


У овом тренутку, само радни директоријум има неки садржај.

Сада жељимо да комитујемо овај фајл, па употребимо `git add` да узме садржај из радног директоријума и да га копира у индекс.



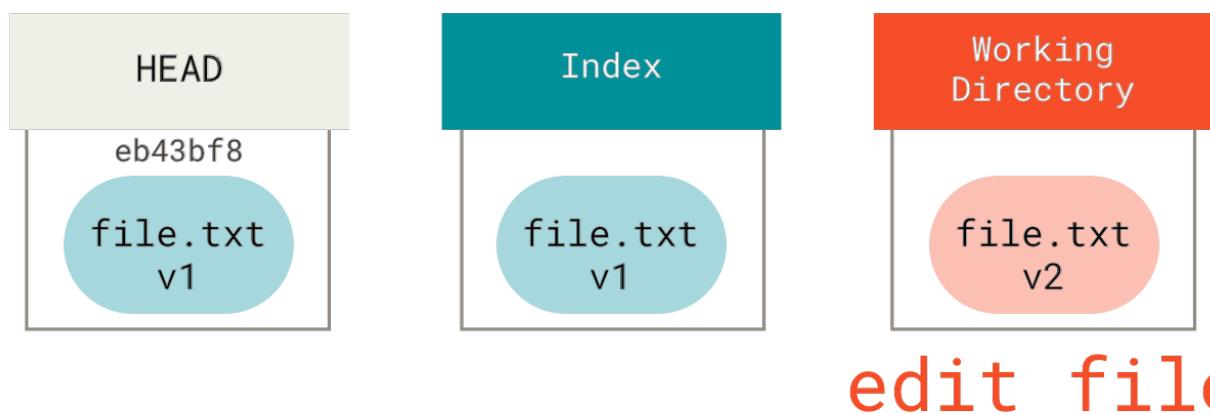
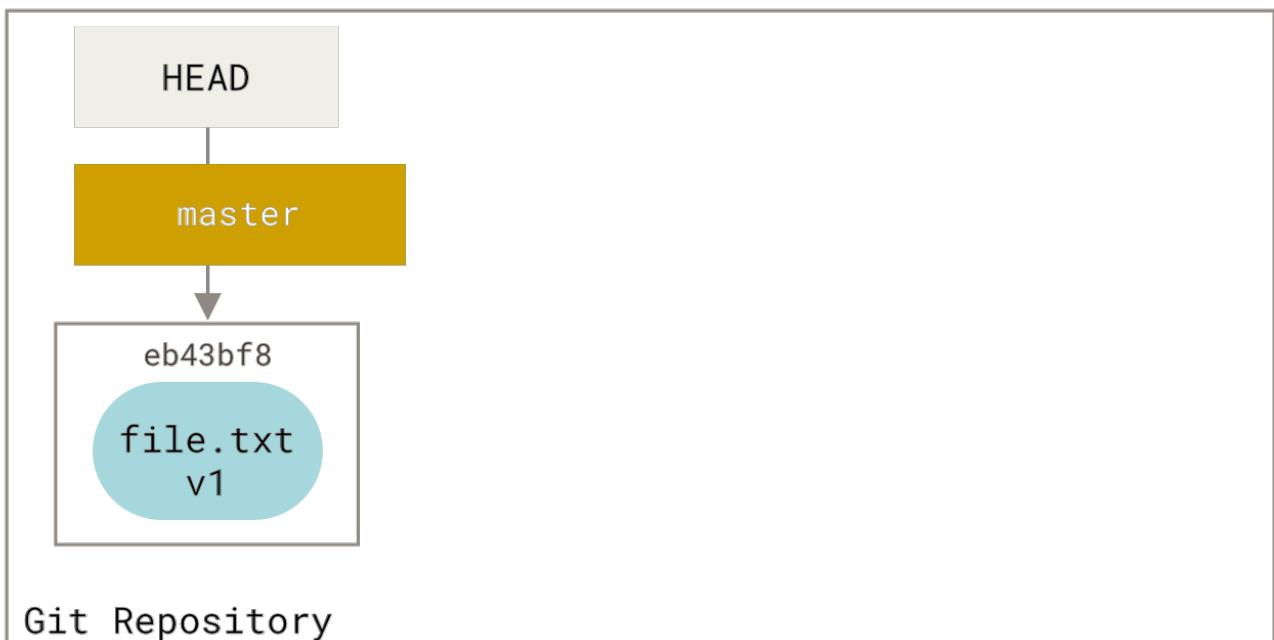
Затим извршимо `git commit`, која узима садржај индекса и чува га као трајни снимак, креира комит објекат који показује на тај снимак и ажурира `master` тако да показује на тај комит.



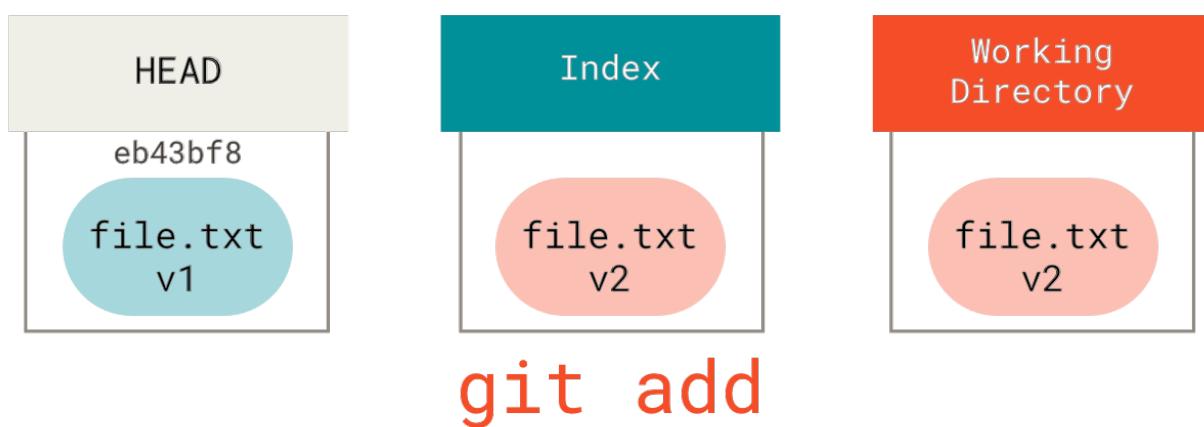
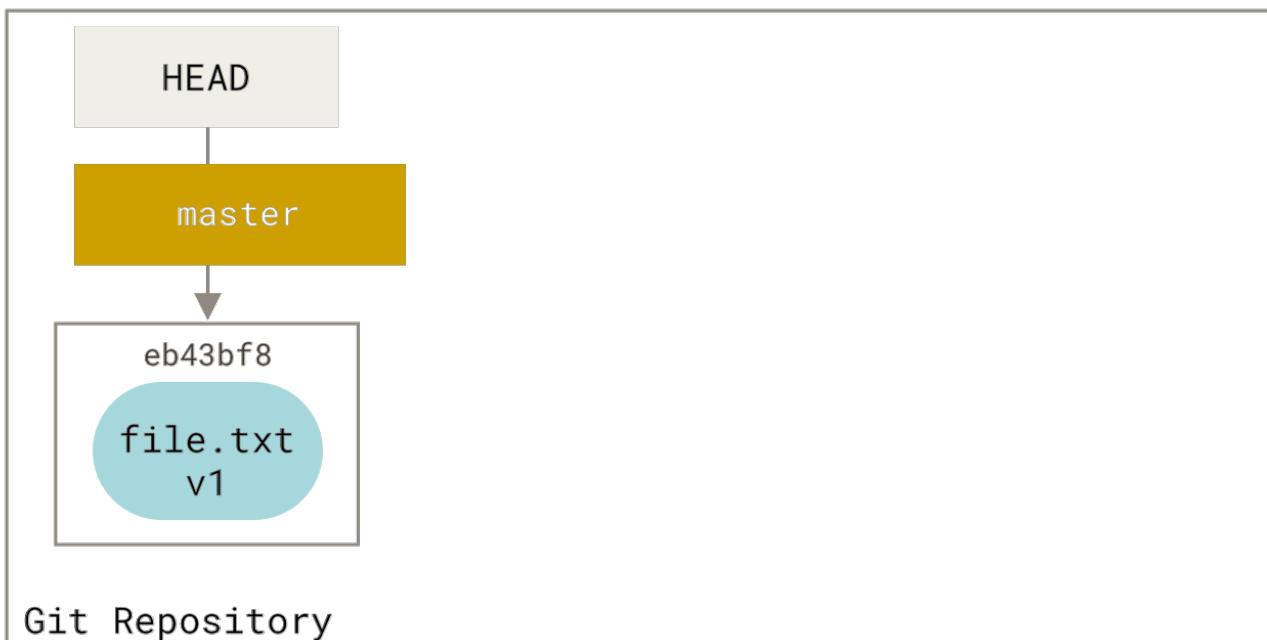
git commit

Ако сада извршимо `git status`, нећемо видети никакве промене јер су сва три стабла потпуно иста.

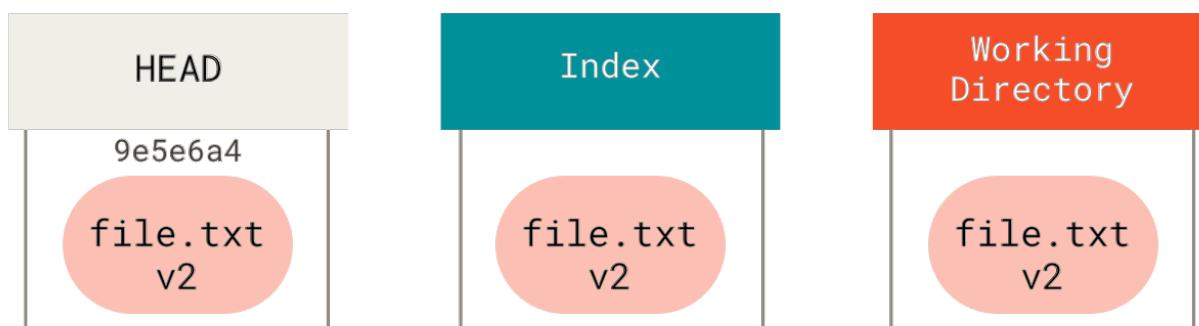
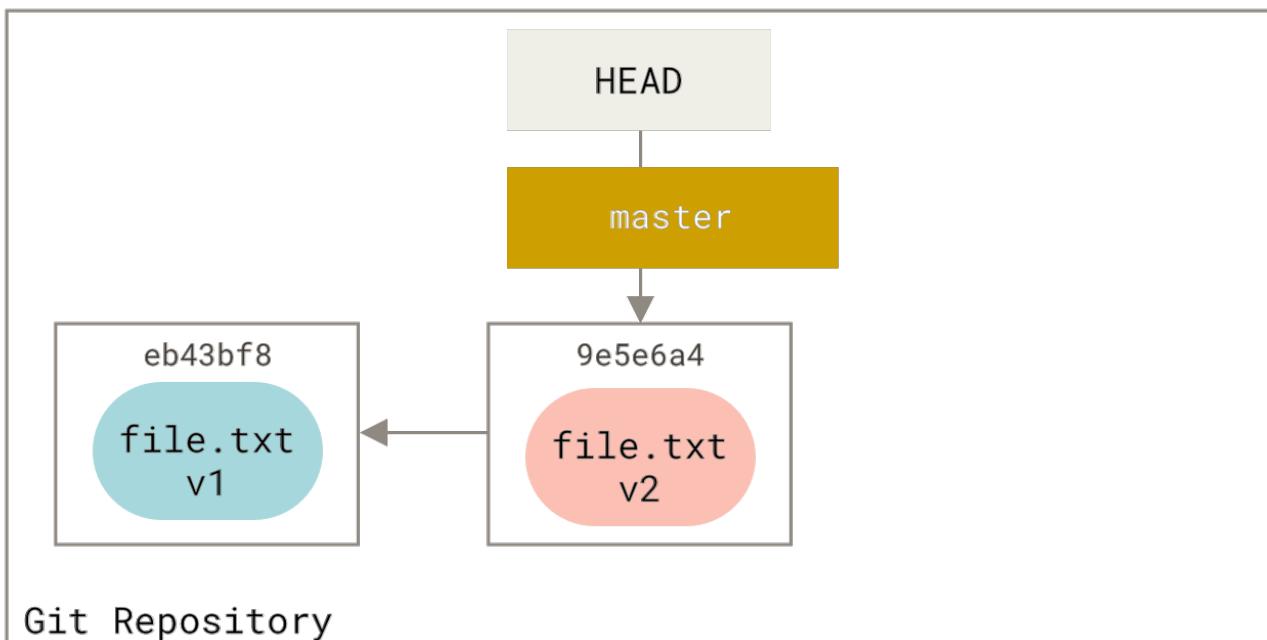
Сада жељимо да изменимо тај фајл и да га комитујемо. Проћи ћемо кроз исти процес; најпре променимо фајл у свом радном директоријуму. Хајде да то назовемо **v2** фајла и да га означавамо црвеном бојом.



Ако сада извршимо `git status` видећемо фајл у црвеној боји ка „Changes not staged for commit” (измене које нису стејџоване за комит), јер се та ставка разликује у односу на индекс и радни директоријум. Затим извршавамо `git add` над њим да га стејџујемо у индекс.



Ако у овом тренутку извршимо `git status` видећемо фајл у зеленој боји под „Changes to be committed” (измене које ће се комитовати) јер се индекс и HEAD разликују – то јест, наш предложени наредни комит се разликује од последњег комита. Коначно извршимо `git commit` да довршимо комит.



git commit

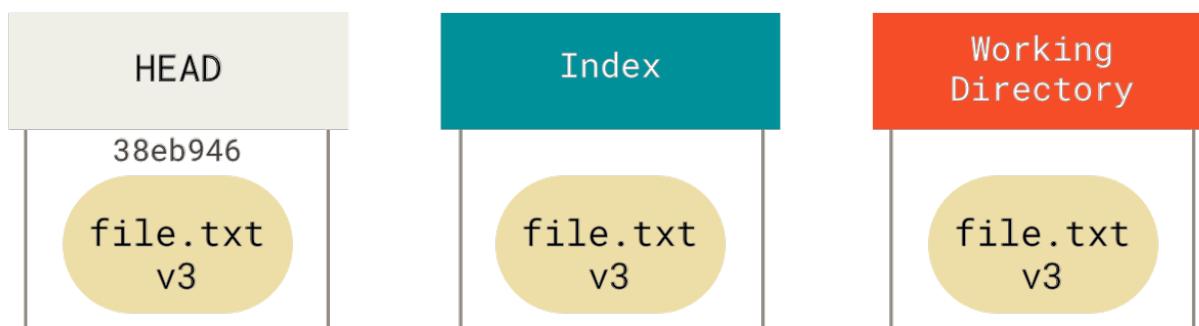
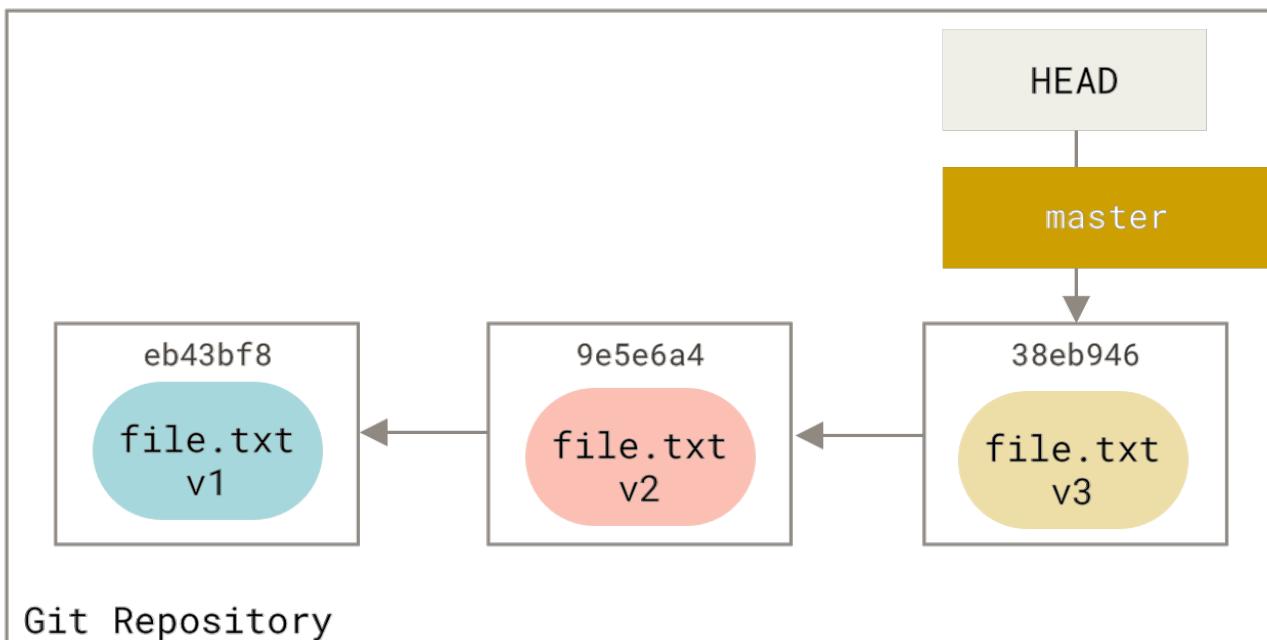
Сада нам `git status` не приказује никакав излаз јер су поново сва три стабла иста.

Прелаз на друге гране или клонирање пролазе кроз сличан процес. Када одјавите грану, то мења **HEAD** тако да показује на референцу нове гране, попуњава ваш **индекс** снимком тог комита, па затим копира садржај **индекса** у ваш **радни директоријум**.

Улога команде `reset`

Команда `reset` има много више смисла када се посматра у овом контексту.

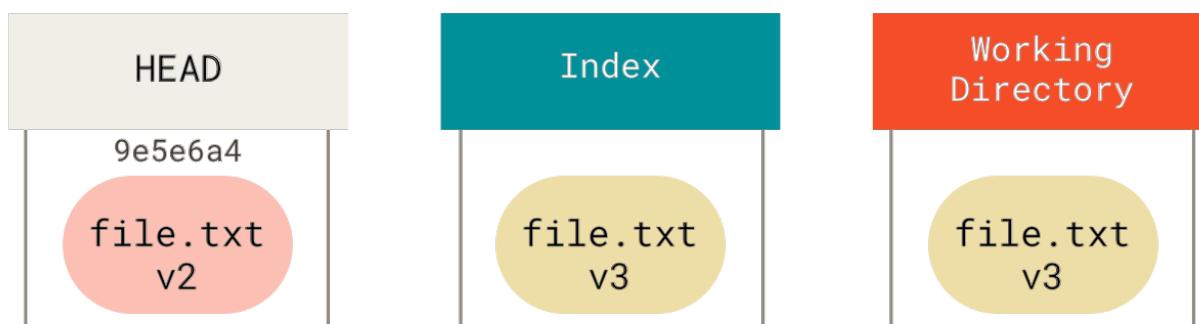
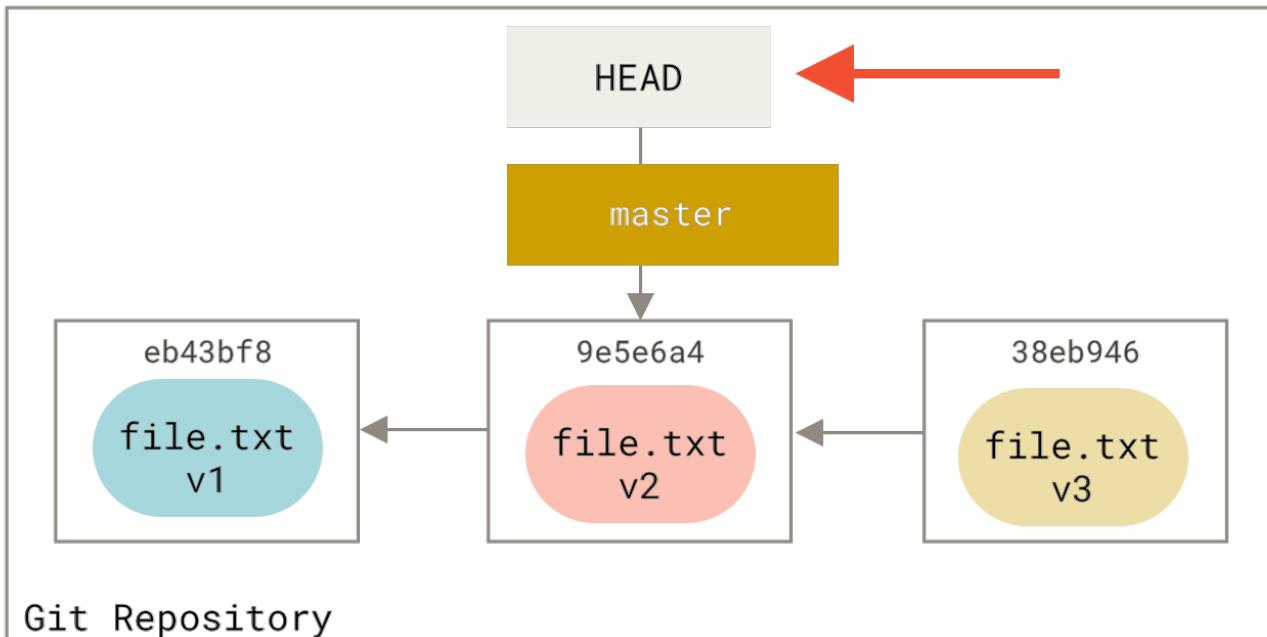
У сврху ових примера, рецимо да смо поново изменили `file.txt` и комитовали га по трећи пут. Тако да наша историја сада изгледа овако:



Хајде да сада прођемо кроз оно што `reset` ради када је позовете. Она на једноставан и предвидив начин директно манипулише ова три стабла. Обавља до три основне операције.

Корак 1: померање HEAD

Прва ствар коју ће команда `reset` урадити је да помери оно на шта показује HEAD. Ово није исто као измена самог HEAD (што је оно што ради команда `checkout`); `reset` помера грану на коју показује HEAD. Ово значи да се HEAD поставља на `master` грану (тј. тада сте тренутно на `master` грани), извршавајући `git reset 9e5e6a4` први корак ће бити да `master` показује на `9e5e6a4`.



git reset --soft HEAD~

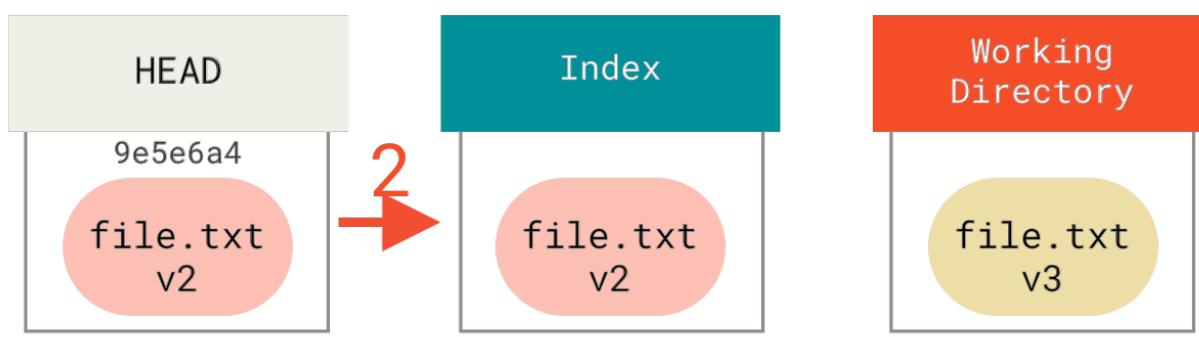
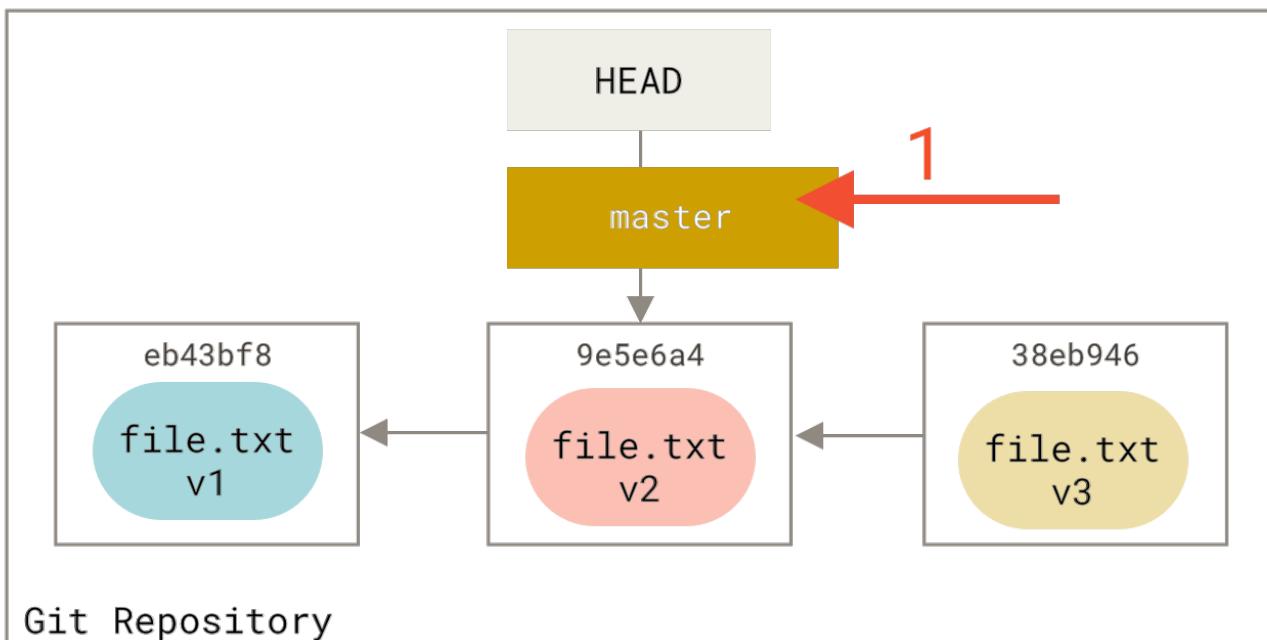
Без обзира на то коју форму команде `reset` са комитом позовете, ово је прва ствар коју ће она увек урадити. У случају `reset --soft`, једноставно ће се ту зауставити.

Застаните на тренутак и погледајте дијаграм да схватите шта се дододило: у суштини је поништена последњу `git commit` команда. Када извршите `git commit`, програм Гит креира нови комит и помера грану на коју показује HEAD на тај комит. Када извршите `reset` назад на `HEAD~` (родитељ од HEAD), грану враћате назад где је била, без измене индекса или радног директоријума. Сада бисте могли да ажурирате индекс и поново извршите `git commit` да постигнете оно што би урадила команда `git commit --amend` (погледајте [Измена последњег комита](#)).

Корак 2: ажурирање индекса (-mixed)

Приметите да ако сада извршите `git status` видећете у зеленој боји разлику између индекса и онога што је сада нови HEAD.

Следећа ствар коју ће урадити `reset` је да ажурира индекс садржајем снимка на који HEAD сада показује.



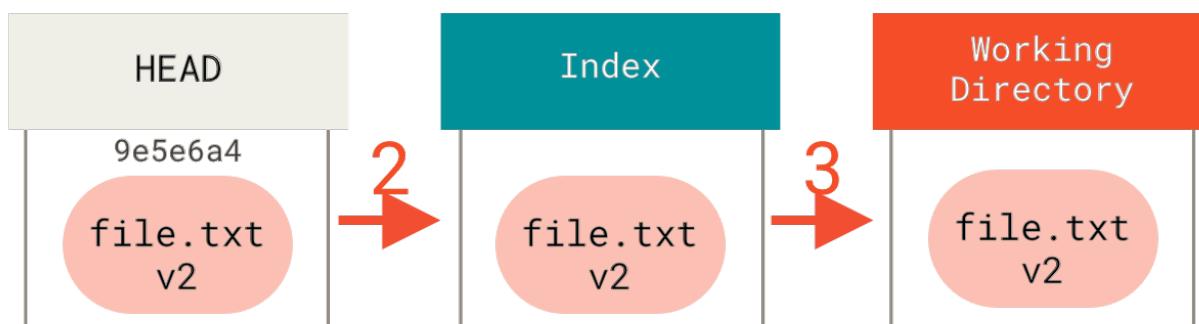
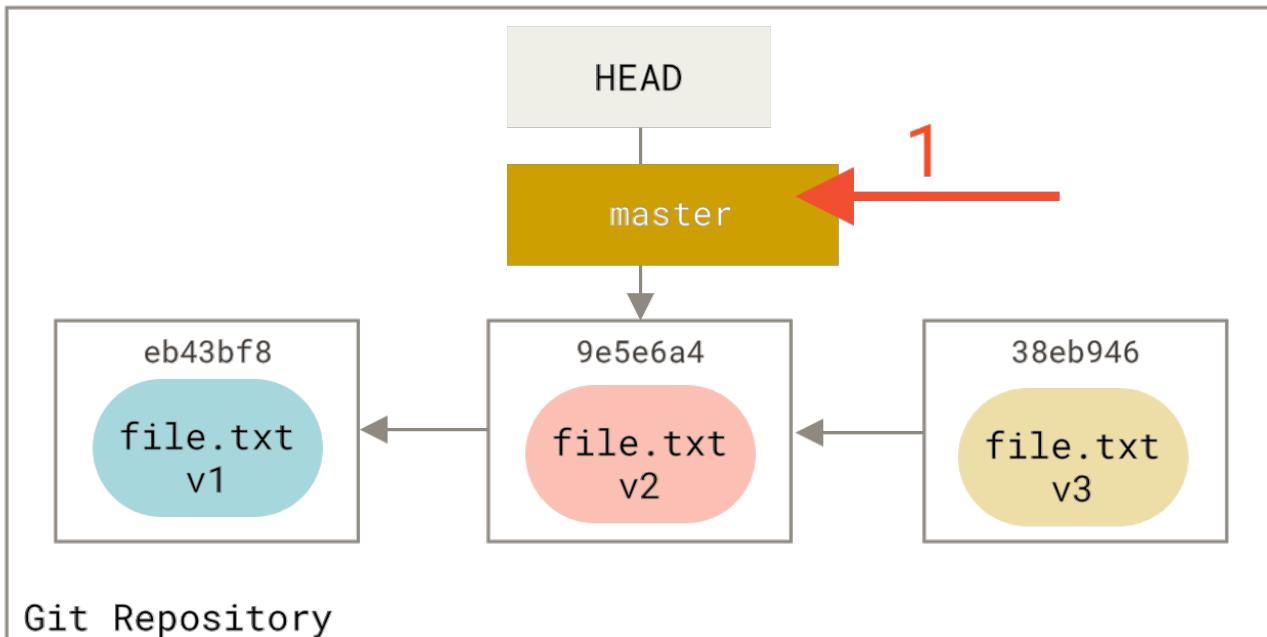
git reset [--mixed] HEAD~

Ако сте задали опцију `--mixed`, команда `reset` ће се зауставити овде. Такође, ово је и подразумевано понашање, па ако уопште не наведете ни једну опцију (само `git reset HEAD~` у овом случају), ово је место на којем ће се команда зауставити.

Погледајте сада још једном тај дијаграм и уочите шта се дододило: поништила је ваш последњи `commit`, али је такође и све уклонила *са стејца*. Премотали сте уназад до места пре покретања свих ваших `git add` и `git commit` команди.

Корак 3: ажурирање радног директоријум (-hard)

Трећа ствар коју ће команда `reset` урадити је да ваш радни директоријум учини да изгледа као индекс. Ако употребите опцију `--hard`, наставиће са извршавањем до ове етапе.



git reset --hard HEAD~

Па хаде да размислимо о ономе што се управо дододило. Поништили сте свој последњи комит, `git add` и `git commit` команде и сви рад који сте урадили у радном директоријуму.

Важно је приметити да је ова заставица (`--hard`) једини начин да команда `reset` буде опасна и један је од врло малог броја случајева у којима програм Гит заиста може да уништи податке. Сваки други начин позива команде `reset` може једноставно да се поништи, али опција `--hard` не може јер насиљно преписује фајлове у радном директоријуму. У овом одређеном случају, још увек имамо **v3** верзију нашег фајла у комиту базе података програма Гит, и могли бисмо да је вратимо ако погледамо у `reflog`, али да је нисмо комитовали, програм Git би преписао фајл и не би било шансе да се он опорави.

Рекапитулација

Команда `reset` преписује ова три стабла у одређеном редоследу, заустављајући се када јој кажете:

1. Помера грану на коју показује HEAD (овде стаје ако задате `--soft`)
2. Чини да индекс изгледа као HEAD (овде стаје нисте задали `--hard`)
3. Чини да радни директоријум изгледа као индекс

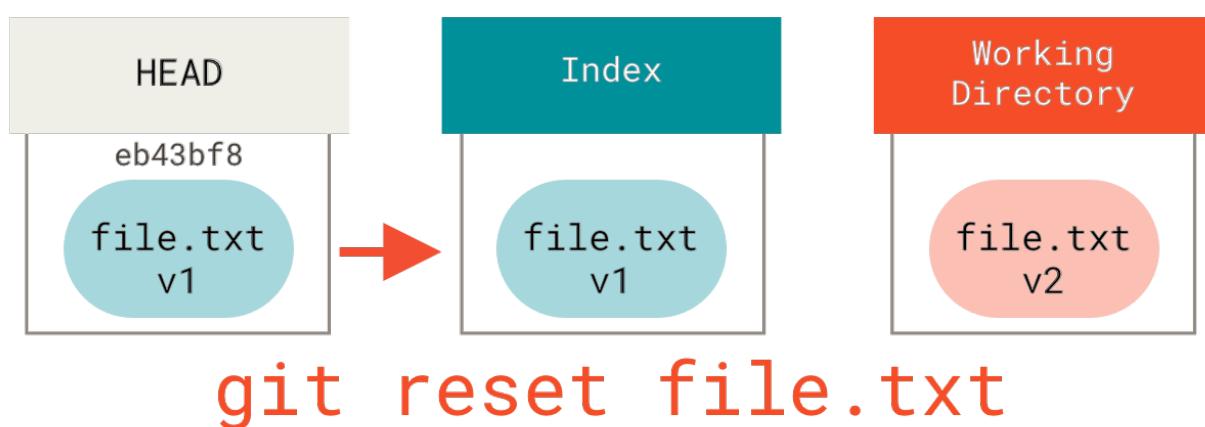
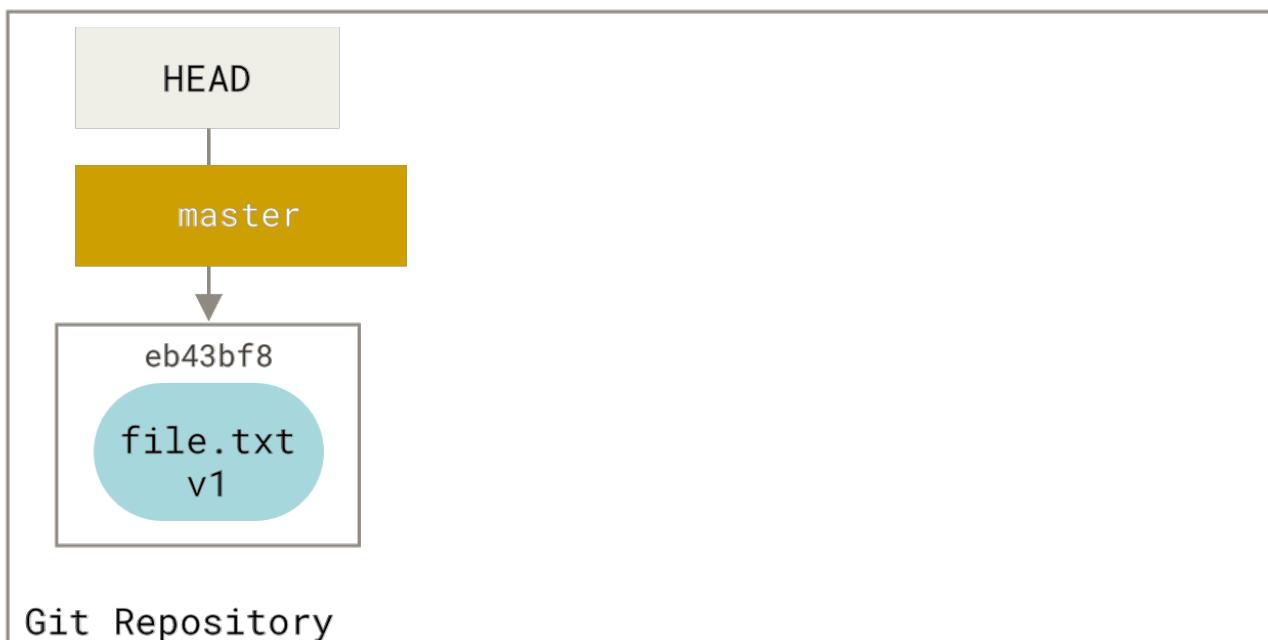
Ресет са путањом

Ово до сада покрива понашање команде `reset` у њеном основном облику, али такође можете да јој задате и путању над којом ће да оперише. Ако наведете путању, команда `reset` ће прескочити корак 1 и ограничити остатак својих акција на одређени скуп фајлова. Ово донекле има смисла – HEAD је само показивач и не може да показује на део једног комита и део неког другог. Али индекс и радни директоријум можете делимично да ажурирате, тако да ресет наставља са корацима 2 и 3.

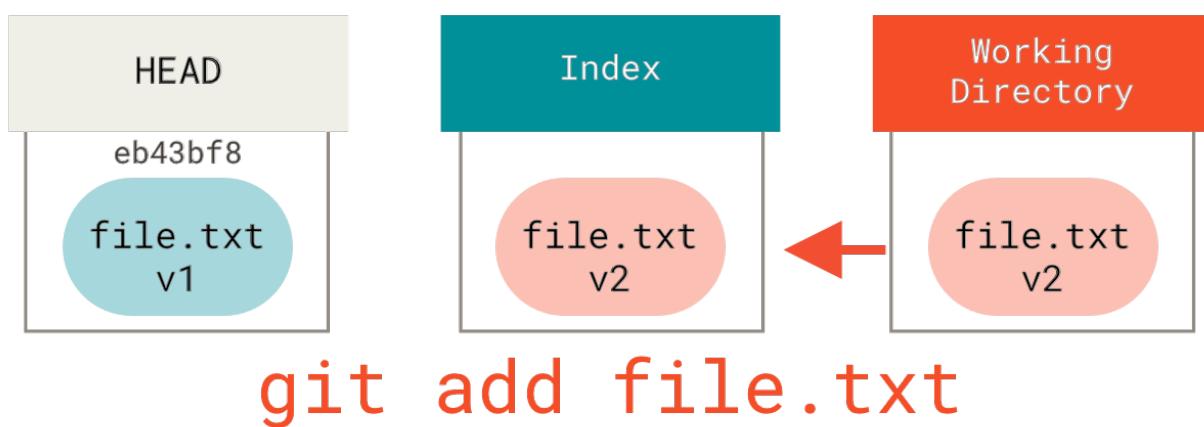
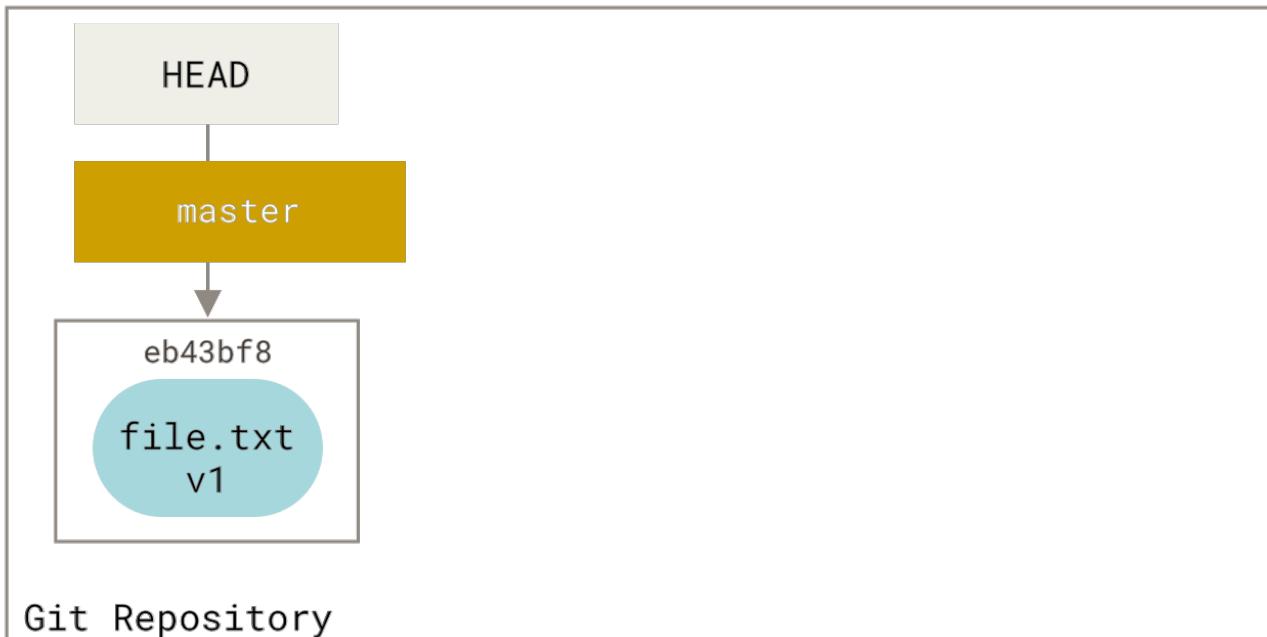
Дакле, претпоставимо да извршимо `git reset file.txt`. Овај облик (пошто нисте навели SHA-1 комита или грану, а нисте навели ни `--soft` ни `--hard`) је скраћеница за `git reset --mixed HEAD file.txt`, што ће:

1. Померити грану на коју показује HEAD (*прескочено*)
2. Учинити да индекс изгледа као HEAD (*овде се зауставља*)

Тако да у суштини само копира `file.txt` из HEAD у индекс.

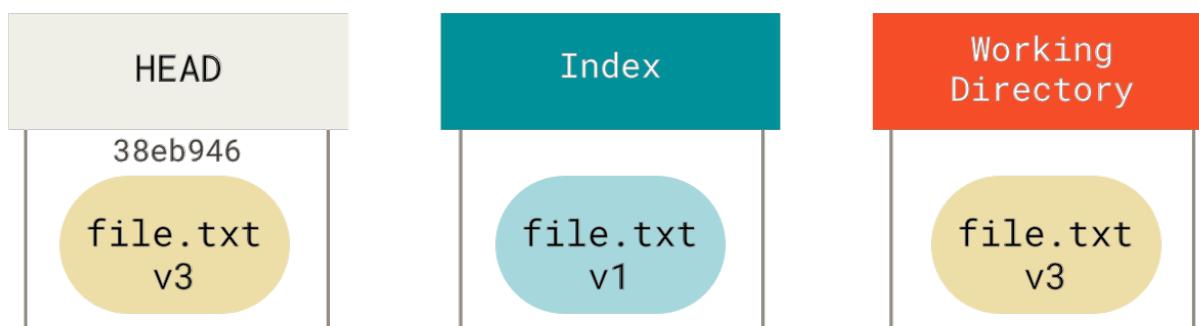
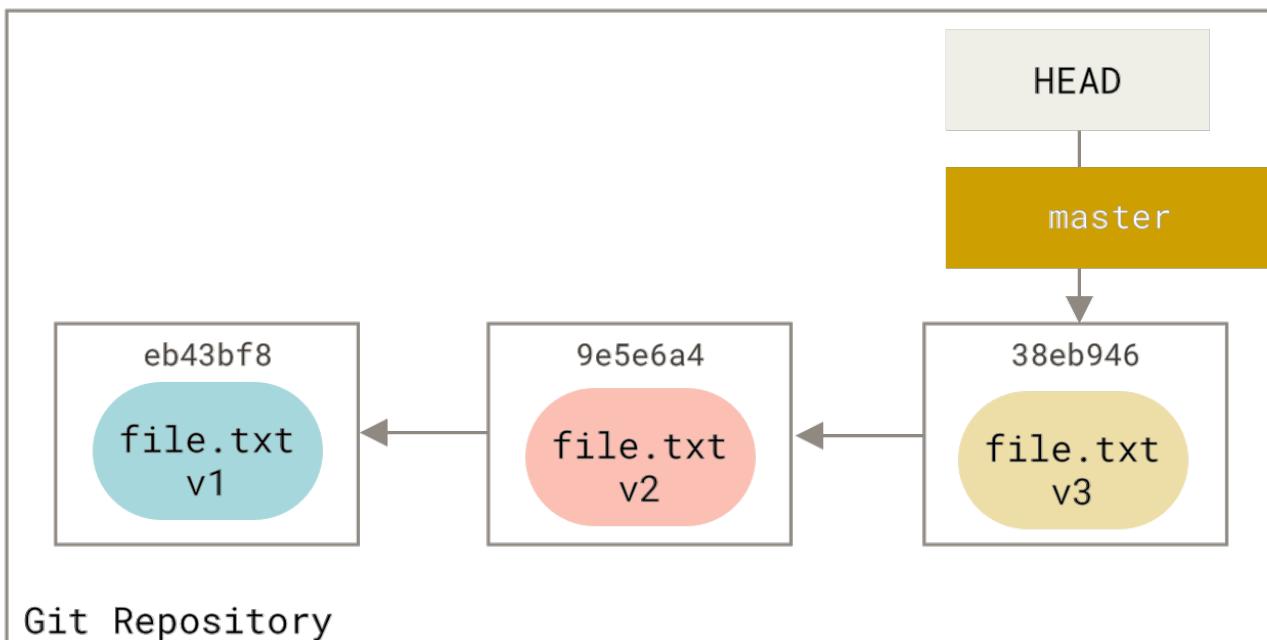


Практични ефекат овога је да се фајл уклања *са стејца*. Ако погледамо дијаграм за ту команду и размислимо о томе шта ради `git add`, оне су потпуно супротне.



Због овога излаз команде `git status` сугерише да ово извршите ако желите да фајл уклоните са стејџа (За више о овоме, погледајте [Уклањање фајла са стејџа](#)).

Исто тако смо могли и да не дозволимо програму Гит да претпостави како смо мислили „повуци податке са HEAD” наводећи одређени комит из којег да повуче верзију фајла. Могли смо да извршимо нешто као што је `git reset eb43bf file.txt`.



`git reset eb43 -- file.txt`

Ово у суштини ради исту ствар као да смо у радном директоријуму вратили садржај фајла назад на **v1**, извршили `git add` над њим, па га затим поново вратили назад на **v3** (а да не прођемо кроз све те кораке). Ако сада извршимо `git commit`, она ће снимити измену која враћа тај фајл назад на **v1**, мада је уствари никада нисмо ни имали поново у радном директоријуму.

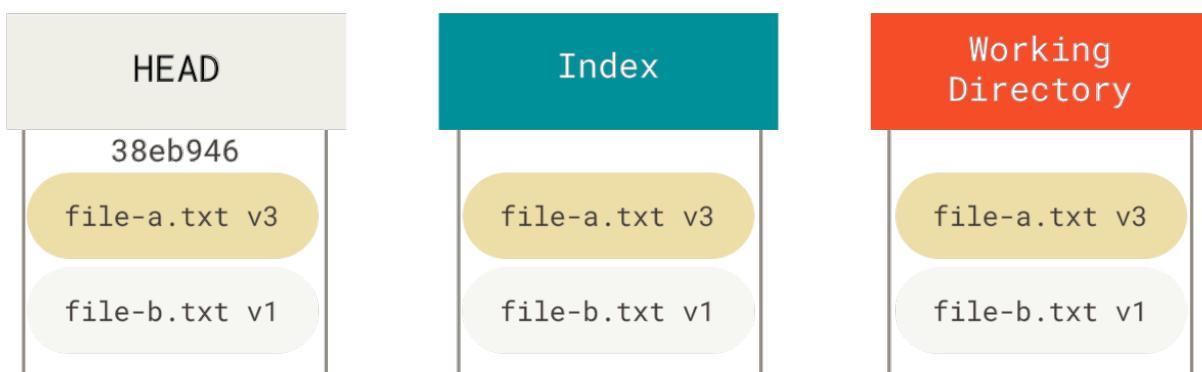
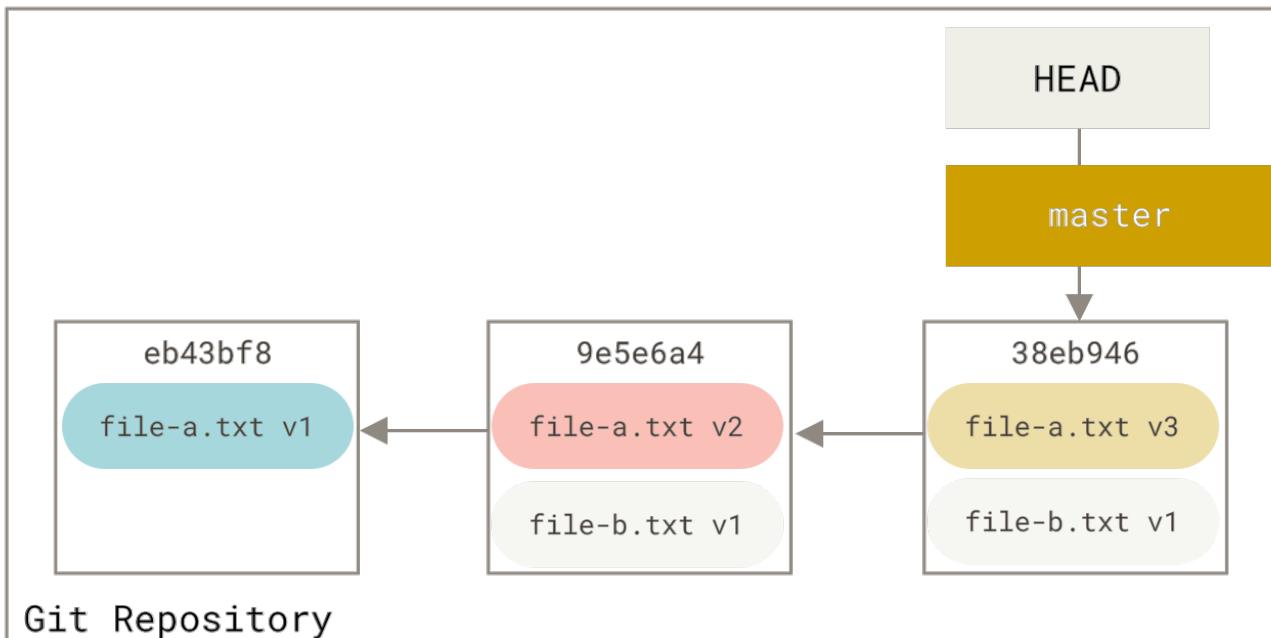
Такође је интересантно приметити да као и `git add`, команда `reset` прихвата опцију `--patch` да са стејџа уклони садржај по принципу комад-по-комад. Тако да садржај селективно можете да уклоните са стејџа или вратите на старије стање.

Гњечење

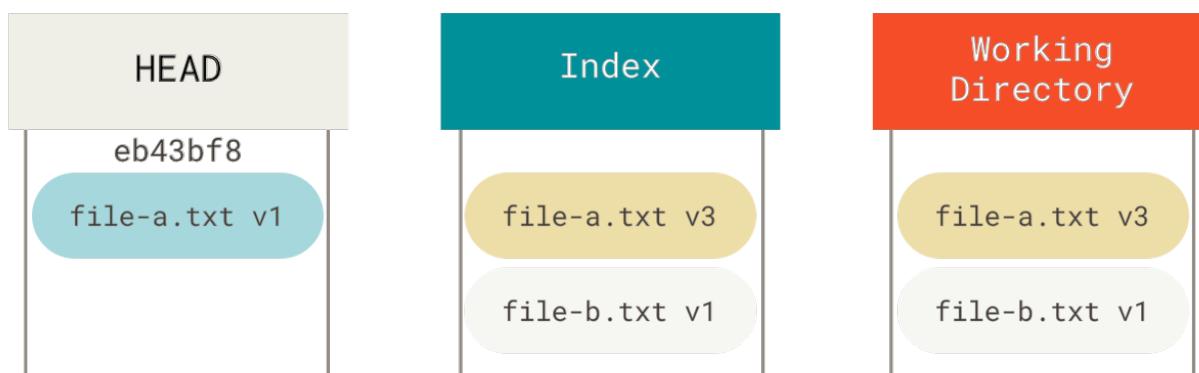
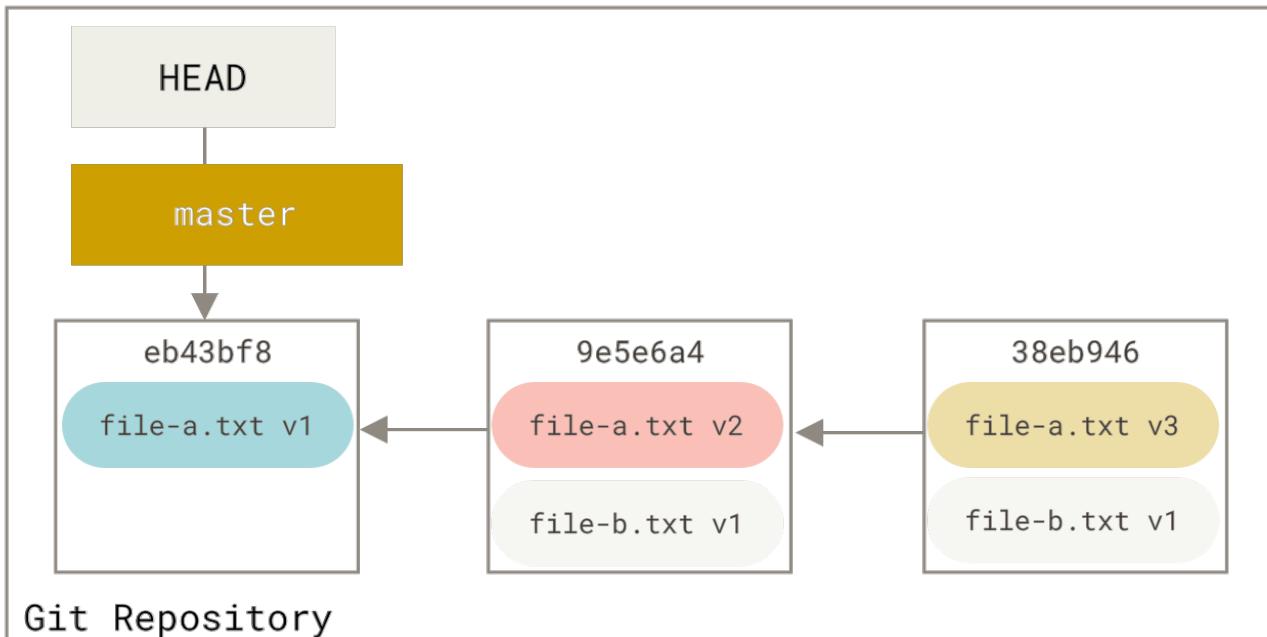
Хајде да видимо како можемо урадити нешто корисно употребом ове управо откривене моћи – гњечење комитова.

Рецимо да имате низ комитова са порукама као што су „ууупс.”, „WIP” (рад је у току) и „заборавио сам овај фајл”. Можете употребити `reset` да их брзо и једноставно згњечите у један једини комит због којег ћете изгледати заиста паметно. [Сажимање комитова](#) приказује други начин да се ово постигне, али у овом примеру је једноставније да се употреби `reset`.

Рецимо да имате пројекат у којем је први комит имао само један фајл, други комит је додао нови фајл и изменио први, и трећи комит је поново изменио први фајл. У другом комиту се налази рад у току и желите да га згњечите.

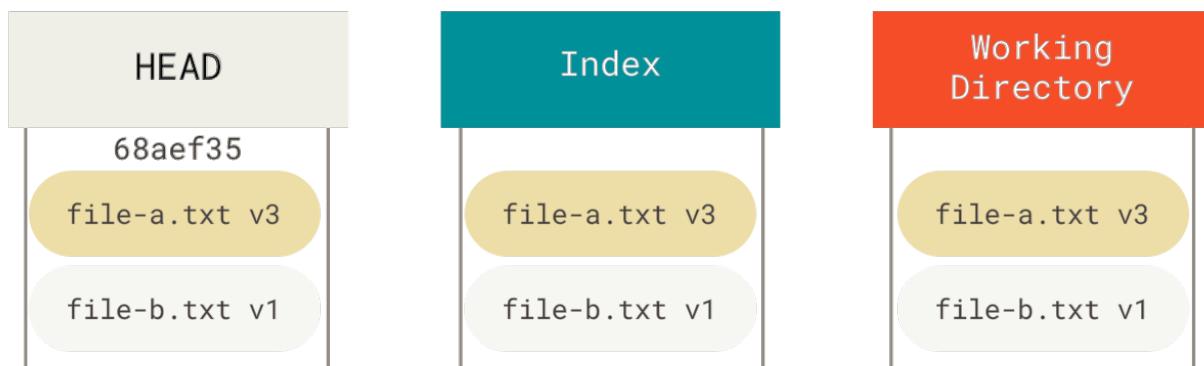
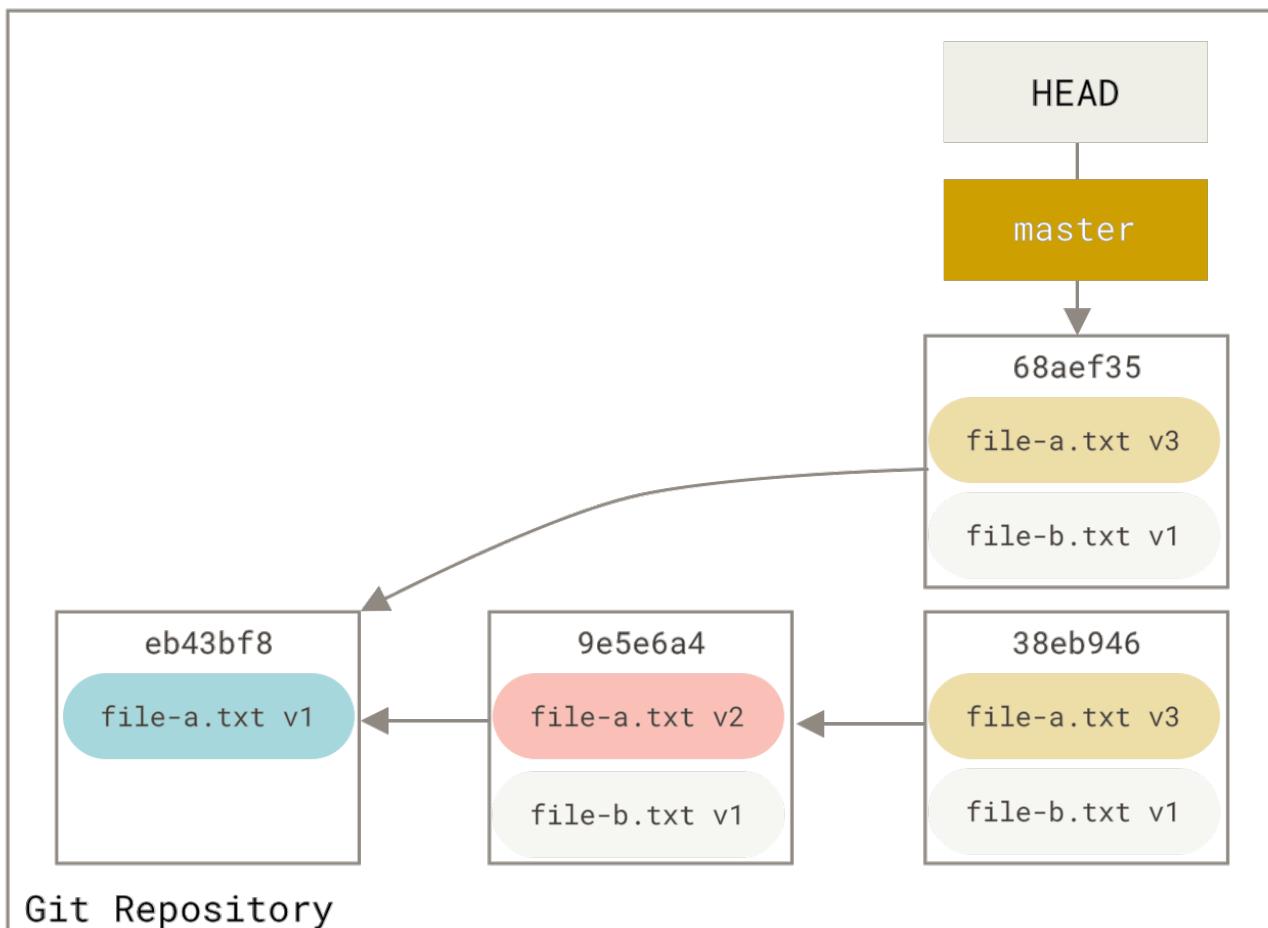


Можете да извршите `git reset --soft HEAD~2` да HEAD грану вратите назад на старији комит (први комит који желите да задржите):



git reset --soft HEAD~2

И да онда једноставно поново извршите `git commit`:



git commit

Сада можете видети да ваша доступна историја, историја коју бисте гурнули, изгледа као да сте направили један комит са `file-a.txt v1`, затим други који је изменио `file-a.txt` на `v3` и додао `file-b.txt`. Комит са `v2` верзијом фајла се више не налази у историји.

Одјавите га

На крају, упитаћете се шта је разлика између `checkout` и `reset`. Као `reset`, и `checkout` манипулише са три стабла, и донекле се разликује у зависности од тога да ли команди задате путању до фајла или не.

Без путањи

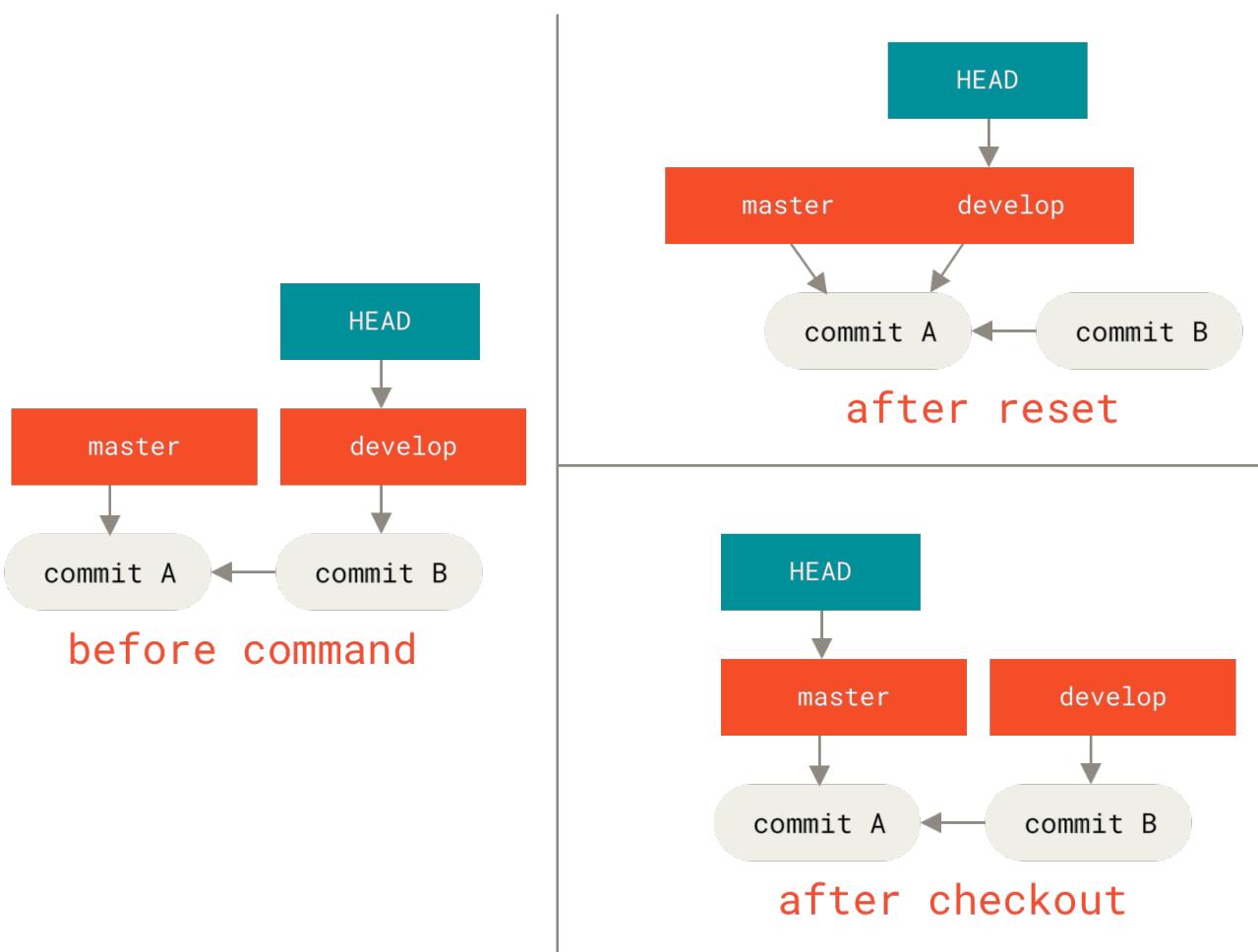
Извршавање `git checkout [грана]` је прилично слично са извршавањем `git reset --hard [грана]` у смислу да вам ажурира сва три стабла тако да изгледају као [грана], али постоје две важне разлике.

Прво, за разлику од `reset --hard`, `checkout` је безбедна за радни директоријум; извршиће проверу којом обезбеђује да вам не одува фајлове које сте изменили. Уствари, још мало је и паметнија – покушава да у радном директоријуму изврши тривијално спајање, тако да ће се ажурирати сви фајлови је *нисте* изменили. С друге стране, `reset --hard` ће једноставно редом заменити све ствари без икакве провере.

Друга важна разлика је начин на који се ажурира HEAD. Док `reset` помера грану на коју указује HEAD, `checkout` помера сам HEAD тако да указује на другу грану.

На пример, рецимо да имамо гране `master` и `develop` које показују на различите комитове, и да се тренутно налазимо на `develop` (тако да HEAD показује на њу). Ако извршимо `git reset master`, сама `develop` грана ће сада показивати на исти комит на који показује и грана `master`. Ако уместо тога извршимо `git checkout master`, `develop` грана се не помера, већ се помера сам показивач HEAD. HEAD ће сада да показује на `master`.

Дакле, у оба случаја померамо HEAD тако да показује на комит A, али *начин* на који то радимо је веома другачији. `reset` помера грану на коју указује HEAD, `checkout` помера сам HEAD показивач.



Са путањама

Други начин да се изврши команда `checkout` је са путањом до фајла, који као и `reset`, не помера показивач HEAD. То је исто као `git reset [грана] фајл` у смислу да ажурира индекс задатим фајлом у том комиту, али такође и преписује фајл у радном директоријуму. Било би потпуно исто као и `git reset --hard [грана] фајл` (ако би вам команда `reset` дозволила да то извршите) – није безбедно по радни директоријум и не помера HEAD.

Такође, као `git reset` и `git add`, команда `checkout` ће прихватити опцију `--patch` која вам омогућава да селективно враћате старо стање садржаја фајла по принципу комад-по-комад.

Резиме

Надамо се да сада разумете команду `reset` и да можете удобније да је користите, али сте вероватно још увек донекле збуњени у вези тога како се прецизно она разликује у односу на команду `checkout` и вероватно не можете да запамтите сва правила различитих начина позивања.

Ево „пушкице” о томе које команде утичу на која стабла. У колони „HEAD” стоји „РЕФ” ако та команда помера референцу (грану) на коју указује HEAD, а „HEAD” ако помера сам HEAD показивач. Посебну пажњу обратите колони ’РД безбедна?’ – ако у њој пише НЕ, застаните на секунд да добро размислите пре извршавања те команде.

| | HEAD | Индекс | Радни директоријум | РД безбедна? |
|--------------------------------------|------|--------|--------------------|--------------|
| Комит ниво | | | | |
| <code>reset --soft [комит]</code> | РЕФ | НЕ | НЕ | ДА |
| <code>reset [комит]</code> | РЕФ | ДА | НЕ | ДА |
| <code>reset --hard [комит]</code> | РЕФ | ДА | ДА | НЕ |
| <code>checkout [комит]</code> | HEAD | ДА | ДА | ДА |
| Фајл ниво | | | | |
| <code>reset (комит) [фајл]</code> | НЕ | ДА | НЕ | ДА |
| <code>checkout (комит) [фајл]</code> | НЕ | ДА | ДА | НЕ |

Напредно спајање

Спајање у програму Гит је у општем случају прилично једноставно. Пошто програм Гит вишеструко спајање неке друге гране чини једноставним, то значи да можете имати веома дуготрајне гране које у ходу можете одржавати ажурирним, често решавајући мале конфликте уместо да будете изненађени огромним конфликтом на крају низова.

Међутим, понекад долази до компликованих конфликтата. За разлику од неких других система за контролу верзије, програм Гит не покушава да буде превише паметан када дође до решавања конфликта при спајању. Филозофија програма Гит је да буде паметан када одређује да ли је решење спајања недвосмислено, али ако постоји конфликт, да се не прави паметан у вези аутоматског решавања. Дакле, ако сувише дugo чекате да спојите две гране

које се брзо разилазе, можете најти на одређене проблеме.

У овом одељку ћемо представити шта могу бити неки од тих проблема и које алате вам програм Гит нуди као помоћ у обради тих компликованијих ситуација. Такође ћемо представити неке другачије, нестандартне врсте спајања које можете урадити, као и како да се избавите из спајања која сте већ обавили.

Конфликти при спајању

Мада смо прешли неке основе решавања конфликта при спајању у [Основни конфликти при спајању](#), за сложеније конфликте програм Git нуди неколико алата који вам помажу да откријете шта се дешава и како да се успешније носите са конфликтом.

Најпре, ако је то уопште и могуће, покушајте обезбедити да вам је радни директоријум чист пре него што покренете спајања која би могла имати конфликте. Ако имате посао у току, или га комитујте у привремену грану или га сакријте. На тај начин можете вратити на старо **све** што овде покушате. Ако у радном директоријуму имате несачуване промене када покушате спајање, неки од ових трикова вам могу помоћи да очувате тај рад.

Хајде да прођемо кроз веома једноставан пример. Имамо супер једноставан Руби фајл који исписује 'hello world'.

```
#!/usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()
```

У нашем репозиторијуму креирајмо нову грану под именом **whitespace** и крећемо да променимо све Јуникс завршетке редова у ДОС завршетке редова, у суштини мењајући сваку линију фајла, али само са празним простором (*whitespace*). Затим променимо линију „hello world” у „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(-)
```

Сада се вратимо назад на нашу **master** грану и додамо документацију функције.

```
$ 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(+)
```

Сада покушамо да спојимо нашу whitespace грану и добијамо конфликте због промена празног простора.

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

Прекид спајања

Сада имамо неколико опција. Најпре, хајде да покажемо како да се избавимо из ове ситуације. Ако можда нисте очекивали конфликте и још увек не желите да се заиста бавите ситуацијом, можете једноставно да се повучете назад из спајања са `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

Опција `git merge --abort` покушава да вас врати на старо стање пре покретања спајања. Једини случајеви када ово не би перфектно могла да уради је ако бисте имали несакривене, некомитоване измене у радном директоријум када сте је покренули. У супротном би

требало да ради како треба.

Ако из неког разлога једноставно желите да почнете из почетка, можете да покренете и `git reset --hard HEAD`, и ваш репозиторијум ће се вратити назад на последње комитовано стање. Упамтите да ће се изгубити сваки рад који није комитован, па будите сигурни да вам не требају никакве промене.

Игнорисање празног простора

У овом посебном случају, конфликти су везани за празан простор. То знамо само зато што је случај једноставан, али је углавном прилично лако да се и у реалним случајевима препозна, јер када се погледа у конфликт, свака линија се уклања на једној страни, па се поново додаје на другој. Програм Гит подразумевано све ове линије види као измене, тако да не може да их споји.

Међутим, подразумевана стратегија спајања може да узме аргументе, а њих неколико су у вези исправног игнорисања измена празног простора. Ако приметите да у спајању имате доста проблема везаних за празан простор, можете једноставно да прекинете спајање па да га покренете поново, али овај пут са `-Xignore-all-space` или `-Xignore-space-change`. Прва опција **потпуно** игнорише празан простор када пореди линије, а друга третира низове од једног или више празних карактера као еквивалентне.

```
$ 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(-)
```

Пошто у овом случају стварне измене фајла нису у конфликуту, чим занемаримо измене празног простора, све се спаја без проблема.

Ово је сламка спаса ако у свом тиму имате некога ко повремено воли да реформатира све из размака у табулаторе или обрнуто.

Ручно поновно спајање фајла

Мада програм Гит прилично добро одрађује препроцесирање празног простора, постоји неколико осталих врста измена које програм Гит можда не може аутоматски да обради, али могу да се реше скриптом. Као пример, хајде да се претворамо да програм Гит није могао да обради измену празног простора и да то морамо да обавимо ручно.

Уствари, оно што морамо да урадимо је да филтрирамо фајл који покушавамо да спојимо кроз `dos2unix` програм пре него што заиста извршимо спајање фајла. Па како то можемо да урадимо?

Најпре, треба да дођемо у стање конфликта при спајању. Затим желимо да имамо копије моје верзије фајла, њихове верзије (из гране коју спајамо) и заједничку верзију (одакле су обе гране потекле). Онда желимо да исправимо било њихову страну или нашу страну и покушамо поново да урадимо спајање, само за овај један фајл.

Добијање три верзије фајла је у суштини прилично лако. Програм Гит чува све ове верзије у индексу под „stages“ (етапе) и свака од њих има придружен број. Етапа 1 је заједнички предак, етапа 2 је ваша верзија и етапа 3 је из `MERGE_HEAD`, тј. верзија коју спајате („њихова“).

Командом `git show`, користећи специјалну синтаксу можете да издвојите копију сваког од ових верзија конфликтног фајла.

```
$ 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 желите још мало хард кора, можете такође да употребите и `ls-files -u` цевоводну команду којом добијате актуелне SHA-1 суме Git блобова за сваки од ових фајлова.

```
$ git ls-files -u  
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1  hello.rb  
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2  hello.rb  
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3  hello.rb
```

`:1:hello.rb` је једноставно скраћеница за проналажење SHA-1 тог блоба.

Сада када у радном директоријуму имамо садржај сваког од ова три фајла, можемо ручно да исправимо њихов тако да средимо проблем са празним простором, задамо поновно спајање фајла са не баш познатом `git merge-file` командом која ради управо то.

```
$ 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()
```

У овом тренутку имамо фино спојени фајл. Уствари, ово практично функционише и боље од опције `ignore-space-change` јер заиста исправља измене празног простора пре спајања, уместо да их једноставно игнорише. У `ignore-space-change` спајању, на крају имамо и неколико линија са ДОС завршетком, чиме је ствар измешана.

Ако желите да стекнете идеју шта је заиста промењено између једне или друге стране пре довршавања овог комита, можете питати `git diff` да упореди оно што је радном директоријуму и што ћете комитовати као резултат спајања са било којом од ових етапа. Хајде да прођемо кроз све.

Да бисте упоредили ваш резултат са оним што сте имали у својој грани пре спајања, другим речима, да видите шта је унело спајање, извршите `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()
```

Тако да овде лако можемо видети шта се дододило у нашој грани, да је измена те једне линије уствари оно што спајањем заиста уводимо у овај фајл.

Ако желимо да видимо како се резултат спајања разликује од онога што је на њиховој страни, можете извршити `git diff --theirs`. У овом и наредном примеру, морамо да употребимо `-b` да уклонимо празан простор јер поређење вршимо са оним што се налази у програму Гит, а не у нашем очишћеном `hello.theirs.rb` фајлу.

```
$ 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
```

Конечно, можете видети како се фајл променио са обе стране командом `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()
```

У овом тренутку можемо употребити команду `git clean` да уклонимо додатне фајлове које смо направили у циљу ручног спајања, јер више нису потребни.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Одјављивање конфликтата

Можда у овом тренутку из неког разлога нисмо задовољни решењем, или можда ручно уређивање једне или обе стране није дало резултат и потребно нам је још контекста.

Хајде да мало изменимо пример. У овом примеру, имамо две дуготрајније гране које обе

имају по неколико комита, али стварају прави конфликт садржаја када се споје.

```
$ 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
```

Сада имамо три јединствена комита који живе само на `master` грани и три друга која живе на `mundo` грани. Ако покушамо да спојимо `mundo` грану, добијамо 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.
```

Волели бисмо да видимо шта је конфликт при спајању. Ако отворимо фајл, видећемо нешто слично овоме:

```
#! /usr/bin/env ruby

def hello
<<<<< HEAD
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> mundo
end

hello()
```

Обе стране спајања су додале садржај у овај фајл, али су неки од комита изменили фајл на истом месту, што је и изазвало овај конфликт.

Хајде да истражимо неколико алата које имате на располагању за одређивање начина на који је дошло до овог конфликта. Можда није очигледно како би тачно требало да решите овај конфликт. Потребно вам је још контекста.

Један користан алат је `git checkout` са опцијом `--conflict'. Ово ће поново одјавити фајл и заменити маркере за конфликт спајања. Биће од користи ако желите да ресетујете маркере и покушате поново да их разрешите.

Опцији `--conflict` можете проследити или `diff3` или `merge` (што је подразумевана вредност).

Ако јој проследите `diff3`, програм Гит ће користити мало изменјену верзију маркера конфликта, који не приказују само „ours” и „theirs” верзије, већ и „base” верзију у линији, тако да вам пружа више контекста.

```
$ git checkout --conflict=diff3 hello.rb
```

Када ово извршимо, фајл ће изгледати овако:

```
#! /usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=====
  puts 'hello mundo'
>>>>> theirs
end

hello()
```

Ако вам се допада овај формат, можете поставити да буде подразумевани за све будуће конфликте при спајању тако што поставите подешавање `merge.conflictstyle` на `diff3`.

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

И `git checkout` команда такође може примити `--ours` и `--theirs` опције, што може бити заиста брз начин да се изабере једна или друга страна без икаквог спајања ствари.

Ово може бити посебно корисно за конфликте бинарних фајлова где једноставно можете изабрати једну страну, или где само желите да спојите одређене фајлове из неке друге гране - можете да обавите спајање па да онда одјавите одређене фајлове са једне или друге стране пре комитовања.

Лог спајања

Још један користан алат за решавање конфликта при спајању је `git log`. Ово може да вам помогне тако што пружа контекст онога што је можда допринело стварању конфликта. Понекада преглед мало историје може да буде од изузетне помоћи да запамтите зашто су две линије развоја утицале на исти део кода.

Да бисте добили потпуну листу свих јединствених комитова који су били део које гране укључене у ово спајање, можемо да употребимо синтаксу „три тачке” коју смо научили у [Трострука тачка](#).

```
$ 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'
```

То је фина листа од укупно шест комитова који су умешани, као и на којој грани развоја је био сваки од њих.

Мада ово можемо даље да упростимо тако да добијемо још одређенији контекст. Ако команди `git log` додамо опцију `--merge`, она ће приказати само комитове у било којој страни који утичу на фајл који је тренутно у конфликту.

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

Ако уместо овога то извршите са опцијом `-p`, добићете само разлике са фајлом који је завршио у конфликту. Ово **заниста** може бити од помоћи тако што вам брзо даје контекст који вам је потребан да разумете зашто је нешто у конфликту и како да интелигентније разрешите тај конфликт.

Формат комбиноване разлике

Пошто програм Гит стејџује све успешне резултате спајања, када извршите `git diff` док се налазите у стању спајања са конфлиktом, добијате само оно што је тренутно још увек у конфлиktу. То може бити од помоћи да видите шта још увек морате да разрешите.

Када `git diff` извршите директно након конфлиktа при спајању, она вам даје информације у прилично јединственом излазном формату разлике.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
 #! /usr/bin/env ruby

 def hello
++<<<<< HEAD
+ puts 'hola world'
+=====
+ puts 'hello mundo'
++>>>>> mundo
end

hello()
```

Формат се зове „комбинована разлика” („*Combined Diff*”) и уз сваку од линија вам приказује две колоне података. Прва колона вам приказује да ли се та линија разликује (додата је или уклоњена) између „ours” гране и фајла у радном директоријуму, а друга колона показује исто то само између „theirs” гране и копије у радном директоријуму.

Тако да у том примеру можете видети да су <<<<< и >>>>> линије у радној копији, али нису ни на једној страни спајања. Ово има смисла јер их је ту сместио алат за спајање како би нам приказао наш контекст, али се од нас очекује да их уклонимо.

Ако разрешимо конфлікт па поново извршимо `git diff`, видећемо исту ствар, али ипак мало корисније.

```
$ 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()
```

Ово нам показује да је „hola world” било на нашој страни али не и у радној копији, да је

„hello mundo” било на њиховој страни алине и у радној копији и на крају да „hola mundo” није било ни на једној страни али се сада налази у радној копији. Ово може бити корисно за преглед пре комитовања разрешења.

Можете да га добијете за било које спајање и од `git log` да видите како је нешто било разрешено након што се то уради. Програм Гит ће исписати овај формат ако извршите `git show` на комиту спајања, или ако команди `git log -p` додате опцију `--cc` (која подразумевано приказује само закрпе за комите који нису комити спајања).

```
$ 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

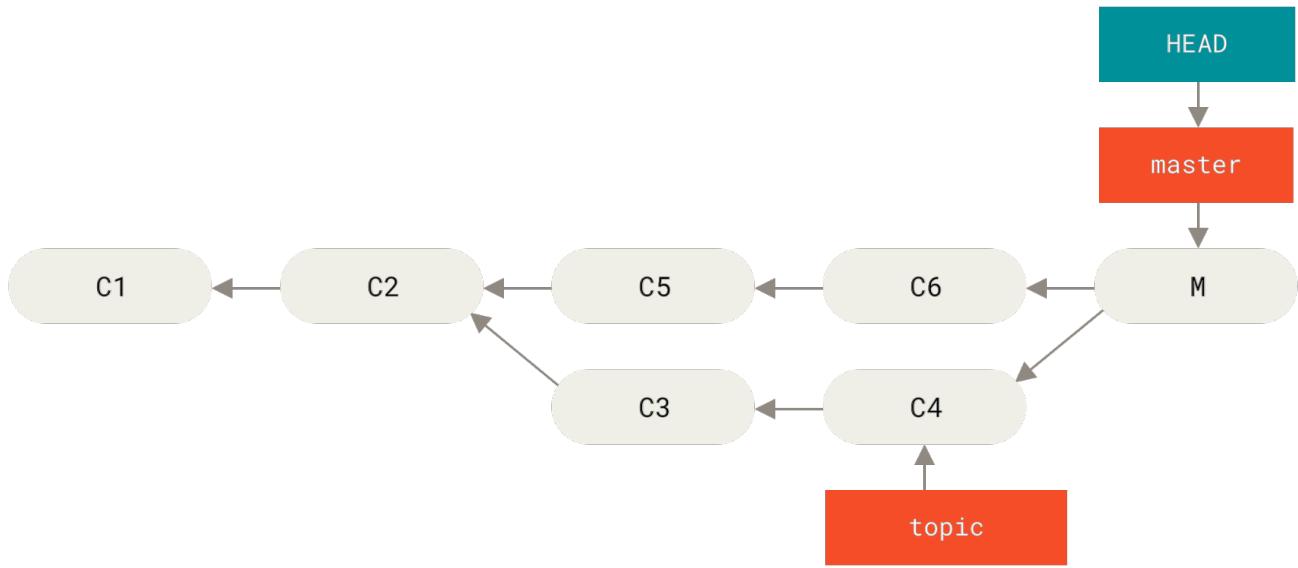
 def hello
-   puts 'hola mundo'
-   puts 'hello mundo'
++   puts 'hola mundo'
 end

hello()
```

Опозив спајања

Сада када знате како да направите комит спајања, вероватно ћете да направите и понеку грешку. Једна од сјајних ствари рада у програму Гит је да уопште није проблем правити грешке, јер могу (а у многим случајевима и једноставно) да се исправе.

Исти је случај и са комитовима спајања. Речимо да се започели рад на тематској грани, грешком је спојили у `master`, па сада историја комитова изгледа овако:

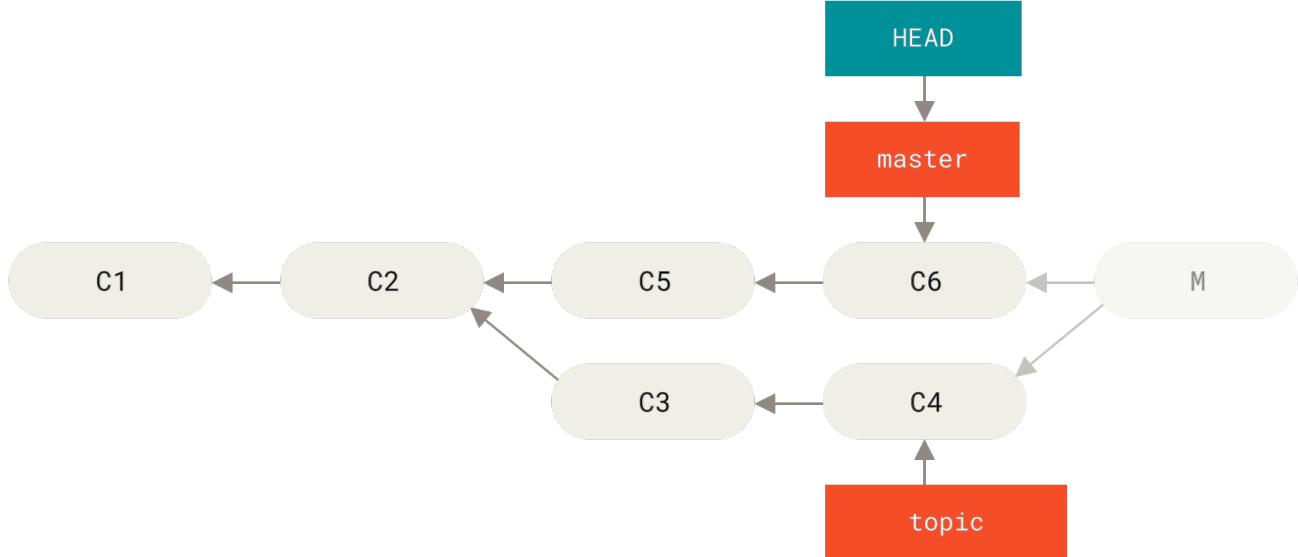


Слика 137. Случајни комит спајања

Постоје два начина да се приступи овом проблему, зависно од тога шта вам је жељени исход.

Исправљање референци

Ако нежељени комит спајања постоји само у вашем локалном репозиторијуму, најлакше и најбоље решење је да померите гране тако показују на жељено место. У већини случајева, ако након погрешног `git merge` извршите `git reset --hard HEAD~`, ресетоваћете показиваче грана тако да изгледају овако:



Слика 138. Историја након `git reset --hard HEAD~`

`reset` смо објаснили раније у [Демистификовани ресет](#), тако да не би требало да вам буде сувише тешко да схватите шта се овде догађа. Ево брзог подсетника: `reset --hard` обично пролази кроз три корака:

1. Померање гране на коју показује HEAD. У овом случају жељимо да се `master` помери на место пре комита спајања (`C6`).

2. Сређивање да индекс изгледа као HEAD.
3. Сређивање да радни директоријум изгледа као индекс.

Мана овог приступа је да поново исписује историју, што може бити проблем за дељени репозиторијум. Погледајте [Опасности ребазирања](#) за више о томе шта може да се догоди; укратко, ако други људи имају комитове које преписујете, врло вероватно би требало да избегнете `reset`. Овај приступ такође неће радити ако су након спајања креирани било који други комитови; померањем референци би се те промене ефективно изгубиле.

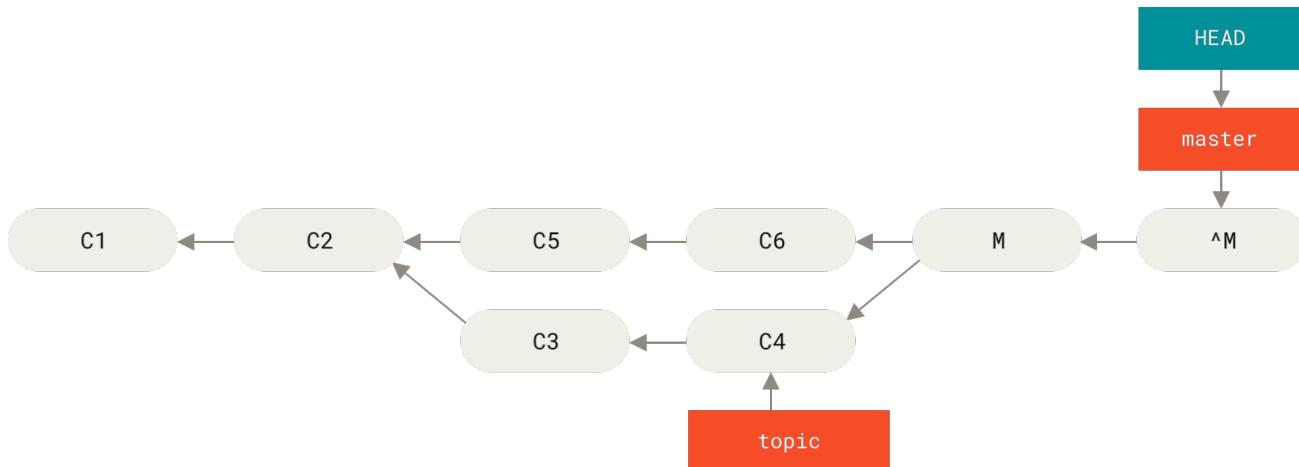
Враћање комита

Ако померање показивача грана неће радити у вашем случају, програм Git вам нуди опцију прављења новог комита који поништава све измене које је увео постојећи. Програм Git ову операцију назива „враћање“ („revert“) и у овом сценарију бисте је позвали на следећи начин:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

Заставица `-m 1` наводи који родитељ „главне линије“ би требало да се задржи. Када позовете спајање у `HEAD (git merge topic)`, нови комит има два родитеља: први је `HEAD (C6)`, а други је врх гране која се спаја (`C4`). У овом случају, желимо да опозовемо све измене уведене спајањем родитеља #2 (`C4`), а да истовремено задржимо сав садржај из родитеља #1 (`C6`).

Након враћања комита историја изгледа овако:



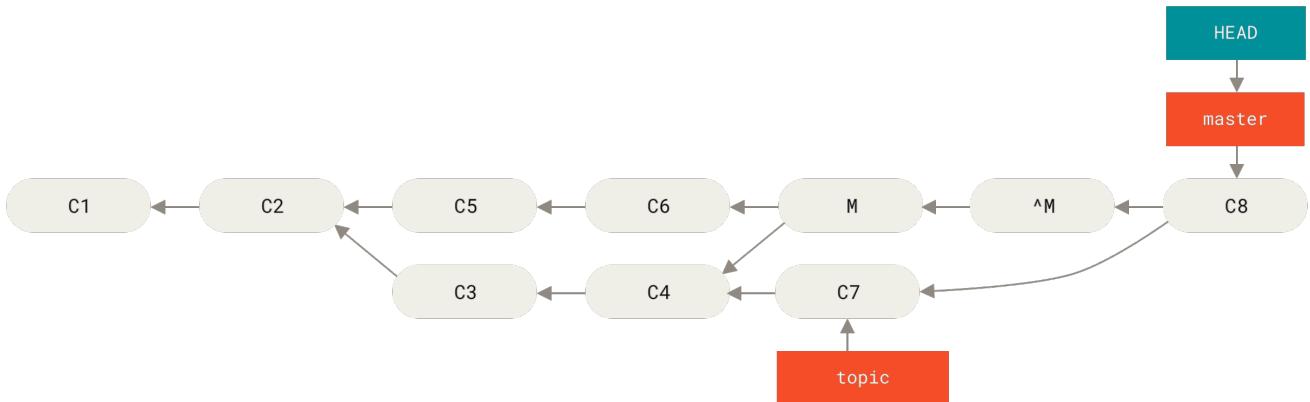
Слика 139. Историја након `git revert -m 1`

Нови комит `^M` има потпуно исти садржај као `C6`, тако да ако се почне одавде, исто је као да се спајање никада није ни додатило, осим што се комитови који нису спајање још увек налазе у историји `HEAD`.

Програм Гит ће се збуњити ако поново покушате да спојите `topic` у `master`:

```
$ git merge topic  
Already up-to-date.
```

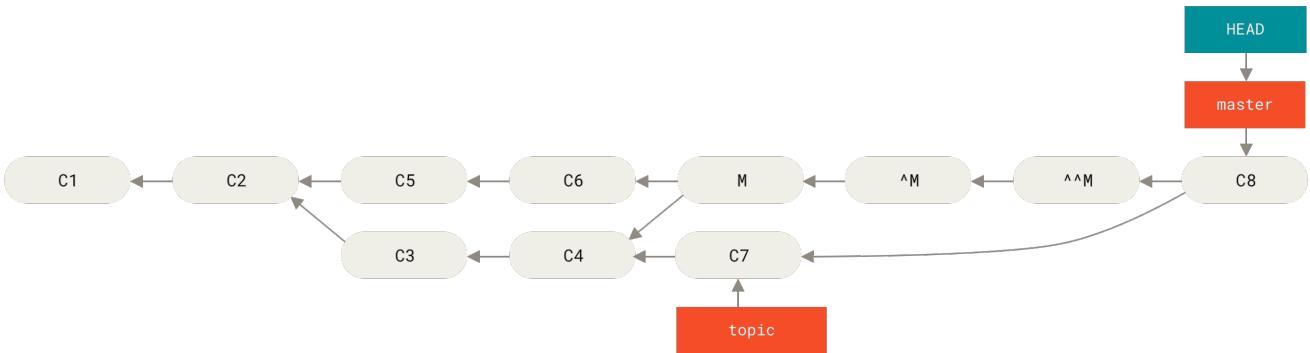
У `topic` грани не постоји ништа што већ није достијено из `master` грane. Још горе, ако додате рад у `topic` и поново урадите спајање, програм Git ће унети само измене настале *након* враћеног спајања:



Слика 140. Историја са лошим спајањем

Најбољи начин да се ово реши је да се поништи враћање оригиналног спајања, јер сада желите да уведете измене које су враћене, па **затим** креирате нови комит спајања:

```
$ git revert ^M  
[master 09f0126] Revert "Revert \"Merge branch 'topic'\""  
$ git merge topic
```



Слика 141. Историја након поновног спајања враћеног спајања

У овом примеру се `M` и `^M` поништавају. `^^M` ефективно спаја измене из `C3` и `C4`, а `C8` спаја измене из `C7`, тако да је `topic` сада у потпуности спојена.

Остале врсте спајања

До сада смо представили обично спајање две гране, које се углавном обрађује оним што се назива „рекурзивна” стратегија спајања. Међутим, постоје и други начини да се споје две гране. Хајде да брзо представимо неколико њих.

Предност нашег или њиховог

Пре свега, постоји још једна корисна ствар коју можемо урадити са обичним „рекурзивним” режимом спајања. Већ смо видели опције `ignore-all-space` и `ignore-space-change` које се прослеђују са `-X`, али програму Гит можемо такође навести да даје предност једној или другој страни када наиђе на конфликт.

Када програм Гит наиђе на конфликт између две гране које се спајају, он ће подразумевано да дода маркере конфликта у ваш кôд и маркираће фајл као конфликтни, па ће вама оставити да разрешите конфликт. Ако бисте желели да програм Гит просто изабере одређену страну и игнорише другу, уместо да од вас очекује да ручно решите конфликт, проследите команди `merge` било `-Xours` или `-Xtheirs`.

Ако програм Гит наиђе на ово, он неће да дода маркере конфликта. У случају било којих разлика које могу да се споје, спојиће. У случају било којих разлика које изазивају конфликт, просто ће изабрати страну коју сте навели у целини, укључујући и бинарне фајлове.

Ако се вратимо на „hello world” пример који смо раније користили, можемо видети да је спајање наше гране изазвало конфликте.

```
$ 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.
```

Међутим, ако га покренемо са `-Xours` или `-Xtheirs` конфликта нема.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 ++
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

У том случају, уместо да постави маркере конфликта у фајлу са „hello mundo” не једној страни и „hola world” на другој, просто ће изабрати „hola world”. Међутим, успешно се спајају све остале измене на тој грани које не праве конфликте.

Ова опција може да се проследи и команди `git merge-file` коју смо раније видели извршавајући нешто као што је `git merge-file --ours` за спајање појединачних фај洛va.

Ако желите да урадите овако нешто, али тако да програм Гит уопште ни не покуша да споји измене са друге стране, постоји драконска опција, под именом „ours” *стратегија* спајања. Ово се разликује од „ours” *опције* рекурзивног спајања.

Ово ће практично да уради лажно спајање. Забележиће нови комит спајања којем су обе

гране родитељи, али чак неће ни да погледа грану коју спајате. Само ће забележити да је резултат спајања потпуно исти код који је у вашој грани.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Видите да нема никаквих разлика између гране на којој смо се налазили и резултата спајања.

Ово често може бити корисно да се програм Гит у суштини завара тако да мисли да је грана већ спојена када касније буде радио спајање. На пример, рецимо да сте разгранали `release` грану и да сте на њој урадили неки посао који ћете у неком тренутку хтети да спојите назад у своју `master` грану. У међувремену, нека исправка бага у `master` грани треба да се портује назад у вашу `release` грану. Грану са исправљеним багом можете да спојите у `release` грану и да такође урадите `merge -s ours` исте гране у своју `master` грану (мада се исправка тамо већ налази), тако да када касније поново спојите `release` грану, неће бити конфликта услед исправке бага.

Спајање подстабла

Идеја код спајања подстабла је да имате два пројекта и један од пројеката се мапира у поддиректоријум другог и обрнуто. Када задате спајање подстабла, програм Гит је често довољно паметан да одреди ако је један подстабло оног другог, па да споји на погодан начин.

Проћи ћемо кроз пример додавања одвојеног пројекта у постојећи пројекат, па затим спајање кода другог у поддиректоријум првог.

На почетку, додајмо *Rack* апликацију нашем пројекту. *Rack* пројекат додајемо као удаљену референцу у пројекат, па је затим одјављујемо у њену сопствену грану:

```
$ 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"
```

Сада у `rack_branch` имамо корен *Rack* пројеката и наш пројекат у `master` грани. Ако одјавите један па онда други, видећете да им се корени пројекта разликују:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile    contrib      lib
COPYING      README        bin         example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Ово је прилично чудан концепт. Не морају све гране у вашем репозиторијуму да буду гране истог пројекта. Није уобичајено, јер је ретко од помоћи, али је прилично једноставно да имате гране које садрже потпуно различите историје.

У овом случају жељимо да повучемо *Rack* пројекат у наш `master` пројекат као поддиректоријум. У програму Гит то можемо да урадимо помоћу `git read-tree`. Научићете више о `read-tree` и његовим другарима у [Гит изнутра](#), али за сада је доволјно да знate да она учитава корено стабло једне од грана у ваш текући стејџ и радни директоријум. Управо смо се вратили у вашу `master` грану и повлачимо `rack_branch` грану у `rack` поддиректоријум наше `master` гране главног пројекта:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Када комитујемо, изгледа као да у том поддиректоријуму имамо све *Rack* фајлове – као да смо их прекопирали из *tarball* архиве. Оно што је интересантно је да прилично једноставно можемо да спојимо измене из једне гране у другу. Дакле, ако се *Rack* пројекат ажурира, узводне промене можемо да повучемо тако што се пребацимо на ту грану и повучемо:

```
$ git checkout rack_branch  
$ git pull
```

Те измене затим можемо да спојимо назад у нашу `master` грану. Да бисте повукли измене и унапред попунили комит подручку, употребите `--squash` опцију, као и `-Xsubtree` опцију рекурзивне стратегије спајања. Рекурзивна стратегија је овде и иначе подразумевана, али је због јаснијег приказа овде наводимо.

```
$ 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
```

Све измене из `Rack` пројекта су спојене и спремне за локално комитовање. Такође можете да урадите и супротно – направите измене у `rack` поддиректоријуму ваше `master` гране па их спојите у `rack_branch` да их касније предате одржаваоцима или да их турнете узводно.

Ово нам омогућава начин да имамо сличан процес рада као у случају подмодула само без потребе да се користе подмодули (које ћемо обрадити у [Подмодули](#)). У нашем репозиторијуму можемо да држимо гране са осталим повезаним пројектима и да их повремено спајамо као подстабло у наш пројекат. То је на неки начин фино, на пример сав кôд се комитује на једно место. Међутим, постоје и лоше стране јер је донекле комплексно и лакше је да се направе грешке приликом реинтеграције измена или нехотичног гурања гране у неповезани репозиторијум.

Још једна помало чудна ствар је начин на који добијате разлику између онога што се налази у `rack` поддиректоријуму и кода у `rack_branch` грани – како бисте видели да ли је потребно да их спојите – не можете употребити обичну `diff` команду. Уместо ње морате извршити `git diff-tree` са граном коју желите да упоредите:

```
$ git diff-tree -p rack_branch
```

Или, да бисте упоредили оно што се налази у `rack` поддиректоријуму са оним из `master` гране на серверу последњи пут кад сте преузели са њега, можете да извршите:

```
$ git diff-tree -p rack_remote/master
```

Rerere

`git rerere` функционалност је донекле скривена могућност. Име значи *reuse recorded resolution* (поново искористи сачувано решење) и као што име наводи, дозвољава вам да од програма Гит затражите да запамти начин на који сте разрешили комад конфликта, тако да кад следећи пут нађе на исти конфликт буде у могућности да га уместо вас аутоматски разреши.

Постоји већи број ситуација у којима је ова функционалност заиста корисна. Један од примера поменут у документацији је када хоћете обезбедити да се у будућности дуговечна тематска грана чисто споји, али не желите у међувремену гомилу комитова спајања. Када је `gегеге` укључено можете повремено да покренуте спајање, разрешите конфликте, па затим откажете спајање. Ако то радите редовно, онда би коначно спајање требало да прође глатко јер `gегеге` може аутоматски да одради све уместо вас.

Иста ова тактика може да се употреби ако грану желите да одржавате ребазираном тако да се не морате суочавати са истим конфликтима ребазирања сваки пут када то радите. Или ако имате грану коју сте спојили и исправили гомилу конфликата, па онда одлучите да уместо спајања урадите ребазирање – највероватније нећете морати да поново решавате све исте конфликте.

Још једна ситуација се дешава када повремено спајате гомилу тематских грана које се развијају у главу која може да се тестира, као што то често чини сам пројекат Гит. Ако тестови не прођу, можете да премотате уназад спајања и одрадите их поново без тематске гране због које тестови нису успешни, а да нема потребе за поновним решавањем конфликата.

Ако желите да укључите `gегеге` функционалност, једноставно извршите следеће конфигурационо подешавање:

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

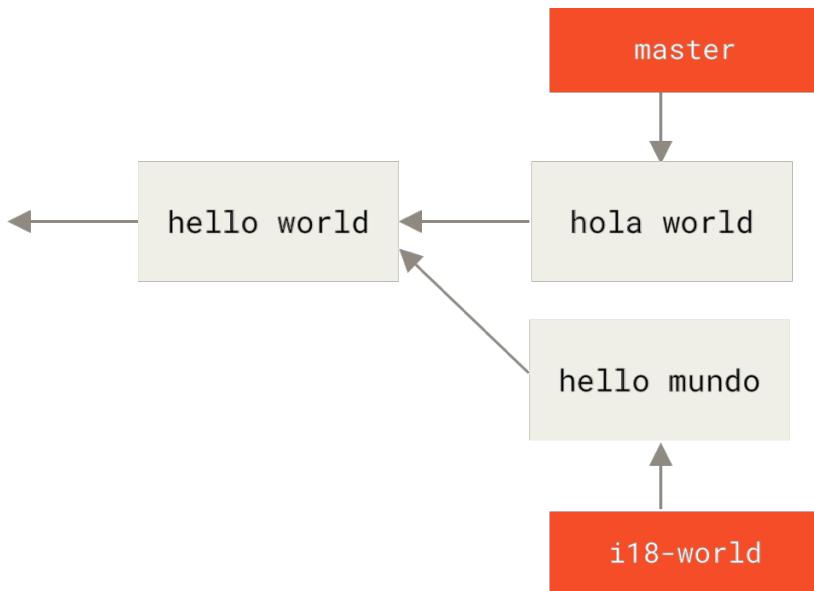
Други начин на који је takoђе можете укључити је да креирате `.git/гг-cache` директоријум у одређеном репозиторијуму, али је конфигурационо подешавање чистије и може да се обави глобално.

Хајде да сада погледамо прости пример, сличан претходном. Рецимо да имамо фајл који изгледа овако:

```
#! /usr/bin/env ruby

def hello
    puts 'hello world'
end
```

У једној грани изменимо реч „hello” у „hola”, па затим у другој грани променимо „world” у „mundo”, као и раније.



Када спојимо две гране, дођи ће до конфликта спајања:

```
$ 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.
```

Овде треба да приметите нову линију `Recorded preimage for` **ФАЈЛ**. Иначе би требало да изгледа као обичан конфликт спајања. У овом тренутку, `rerere` нам може рећи неколико ствари. Обично би сада извршили `git status` да видите шта је све у конфликту:

```
$ 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
#
```

Међутим, команда `git rerere` ће вам са `git rerere status` такође рећи и за које стање је сачувала стање пре спајања:

```
$ git rerere status
hello.rb
```

А `git rerere diff` ће приказати текуће стање решења – шта сте почели да решавате и на који начин сте га решили.

```
$ 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
```

Можете такође (и ово није у суштини у вези са `rerere`) употребити и `ls-files -u` да видите фајлове у конфликту и пре, леве и десне верзије:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1  hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2  hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3  hello.rb
```

Сада можете да га решите тако да једноставно буде `puts 'hola mundo'` и можете поново да извршите команду `rerere diff` да видите шта ће `rerere` запамтити:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env ruby

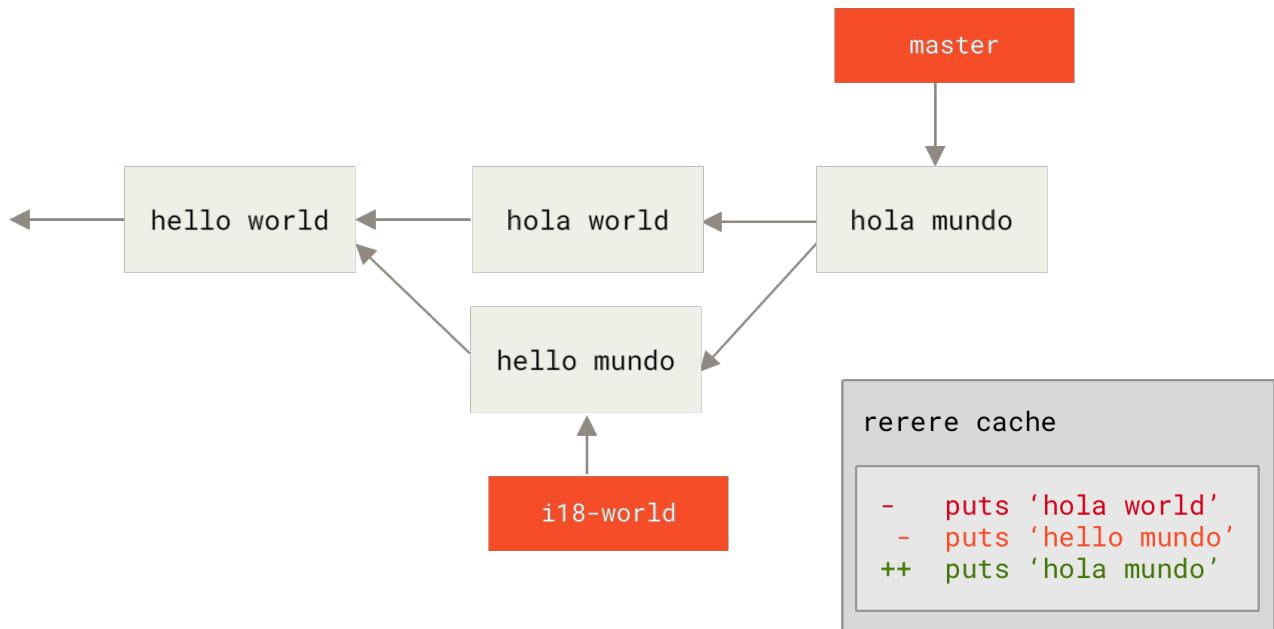
def hello
-<<<<<
- puts 'hello mundo'
-=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
end
```

Ово у суштини каже, када програм Гит нађе на комад конфликта у фајлу `hello.rb` који са једне стране има „hello mundo”, а „hola world” са друге, разрешиће се на „hola mundo”.

Конфликт сада можемо да обележимо као решен и да то комитујемо:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Видите да је „Сачувано решење за ФАЈЛ” (*Recorded resolution for ФАЈЛ*).



Хајде да сада поништимо то спајање па да уместо њега извршимо ребазирање на врх наше `master` гране. Грану можемо вратити уназад користећи `reset`, као што смо видели у [Демистификовани ресет](#).

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Поништили смо наше спајање. Хајде да сада ребазирамо тематску грану.

```
$ 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
```

Добили смо исти конфликт спајања, као што смо и очекивали, али погледајте линију **Resolved** **ФАЈЛ** **using previous resolution** (Решен ФАЈЛ употребом ранијег решења). Ако погледамо у фајл, видећемо да је већ разрешен, у њему нема маркера конфликта.

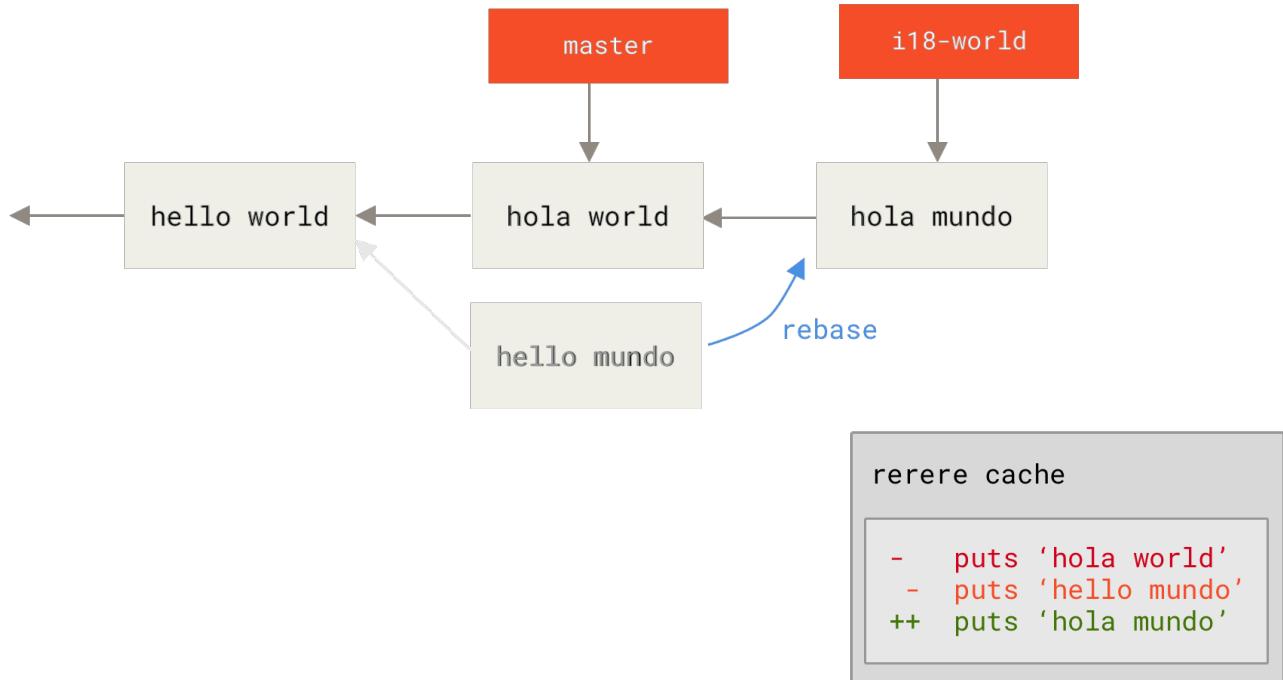
```
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

git diff ће вам такође показати како се аутоматски поново решио конфликт:

```
$ 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
```



Командом `checkout` такође можете и да поново вратите конфликтно стање фајла на следећи начин:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> theirs
end
```

Пример за ово смо видели у [Напредно спајање](#). Међутим, за сада, хаде да га још једном поново разрешимо употребом команде `regere`:

```
$ git regere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Поново смо аутоматски разрешили фајл употребом решења које је кеширала команда `regere`. Фајл сада можете да додате и наставити са ребазирањем до краја.

```
$ git add hello.rb  
$ git rebase --continue  
Applying: i18n one word
```

Дакле, ако имате много поновних спајања, или желите да тематску грану одржавате ажуарну са својом `master` граном без потребе за гомилом спајања, или ако често ребазирате, можете укључити `rebase` да вам макар мало олакша живот.

Отклањање грешака са програмом Git

Програм Гит такође обезбеђује и неколико алата који помажу при отклањању проблема у вашим пројектима. Пошто је програм Гит дизајниран тако да ради са скоро било којом врстом пројекта, ови алати су прилично општи, али вам често могу помоћи да уловите баг или кривца када ствари крену низбрдо.

Означавање фајла

Ако пратите баг у свом коду и желите да сазнате када је уведен и зашто, означавање фајла вам је најчешће најбољи алат. Ознаке показују који комит је последњи изменио сваку линију било ког фајла. Дакле, ако приметите да метода у коду има багове, фајл можете да означите напоменама командом `git blame` и да одредите који комит је одговоран за увођење те линије.

У следећем примеру се користи `git blame` да се одреди који комит и аутор је одговоран за линије у `Makefile` највишег нивоа Линукс кернела, а такође користи и опцију `-L` да ограничи излаз напомена на линије 69 до линије 82 тог фајла:

```
$ 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
```

Приметите да је прво поље део SHA-1 контролне суме комита који је последњи изменио ту линију. Наредна два поља су вредности издвојене из тог комита — име аутора и датум настанка тог комита – тако да лако можете видети ко је изменио ту линију и када. Након

тога долазе број линије и садржај фајла. Такође приметите `^1da177e4c3f4` комит линије, у којима префикс `^` означава линије које су уведене у репозиторијумов почетни комит и од тада су остале непромењене. Ово уноси поприличну забуну јер сте до сада видели барем три различита начина на које програм Гит користи `^` за измену SHA-1 комита, али то је значење овде.

Још једна одлична ствар у вези програма Гит је да он експлицитно не прати имена фајлова. Он чува снимке и онда покушава да одреди чему је имплицитно промењено име, након што се то већ дододило. Једна од интересантних могућности овога је да од програма можете тражити и да открије разноразна померања кода. Ако команди `git blame` проследите опцију `-C`, програм Git анализира фајл који означавате и покушава да открије одакле су дошли сегменти кода у случају да су били ископирани са неког другог места. На пример, рецимо да рефакторишете фајл под именом `GITServerHandler.m` у више фајлова, од којих је један `GITPackUpload.m`. Окривљујући (*blaming — оригинално значење команде, прим. прев.*) `GITPackUpload.m` уз опцију `-C`, видећете одакле су потекли делови кода:

```
$ 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)
```

Ово је веома корисно. Обично као оригинални комит добијате комит у којем сте кôд прекопирали, јер је то први пут да сте дотакли те линије кода у овом фајлу. Програм Гит вам наводи оригинални комит у којем сте уписали те линије, чак и ако је то било у неком другом фајлу.

Бинарна претрага

Означавање фајла помаже ако за почетак знате где се проблем налази. Ако не знате шта прави проблем, а било је десетине или стотине комитова од последњег стања за које знате да је кôд радио, највероватније ћете се обратити команди `git bisect` за помоћ. Команда `bisect` врши бинарну претрагу кроз историју ваших комитова како бисте што брже пронашли комит који је увео проблем.

Рецимо да сте управо објавили издање свог кода на производном окружењу, па добијате извештаје о багу који наводе да се нешто у производном окружењу не дешава, а ви не можете да замислите зашто се кôд тако понаша. Враћате се на свој кôд и испоставља се да можете поново изазвати проблем, али нема шансе да откријете шта не ваља. Да бисте

открили, можете да извршите бисекцију кода. Најпре извршите `git bisect start` да покренете ствари, па затим употребите `git bisect bad` да систему кажете да је комит на којем се тренутно налазите неисправан. Затим, помоћу `git bisect good [исправан_комит]` бисекцији морате навести последње познато исправно стање:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] Error handling on gero
```

Програм Git је открио да се око 12 комитова појавило између комита који сте означили као последњи исправни комит (v1.0) и тренутне неисправне верзије, па је уместо вас одјавио онај у средини. У овом тренутку можете да извршите своје тестове и откријете да ли проблем постоји до овог комита. Ако постоји, онда је уведен у неко време пре овог средишњег комита; ако не постоји, онда је проблем настао након овог средишњег комита. Испоставља се да овде нема проблема, а то програму Гит наводите куцајући `git bisect good`, па настављате своје путовање:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Secure this thing
```

Сада сте на још једном комиту, на пола пута између онога који сте тестирали и вашег неисправног комита. Поново покрећете свој тест и откривате да је овај комит неисправан, па то саопштавате програму Гит са `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Drop exceptions table
```

Овај комит је у реду, а програм Гит сада има све информације потребне да одреди место на којем је уведен проблем. Наводи вам SHA-1 суму првог неисправног комита и приказује неке од комит информација, као и фајлове који су изменјени у том комиту, тако да можете одредити шта се догодило и увело овај баг:

```
$ 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  
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

Када завршите, требало би да извршите `git bisect reset` чиме ресетујете HEAD на место на коме сте били пре почетка, или ћете завршити у чудном стању:

```
$ git bisect reset
```

Ово је моћан алат који вам може помоћи да у неколико минута проверите на стотине комитова и пронађете онај који је увео баг. Уствари, ако имате скрипту која ће вратити излазну вредност 0 ако је пројекат добар, или нешто различито од 0 ако није, `git bisect` можете потпуно аутоматизовати. Најпре, поново наводите опсег бисекције задајући познате неисправне и исправне комитове. То можете урадити наводећи их у команди `bisect start` ако желите, тако што прво иде познати неисправан комит, па затим познати исправан комит:

```
$ git bisect start HEAD v1.0  
$ git bisect run test-error.sh
```

Када ово покренете, `test-error.sh` се аутоматски извршава након сваког одјављеног комита све док програм Гит не пронађе први неисправан комит. Такође можете да извршите нешто као што је `make` или `make tests` или штагод имате да уместо вас покреће аутоматизоване тестове.

Подмодули

Док радите на једном пројекту, често имате потребу да унутар њега користите неки други пројекат. Можда је то библиотека коју је развио неко други или коју ви развијате одвојено користећи више родитељских пројеката. У овим случајевима долази до заједничког проблема: желите имати могућност да два пројекта третирате као одвојене, а да у исто време један од њих можете користити из другог.

Ево примера. Претпоставимо да развијате веб сајт и креирате Атом вести. Уместо да пишете свој сопствени код који генерише Атом, одлучујете да употребите библиотеку. Највероватније ћете морати или да укључите тај код из дељене библиотеке као што је CPAN `install` или Руби `gem`, или да у своје стабло пројекта копирате тај изворни код. Проблем са додавањем библиотеке представља тешкоћа у прилагођавању библиотеке на било који

начин, а често је још теже и да се библиотека достави јер морате обезбедити да библиотека буде доступна сваком клијенту. Проблем са копирањем кода у ваш пројекат је у компликованом спајању ваших локалних промена кода библиотеке са новим узводним променама.

Програм Гит решава овај проблем употребом подмодула. Подмодули вам омогућавају да Гит репозиторијум чувате као поддиректоријум неког другог Гит репозиторијума. На тај начин можете да клонирате други репозиторијум у свој пројекат и да ваше комитове држите раздвојене.

Први кораци са подмодулима

Прођи ћемо кроз развој једноставног пројекта који је подељен на неколико потпројеката.

Хајде да почнемо тако што је ћемо постојећи Гит репозиторијум додати као подмодул репозиторијума у којем радимо. Да бисте додали нови подмодул, користите `git submodule add` команду са апсолутном или релативном URL адресом пројекта који желите почети да пратите. У овом примеру ћемо додати библиотеку под називом „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.
```

Submodules ће подразумевано додати потпројекат у директоријум са истим именом као и репозиторијум, у овом случају „DbConnector”. Ако желите да се смести на неко друго место, на крај команде можете додати неку другу путању.

Ако у овом тренутку извршите `git status`, приметићете неколико ствари.

```
$ 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
```

Најпре треба да приметите нови фајл `.gitmodules`. Ово је конфигурациони фајл који чува мапирање између URL адресе пројекта и локалног директоријума у који сте га повукли:

```
$ cat .gitmodules
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

Ако имате више подмодула, постојаће више ставки у овом фајлу. Важно је приметите да је овај фајл део контроле верзије заједно са осталим фајловима, као што је ваш `.gitignore` фајл. Гура се и повлачи заједно са остатком вашег пројекта. На тај начин остали људи који клонирају овај пројекат знају одакле да преузму подмодул пројекте.



Пошто је URL у `.gitmodules` фајлу прво место одакле други људи покушају да клонирају/добаве, ако је то могуће, обезбедите да се користи URL адреса којој могу да приступе. На пример, ако користите различиту URL адресу на коју гурате од оне са које остали повлаче, користите ону којој остали могу да приступе. Ову вредност можете локално да препишете помоћу `git config submodule.DbConnector.url PRIVATE_URL` за вашу личну употребу. Када је могућа, релативна URL адреса може бити од помоћи.

Други листинг у излазу команде `git status` је ставка директоријума пројекта. Ако извршите `git diff` на њој, видећете нешто интересантно:

```
$ 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
```

Мада је `DbConnector` поддиректоријум у вашем радном директоријуму, програм Гит га види као подмодул и не прати његов садржај када се не налазите у том директоријуму. Уместо тога, Гит поддиректоријум види као одређени комит из тог репозиторијума.

Ако желите лепши приказ из команде `diff`, проследите опцију `--submodule` команди `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)
```

Када комитујете, видећете нешто овако:

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Уочите режим **160000** за ставку **DbConnector**. То је посебан режим у програму Гит који у суштини значи да комит бележите као директоријум, а не поддиректоријум или фајл.

Конечно, гурните ове измене:

```
$ git push origin master
```

Клонирање пројекта са подмодулима

Овде ћемо клонирати пројекат који садржи подмодул. Када клонирате један такав пројекат, подразумевано добијате директоријуме који садрже подмодуле, али ниједан од њих не садржи фајлове:

```
$ 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
$
```

Директоријум `DbConnector` је ту, али је празан. Морате извршити две команде: `git submodule init` да иницијализујете свој локални конфигурациони фајл и `git submodule update` да преузмете све податке из тог пројекта и одјавите одговарајући комит наведен у вашем суперпројекту:

```
$ 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'
```

Сада је ваш `DbConnector` поддиректоријум у потпуно истом стању као што је био када сте малопре комитовали.

Постоји и један мало једноставнији начин да се ово уради. Ако команди `git clone` наведете `--recurse-submodules`, она ће аутоматски да иницијализује и ажурира сваки подмодул у репозиторијуму, чак и угњеждене подмодуле у случају да неки од подмодула у репозиторијуму и сами поседују подмодуле.

```
$ 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 сте већ клонирали пројекат и заборавили да наведете `--recurse-submodules`, `git submodule init` и `git submodule update` кораке можете комбиновати извршавањем by running `git submodule update --init`. Ако такође желите и да иницијализујете, преузмете и одјавите све постојеће угњежђене подмодуле, искористите беспрекорну `git submodule update --init --recursive`.

Рад на пројекту са подмодулима

Сада имамо копију пројекта који у себи има подмодуле и сарађиваћемо са члановима нашег тима и на главном и на подмодул пројекту.

Повлачење узводних измена са удаљеног репозиторијума подмодула

Најједноставнији модел коришћења подмодула у пројекту је када једноставно употребљавате потпројекат и желите да с времена на време преузмете његова ажурирања, али ништа не мењате у својој одјављеној верзији пројекта. Хајде да прођемо кроз једноставни пример.

Ако желите проверити има ли новог рада у подмодулу, можете да одете у директоријум и извршите `git fetch` и `git merge` узводне гране чиме ажурирате локални код.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
  c3f01dc..d0354fc  master      -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Ако се сада вратите назад у главни пројекат и извршите `git diff --submodule` видећете да је подмодул ажуриран и исписаће вам се листа комитова који су му додати. Ако не желите да куцате `--submodule` сваки пут када извршавате `git diff`, то можете да подесите као подразумевани формат постављењем конфигурационе вредности `diff.submodule` на „log”.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Ако у овом тренутку комитујете, онда ћете подмодул закључати тако да садржи нови код онда када остали људи ажурирају.

Постоји такође и лакши начин да се ово уради, ако вам се не свиђа да поддиректоријум ручно преузмете и спојите. Ако извршите команду `git submodule update --remote`, програм Git ће прећи у ваш подмодуле и уместо вас урадити преузимање и ажурирање.

```
$ 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'
```

Ова команда ће подразумевано претпостављати да желите одјавити `master` грану подмодул репозиторијума. Међутим, ако желите, то можете поставити на нешто друго. На пример, ако желите да `DbConnector` подмодул прати „stable” грану тог репозиторијума, можете то да подесите или у свом `.gitmodules` фајлу (тако да је и сви остали прате), или само у свом локалном `.git/config` фајлу. Хајде да је поставимо у `.gitmodules` фајлу:

```
$ 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 'c87d55d4c6d4b05ee34fbcb6f7bf4585ae6687'
```

Ако изоставите `-f .gitmodules` измена ће важити само за вас, али вероватно има више смисла да се та информација прати у репозиторијуму, тако да сви остали то раде.

Када у овом тренутку извршимо `git status`, програм Гит ће нам приказати да имамо „new commits” (нове комитове) у подмодулу.

```
$ 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 подесите конфигурационо подешавање `status.submodulesummary`, програм Гит ће вам такође приказати и кратак резиме измена у вашим подмодулима:

```
$ 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
```

Ако у овом тренутку извршимо `git diff` видећемо и да нам је изменјен `.gitmodules` фајл и да такође постоји већи број комитова које смо повукли и који су спремни за комит у нашем подмодул пројекту.

```
$ 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
```

Ово је заиста одлично јер можемо да видимо лог комитова које ћемо управо комитовати у наш подмодул. Након комитовања, ту информацију можете видети када извршите `git log -p`.

```
$ git log -p --submodule
commit 0a24cf121a8a3c118e0105ae4ae4c00281cf7ae
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
```

Када извршите `git submodule update --remote`, програм Гит ће подразумевано покушати да ажурира **све** ваше подмодуле. Ако их имате доста, вероватно ћете пожелети да проследите име оног подмодула за који желите да се покуша ажурирање.

Повлачење узводних измена са удаљеног репозиторијума пројекта

Хајде да сада ускочимо у ципеле вашег сарадника који има сопствени локални клон репозиторијума *MainProject*. Просто извршавање `git pull` за преузимање свеже комитованих измена није довољно:

```
$ 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")
```

Команда `git pull` подразумевано рекурзивно преузима измене подмодула, као што видимо изнад у излазу прве команде. Међутим, она не **ажурира** подмодуле. Ово се види у излазу `git status` команде који приказује да је подмодул „modified” (измењен) и да има „new commits” (нове комитове). Уз то, заграде говоре да нови комитови показују улево (<), што значи да су ти комитови забележени у *MainProject* али нису присутни у локалном *DbConnector* одјављивању. Да бисте довршили ажурирање, морате да извршите `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
```

Да бисте били сигурни, приметите да би `git submodule update` требало да извршите са заставицом `--init` у случају да су нови *MainProject* комитови које сте управо повукли додали нове подмодуле и са `--recursive` заставицом за случај да неки од подмодула имају угњежђено подмодуле.

Ако овај процес желите да аутоматизујете, команди `git pull` (почевши од Гит верзије 2.14) можете да додате заставицу `--recurse-submodules`. Ово ће наложити програму Гит да изврши `git submodule update` непосредно након повлачења, постављајући подмодуле у исправно стање. Уз то, ако желите да програм Гит увек повлачи са `--recurse-submodules`, можете да поставите конфигурациону опцију `submodule.recurse` на `true` (ово функционише за `git pull` почевши од верзије 2.15 програма Гит). Ова опција ће навести програм Гит да заставицу `--recurse-submodules` употребљава за све команде које је подржавају (осим `clone`).

Постоји специјална ситуација која може да се догоди када се повлаче ажурирања суперпроекта: може се десити да је у једном од комитова које повлачите узводни репозиторијум променио URL адресу подмодула у `.gitmodules` фајлу. Ово може да се деси, на пример, ако This can happen for example if the submodule project changes its hosting platform. У том случају је могуће да `git pull --recurse-submodules` или `git submodule update` не успеју да се изврше ако суперпројекат указује на комит подмодула који не може да се нађе у удаљеном репозиторијуму подмодула локално конфигурисаном у вашем репозиторијуму. Да би се решила ова ситуација, потребна је команда `git submodule sync`:

```
# копира нову URL адресу у вашу локалну конфигурацију
$ git submodule sync --recursive
# ажурира подмодул са нове URL адресе
$ git submodule update --init --recursive
```

Рад на подмодулу

Врло је вероватно да ако користите подмодуле, то чините јер заиста желите да радите на коду у подмодулу истовремено уз рад на коду главног пројекта (или преко неколико подмодула). У супротном бисте сигурно користили једноставнији систем за управљање зависностима (као што је *Maven* или *Rubygems*).

Па хаде да сада прођемо кроз пример прављења измена у подмодулу истовремено са изменама у главном пројекту и комитовање и објављивање тих измена у исто време.

До сада, кадгод смо извршавали команду `git submodule update` да преузмемо измене из

репозиторијума подмодула, програм Гит би преузео измене и ажурирао фајлове у поддиректоријуму, али би под-репозиторијум оставио у такозваном „detached HEAD” (одвојен HEAD) стању. То значи да нема локалне радне гране (као што је `master`, на пример) која прати измене. Када нема радне гране која прати измене, чак и ако комитујете измене у подмодулу, оне ће највероватније изгубити када следећи пут извршите `git submodule update`. Морате урадите неке додатне кораке ако желите да се измене у подмодулу прате.

Ако подмодул желите подесити тако да буде лакше да само уђете у њега и почнете да дељите, морате урадити две ствари. Потребно је да уђете у сваки подмодул и одјавите грану у којој ћете радити. Затим програму Гит морате рећи шта да ради ако сте направили измене, а `git submodule update --remote` он да повуче нови рад са узводног репозиторијума. Имате две опције: можете да их спојите у свој локални рад, или можете покушати да ребазирате свој локални рад на врх нових измена.

Најпре, хајде да одемо у директоријум подмодула и одјавимо грану.

```
$ cd DbConnector/
$ git checkout stable
Switched to branch 'stable'
```

Покушајмо да „merge” опцијом ажурирамо наш подмодул. Ручно је наводимо тако што `update` позиву једноставно додамо опцију `--merge`. Овде ћемо видети да је на серверу дошло до измене за овај подмодул и она се спаја.

```
$ 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 пређемо у *DbConnector* директоријум, видећемо да су нове измене већ спојене у нашу локалну `stable` грану. Хајде сада да видимо шта се дешава када направимо локалну измену библиотеке па неко други у исти време узводно другу промену.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
 1 file changed, 1 insertion(+)
```

Аса сада ажурирамо свој подмодул видећемо шта се дешава у случају када смо направили локалну измену, а постоји и узводна измена коју морамо да уведемо.

```
$ 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 заборавите да наведете `--rebase` или `--merge`, програм Гит ће једноставно ажурирати подмодул на оно што се налази на серверу и ресетоваће ваш пројекат на стање одвојен HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Ако се то догоди, не брините, једноставно се можете вратити назад у директоријум и поново одјавити своју грану (која још увек садржи ваш рад) и ручно спојити или ребазирати `origin/stable` (или коју год удаљену грану желите).

Ако своје измене подмодула још увек нисте комитовали, па покренете ажурирање подмодула, имаћете проблема, програм Гит ће преузети измене али неће преписати несачуван рад у вашем директоријуму подмодула.

```
$ 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'
```

Ако сте направили измене које су у конфликту са нечим што се променило узводно, програм Гит ће вас обавестити о томе када покренете ажурирање.

```
$ 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 отићи у директоријум подмодула и исправити конфликте, као што бисте и иначе урадили.

Објављивање измена подмодула

Сада имамо неке измене у нашем директоријуму подмодула. Неке од њих су дошле узводно кроз наше ажурирање, а остале су урађене локално и још увек никоме нису доступне јер их нисмо објавили.

```
$ 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 комитујемо у главном пројекту и гурнемо га узводно, а да не гурнемо и измене у подмодулу, остали људи који покушају да одјаве наше измене биће у проблему јер неће имати начина да дођу до измена подмодула од којих зависе. Те измене ће постојати само у нашој локалној копији.

Ако желите обезбедити да до овога не дође, можете затражити од програма Гит да провери да ли су сви важни подмодули исправно турнути пре него што гурне главни пројекат. Команда `git push` узима `--recurse-submodules` аргумент који може да се постави било на „check” или на „on-demand”. Опција „check” ће учинити да `push` једноставно не успе са извршавањем ако било које од комитованих измена подмодула нису још увек турнуте.

```
$ 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.

Као што видите, команда вам приказује и корисне савете у вези могућих наредних корака. Једноставна могућност је да пређете у сваки подмодул и да ручно турнете на удаљене репозиторијуме и тако обезбедите да су доступни споља када поново покушате турње главног пројекта. Ако желите да се ово понашање провере дешава за сва турња, поставите га да буде подразумевано са `git config push.recurseSubmodules check`.

Друга могућност је да се користи вредност „on-demand“ која ће ово покушати да уради уместо 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
```

Као што видите, програм Гит је прешао у *DbConnector* модул у турнуо га пре него што је турнуо главни пројекат. Ако из неког разлога то турње не успе, неће успети ни турње главног пројекта. Ово понашање можете поставити као подразумевано извршавањем `git config push.recurseSubmodules on-demand`.

Спајање измена подмодула

Ако измените референцу на подмодул у исто време кад и неко други, можете наићи на проблеме. То јест, ако су се историје подмодула разишле и комитоване у одвојене гране суперпроекта, биће потребно мало рада да то поправите.

Ако је један од комитова директни предак другог (спајање методом брзог премотавања унапред), онда ће програм Гит за спајање једноставно изабрати тај други и то лепо функционише.

Међутим, програм Гит уместо вас неће покушати чак ни тривијално спајање. Ако се комитови подмодула разилазе и потребно је да се споје, видећете нешто слично овоме:

```
$ 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.
```

Оно што се у суштини десило овде је да је програм Гит открио да све гране бележе тачке у историји подмодула које се разилазе и потребно је да се споје. Он то објашњава као „merge following commits not found” (није пронађено спајање након комитова), што донекле збуњује, па ћемо ускоро објаснити зашто је тако.

Да бисте решили проблем, потребно је да одредите стање у којем подмодул треба да се налази. Чудно, али програм Гит вам у овој ситуација не пружа доста информација које могу да вам помогну, чак ни SHA-1 суме комитова са обе стране историје. Не срећу, ово једноставно може да се одреди. Ако извршите `git diff` видећете SHA-1s суме комитова који су забележени у обе гране које покушавате да спојите.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Дакле, у овом случају је `eb41d76` комит у нашем подмодулу који **ми** имамо, а `c771610` је комит који долази узводно. Ако пређемо у директоријум подмодула, већ би требало да се налази на `eb41d76` јер спајање није утицало на њега. Ако из било ког разлога није тако, једноставно можете да креирате и одјавите грану која показује на њега.

Важна је SHA-1 сума комита са друге стране. То је оно што треба да спојите и да разрешите. Можете просто да пробате спајање директно са SHA-1, или можете креирати грану за њега, па покушати њу да спојите. Ми предлажемо ово друго, ако ништа друго, да бисмо могли написати лепшу поруку за комит спајања.

Дакле, прећи ћемо у директоријум подмодула, направићемо грану базирану на том другом SHA-1 из излаза комаде `git diff` и ручно ћемо да је спојимо.

```
$ 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.
```

Овде је дошло до конфликта при спајању, тако да ако га разрешимо и комитујемо, онда једноставно можемо да ажурирамо главни пројекат тим резултатом.

```
$ 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 Прво разрешимо конфликт.

② Затим се вратимо у директоријум главног пројекта.

③ Можемо поново да проверимо SHA-1 суме.

④ Означимо да је ставка подмодула у конфлику решена.

⑤ Комитујемо спајање.

Можда помало збуњује, али заиста није тешко.

Интересантно је да постоји још један случај који програм Гит сам обрађује. Ако у директоријуму подмодула постоји комит спајања који у својој историји садржи **оба** комита, програм Гит ће вам то предложити као могуће решење. Он види да је у неком тренутку подмодул пројекта неко спојио гране које садрже ова два комита, па је могуће да ћете пожелети тај.

Ово је разлог што се приказује порука „merge following commits not found”, јер није био у стању да уради **ово**. Забуну уноси питање да ли би било ко и очекивао да програм Гит ово покуша?

Ако пронађе макар један прихватљиви комит спајања, видећете нешто слично овоме:

```
$ 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.
```

Команда коју програм Гит предлаже ће ажурирати индекс исто као да сте извршили `git add` (што означава да је конфликт разрешен), па затим комитује. Мада то вероватно не би требало да урадите. Исто тако можете лако да одете у директоријум подмодула, погледате шта је разлика, брзо премотате унапред на овај комит, тестирате га како треба, па га онда комитујете.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forward to a common submodule child'
```

Ово постиже исти резултат, али барем можете проверити да функционише и када завршите

имате код у директоријуму свог подмодула.

Савети за подмодуле

Постоји неколико ствари које могу да вам олакшају рад са подмодулима.

Submodule Foreach

Постоји **foreach** подмодул команда која служи за извршавање произвољне комаде у сваком од подмодула. Ово може дosta да помогне ако у истом пројекту имате већи број подмодула.

На пример, рецимо да желимо почети рад на новој могућности или да исправимо баг, а рад се одвија у више подмодула. Лако можемо да сакријемо комплетан рад у свим нашим подмодулима.

```
$ 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
```

Затим можемо да креирамо нову грану и пређемо на њу у свим нашим подмодулима.

```
$ git submodule foreach 'git checkout -b featureA'  
Entering 'CryptoLibrary'  
Switched to a new branch 'featureA'  
Entering 'DbConnector'  
Switched to a new branch 'featureA'
```

Схватате поенту. Једна заиста корисна ствар коју можете да урадите је да направите лепу уједињену разлику онога што је промењено у главном пројекту, као и у свим потпројектима.

```

$ 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_page_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;

```

Овде можемо видети да дефинишемо функцију у подмодулу и позивамо је из главног пројекта. Ово је очигледно упрошћен пример, али надамо се да вам даје представу колико може бити корисно.

Корисни алијаси

Вероватно ћете хтети да поставите неколико алијаса за неке од ових команда, јер могу бити прилично дугачке и за већину не можете поставити подразумеване опције конфигурацијом. Постављање алијаса у програму Гит смо објаснили у covered setting up Git aliases in [Гит алијаси](#), али овде дајемо пример шта можете поставити ако планирате доста да радите са подмодулима у програму Гит.

```

$ git config alias.sdiff '!"git diff && git submodule foreach \'git diff\'"
$ git config alias.push 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'

```

На овај начин једноставно можете извршити `git submodule update` кадогод желите да ажурирате своје подмодуле, или `git submodule push` да гурнете подмодул уз проверу зависности.

Проблеми са подмодулима

Међутим, употреба подмодула није тако глатка.

Пребацивање грана

На пример, пребацивање са гране на грани која у себи има подмодуле може бити компликовано. Ако креирате нову грани, тамо додате подмодул, па се вратите назад на грани без подмодула, још увек ћете имати директоријум подмодула као непраћен директоријум:

```
$ 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)
```

Уклањање директоријума није тако тешко, али збуњује то што га уопште имате тамо. Ако га уклоните, па се онда вратите назад на грани која има тај подмодул, мораћете да извршите `submodule update --init` да бисте га поново попунили.

```
$ 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
```

Да поновимо, није толико компликовано, али помало збуњује.

Новије верзије програма Гит (Git >= 2.13) све ово поједностављују додавањем заставице **--recurse-submodules** команди `git checkout` која брине о постављању подмодула у стање које одговара грани на коју прелазимо.

```
$ 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
```

Употреба заставице **--recurse-submodules** команде `git checkout` такође може бити корисно када радите на неколико грана суперпроекта, а свака од њих има ваше подмодуле који показују на различите комитове. Заиста, ако пређете са грани на грани која не бележи

подмодул на истом комиту, након извршавања команде `git status` подмодул ће се појавити као „modified” (измењен) и означаваће „new commits” (нови комитови). Разлог за ово је што се стање подмодула подразумевано не преноси приликом преласка на другу грану.

Ово заиста може уносити забуну, тако да је добра идеја да увек извршавате `git checkout --recurse-submodules` када ваш пројекат има подмодуле. У старијим верзијама програма Гит које немају заставицу `--recurse-submodules`, након одјављивања можете употребити `git submodule update --init --recursive` и тако поставите подмодуле у одговарајуће стање.

Срећом, програму Гит ($>=2.14$) можете конфигурационом опцијом `submodule.recurse` навести да увек корсити заставицу `--recurse-submodules`: `git config submodule.recurse true`. Као што је напоменуто изнад, то ће навести програм Гит да рекурзивно посети подмодуле за свку команду која у себи има наведену опцију `--recurse-submodules` (осим `git clone`).

Претварање поддиректоријума у подмодуле

Друго главно ограничење на које многи људи наиђу се тиче претварања поддиректоријума у подмодуле. Ако сте у свом пројекту пратили фајлове и желите да их померите у подмодул, морате бити опрезни иначе ће програм Гит да се наљути на вас. Претпоставимо да имате фајлове у поддиректоријуму свог пројекта и да желите да га претворите у подмодул. Ако обришете поддиректоријум па онда извршите `submodule add`, програм Гит вам каже следеће:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Директоријум `CryptoLibrary` најпре морате да уклоните са стејца. Након тога можете да додате подмодул:

```
$ 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.
```

Претпоставимо сада да сте то урадили у грани. Ако покушате да се вратите на грани у којој су ти фајлови још увек у стаблу, а не у подмодулу, враћа вам се следеће 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 принудно да пређете са `checkout -f`, али будите пажљиви да тамо немате несачуване измене јер би ова команда могла да их препише.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Затим, када се вратите назад, имате празан `CryptoLibrary` директоријум и можда га ни `git submodule update` не поправи. Потребно је да се вратите и директоријум подмодула и извршите `git checkout .` да вратите назад све своје фајлове. Ово бисте могли да извршите у `submodule foreach` скрипти ако имате више подмодула.

Важно је приметите да у данашње време подмодули чувају све своје Гит податке у `.git` директоријуму основног пројекта, за разлико од много старијих верзија програма Гит, уништавање директоријума подмодула нећете изгубити ниједан комит или грану коју сте имали.

Уз ове алате, подмодули могу да буду прилично једноставан и ефективан метод за истовремено развијање неколико повезаних, али ипак одвојених пројеката.

Паковање

Мада смо представили уобичајене начине за пренос Гит података преко мреже (HTTP, SSH, итд.), постоји уствари још један начин за то који се не користи тако често, али ипак може бити прилично користан.

Програм Git је способан да своје податке „пакује” у један једини фајл. То може да се искористи у разним ситуацијама. Можда вам мрежа не ради, а промене морате да пошаљете својим сарадницима. Можда радите негде ван радног места и из разлога безбедности немате приступ локалној мрежи. Можда се управо покварила ваша бежична/етернет картица. Можда тренутно немате приступ дељеном серверу, неком хоћете да пошаљете ажурирања имејлом, али не желите да преносите 40 комитова путем `format-patch`.

У оваквим ситуацијама вам у помоћ стиже `git bundle` команда. Команда `bundle` ће запаковати све што би се иначе гурнуло кроз жицу `git push` комадом у бинарни фајл који некоме можете да пошаљете имејлом или да га ставите на флеш драјв, који затим распакујете у други репозиторијум.

Хајде да погледамо једноставан пример. Рецимо да имате репозиторијум са два комита:

```
$ 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 некоме желите да пошаљете тај репозиторијум, а немате право гурања на циљни репозиторијум, или једноставно не желите да га подесите, можете да га запакујете са [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)
```

Сада имате фајл под именом [repo.bundle](#) који садржи све податке потребне да се поново креира [master](#) грана репозиторијума. У команди [bundle](#) морате да наведете сваку референцу или одређени опсег комитова који желите да буду део пакета. Ако имате намеру да се ово клонира негде другде, требало би да додате и [HEAD](#) као референцу, као што смо ми овде урадили.

Овај [repo.bundle](#) фајл можете да пошаљете имејлом неком другом, или да га прекопирате на USB драјв и однесете га.

С друге стране, рецимо да је вами послат овај [repo.bundle](#) фајл и да желите да радите на пројекту. Можете да клонирате из бинарног фајла у директоријум, исто као што бисте и са URL адресе.

```
$ git clone repo.bundle repo  
Cloning into 'repo'...  
...  
$ cd repo  
$ git log --oneline  
9a466c5 Second commit  
b1ec324 First commit
```

Ако HEAD не укључите у референце, морате такође да наведете `-b master` или која год грана да је укључена у фајл, јер у супротном програм неће знати коју грну да одјави.

Рецимо да сада направите три комита на њој и нове комитове желите да пошаљете назад пакетом на USB драјву или имејлом.

```
$ git log --oneline  
71b84da Last commit - second гепо  
c99cf5b Fourth commit - second гепо  
7011d3d Third commit - second гепо  
9a466c5 Second commit  
b1ec324 First commit
```

Најпре морамо да одредимо опсег комитова које желимо да запакујемо. За разлику од мрежних протокола који уместо нас одређују минимални скуп података који треба пренети преко мреже, ово ћемо морати ручно да одредимо. Истина да бисте могли једноставно да урадите исту ствар и запакујете комплетан репозиторијум, што би радило, али је боље да запакујете само разлике - само три комита које смо управо креирали локално.

Да бисте то урадили, мораћете да израчунате разлику. Као што смо описали у [Опсези комитова](#), опсег комитова можете навести на разне начине. Да бисте добили три комита које имамо у нашој мастер грани, а који нису били у грани коју смо првобитно клонирали, можемо употребити нешто као што је `origin/master..master` или `master ^origin/master`. То можете да тестирате `log` командом.

```
$ git log --oneline master ^origin/master  
71b84da Last commit - second гепо  
c99cf5b Fourth commit - second гепо  
7011d3d Third commit - second гепо
```

Дакле, сада када имамо листу комитова које желимо да запакујемо, хајде да то и урадимо. Команди `git bundle create` треба да наведете име фајла пакета и опсег комитова који желите да запакујете.

```
$ 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)
```

Сада у нашем директоријуму имамо фајл `commits.bundle`. Ако га пошаљемо партнери, она може да га увезе у оригинални репозиторијум, чак и ако је тамо у међувремену урађено још посла.

Када прими пакет, она може да га испита и сазна шта се у њему налази пре него што га увезе у свој репозиторијум. Прва команда је `bundle verify` команда која ће проверити да ли је

фајл исправан Гит пакет и да имате све неопходне претке потребне да се исправно састави.

```
$ 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
```

Да је особа која је направила пакет запаковала само два последња комита која је направила, уместо сва три, оригинални репозиторијум не би био у стању да га увезе, јер му недостаје потребна историја. Тада би `verify` команда изгледала овако:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit - second repo
```

Међутим, наш први пакет је исправан, па из њега можемо да преузмемо комитове. Ако желите да видите гране у пакету које могу да се увезу, постоји и команда која приказује само главе:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Поткоманда `verify` ће вам такође приказати главе. Поента је да се види шта може да се повуче, тако да можете употребити `fetch` или `pull` команду и увезете комитове из овог пакета. Овде ћемо преузети 'master' грану пакета у грану под именом 'other-master' у нашем репозиторијуму:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master      -> other-master
```

Сада можемо видети да увезене комитове имамо на 'other-master' грани, као и евентуалне комитове које смо у међувремену направили на својој 'master' грани.

```
$ 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
```

Дакле, `git bundle` може бити веома корисна за дељење или обављање мрежних операција када немате праву мрежу или дељени репозиторијум који вам то омогућавају.

Замена

Објекти програма Гит не могу да се промене, али он обезбеђује интересантан начин претварања да у својој бази података замењује са другим објектима.

Команда `replace` вам омогућава да наведете објекат у програму Гит и кажете „сваки пут се обратиш овом објекту, претварај се да је то неки други објекат”. Ово је најчешће корисно да се један комит у вашој историји замени неким другим, без потребе да са рецимо `git filter-branch` морате да поново изградите комплетну историју.

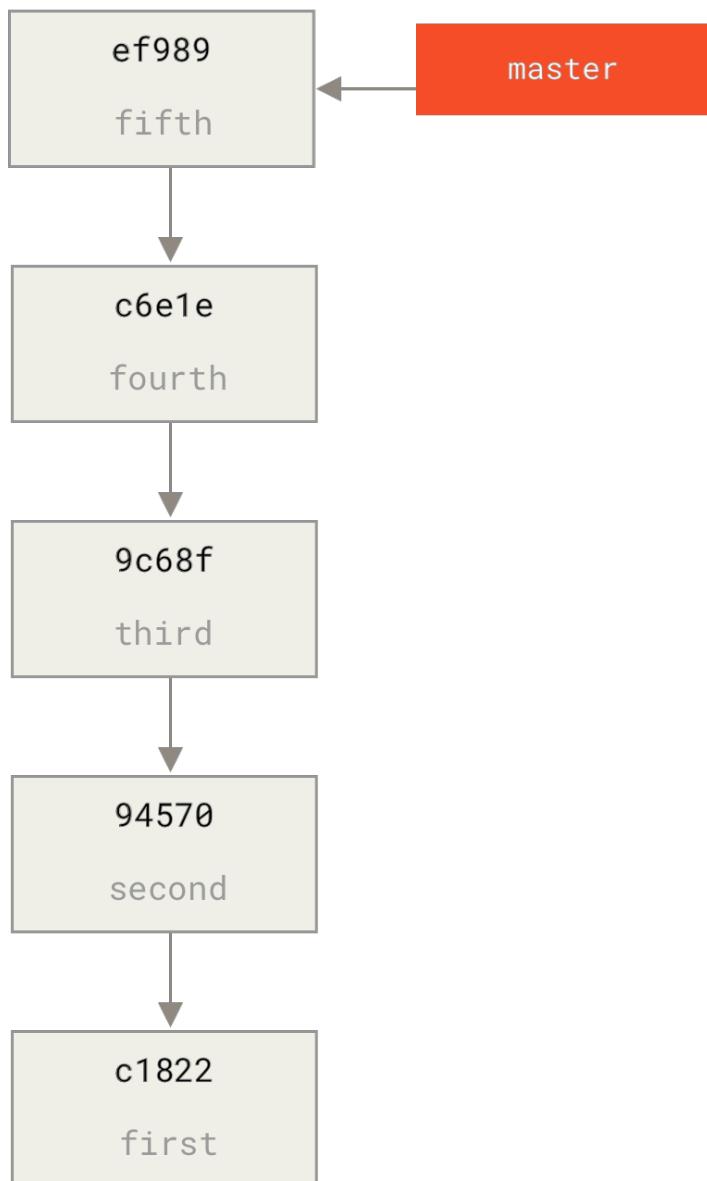
На пример, рецимо да имате огромну историју кода и желите да поделите свој репозиторијум у једну кратку историју за нове програмере и једну много дужу и већу историју за људе које интересује рударење података. Једну историју можете да накалемите на другу „замењивањем” најранијег комита у новој линији са последњим комитом у старијој. Ово је лепо јер значи да нема потребе да заиста поново испишете сваки комит у новој историји, као што би то морали ако бисте их спојили заједно (јер родитељство утиче на SHA-1 суме).

Хајде да ово испробамо. Узмимо постојећи репозиторијум, поделимо га у два репозиторијума, један скрашњи и један историјски, па ћемо видети како можемо да их рекомбинујемо без потребе за изменом SHA-1 вредности скрашњих репозиторијума помоћу `replace`.

Користићемо једноставан репозиторијум са пет простих комитова:

```
$ git log --oneline
ef989d8 Fifth commit
cbe1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Ово желимо да поделимо у две линије историје. Једна линија иде од комита један до комита четири - то ће бити историјска. Друга линија ће представљати само комитове четири и пет - то ће бити скрашња историја.

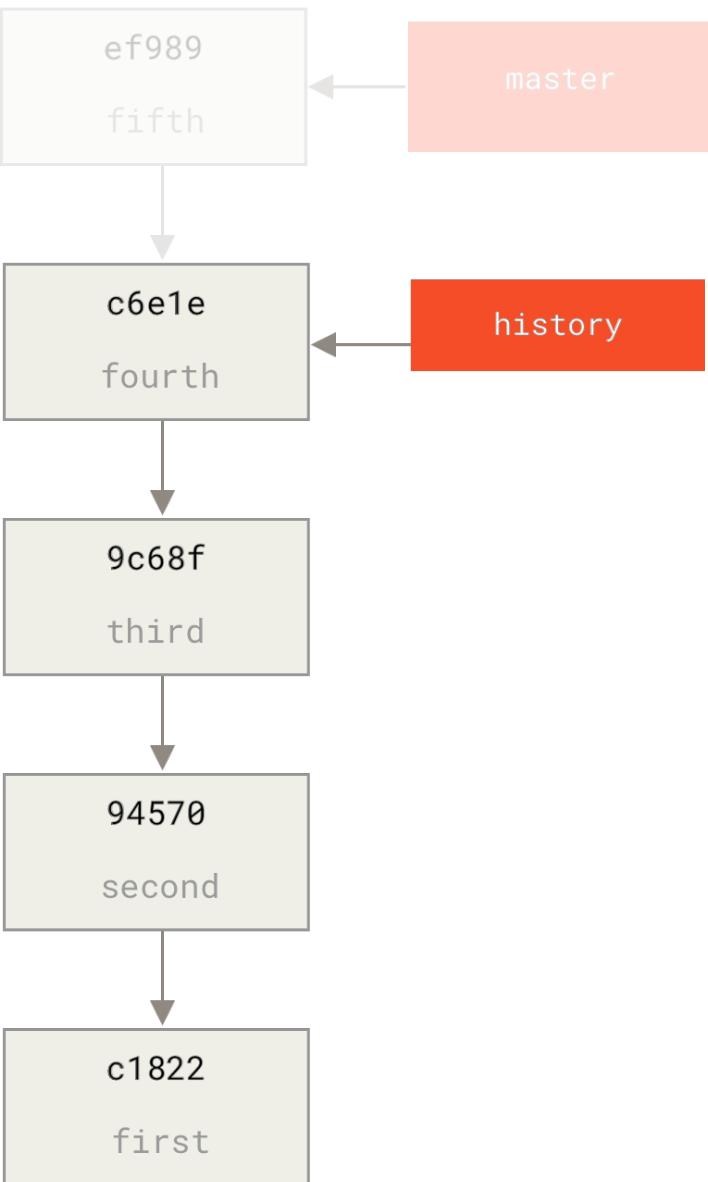


Дакле, креирање историје је једноставно, потребно је само да у историју поставимо грану, па да је онда турнемо на `master` грану новог удаљеног репозиторијума.

```

$ 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

```



Сада нову **history** грану можемо да гурнемо на **master** грану нашег новог репозиторијума:

```

$ 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

```

ОК, објавили смо своју историју. Сада долази тежи део одсецања скрашње историје, тако да

постане мања. Потребно нам је преклапање тако да комит у једној заменимо еквивалентним комитом у другој, тако да ћемо ово скратити само на комитове четири и пет (дакле, преклапа се комит четири).

```
$ git log --oneline --decorate  
ef989d8 (HEAD, master) Fifth commit  
cbe1e95 (history) Fourth commit  
9c68fdc Third commit  
945704c Second commit  
c1822cf First commit
```

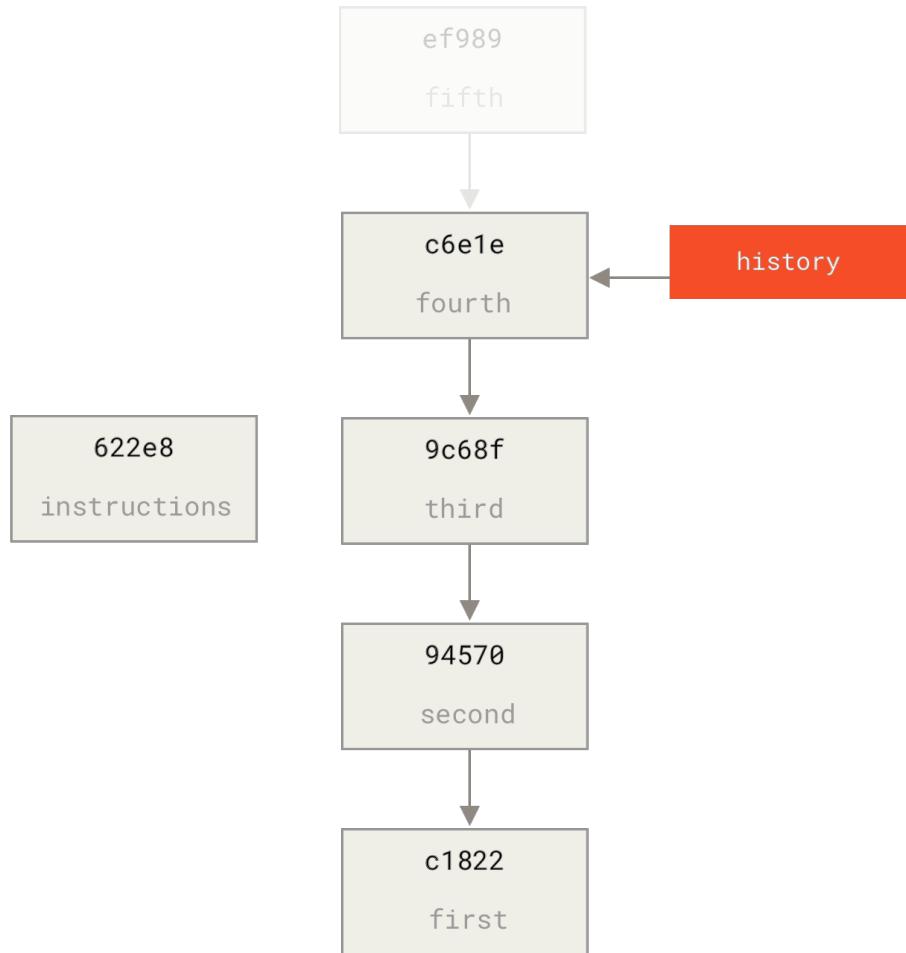
У овом случају је корисно да се креира базни комит који садржи упутства како да се историја прошири, тако да остали програмери знају шта да раде ако дођу до првог комита у скраћеној историји, а потребно им је још. Дакле, оно што ћемо урадити је креирати објекат почетног комита као базну тачку са упутствима, па да затим ребазирамо преостале комитове (четири и пет) преко њега.

Да би то постигли, морамо да изаберемо тачку поделе, што је у нашем случају трећи комит, `9c68fdc` у SHA говору. Значи, наш базни комит ће се базирати од тог трећег. Базни комит можемо креирати употребом `commit-tree` команде која једноставно узима стабло и враћа нам SHA-1 потпуно новог комит објекта који нема родитеља.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}  
622e88e9cbfbacfb75b5279245b9fb38dfa10cf
```

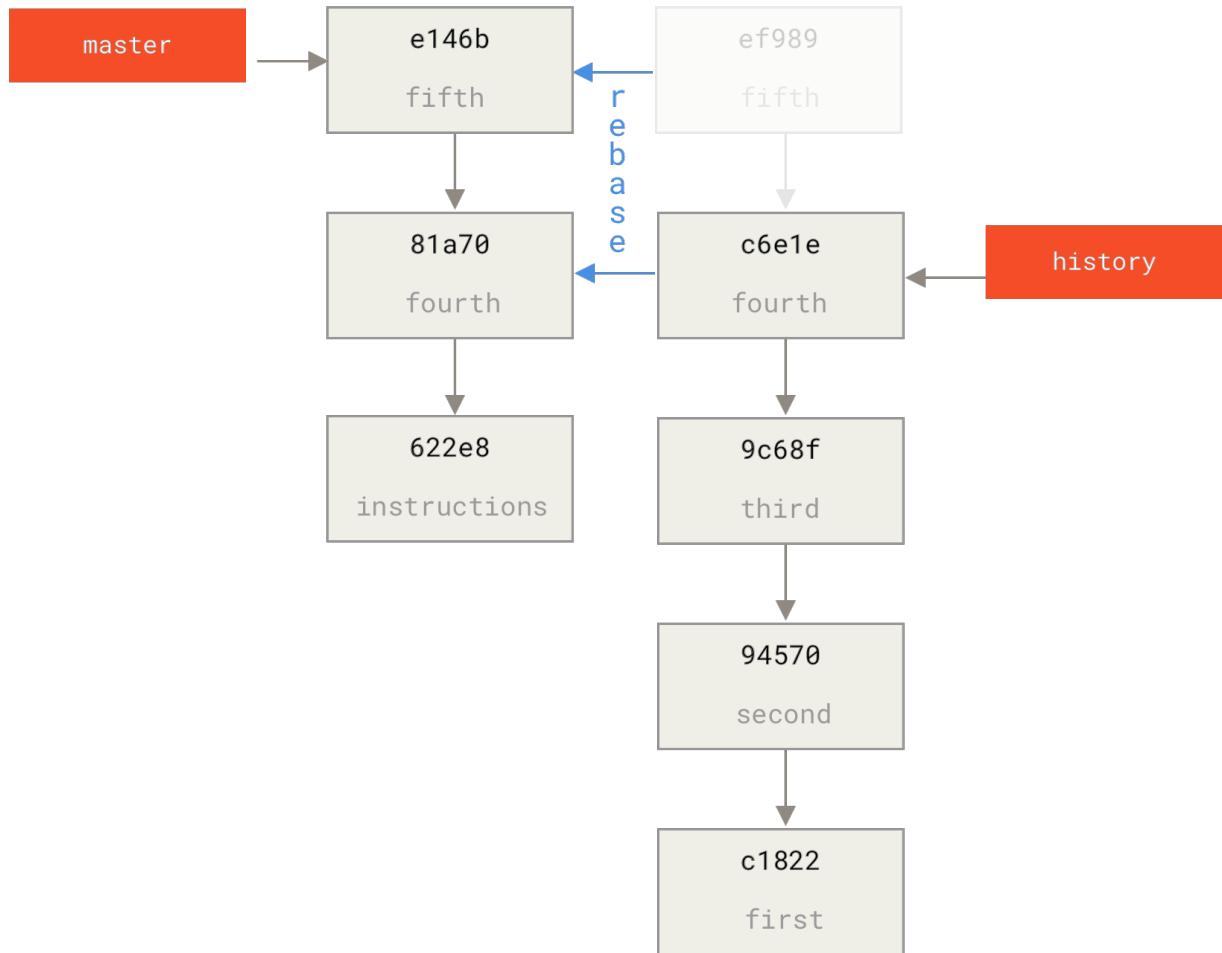
Команда `commit-tree` је једна од из скупа команди које се често називају 'водоводне' (*plumbing*) команде. Оне у општем случају нису намењене за директно извршавање, већ их **остале** Гит команде користе да за њих обаве неке мање послове. У случајевима када радимо неке чудније ствари као што је ова, те команде нам омогућавају да одрадимо ствари заиста ниског нивоа, али нису предвиђене за свакодневну употребу. Више о водоводним командама можете прочитати у [Водовод и порцелан](#)





OK, сада када имамо базни комит, можемо да ребазирамо остатак наше историје преко њега са `git rebase --onto`. Аргумент `--onto` ће бити SHA-1 који смо управо добило од команде `commit-tree` и тачка ребазирања ће бити трећи комит (родитељ првог комита који желимо да задржимо, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: Fourth commit
Applying: Fifth commit
```



OK, сада смо поново исписали нашу скрашњу историју преко базног комита за одбацивање који сада у себи има упутство како да се у случају потребе реконституише комплетна историја. Ту нову историју можемо да гурнемо у нови пројекат, па када онда људи клонирају тај репозиторијум, видеће само последња два комита, а онда и базни комит са упутством.

Хајде да сада заменимо улоге и постанемо неко ко по први пут клонира пројекат и жели комплетну историју. Да би добио податке о историји након клонирања овог скраћеног репозиторијума, он би требало да дода један удаљени репозиторијум за онај који чува комплетну историју и да преузме одатле:

```
$ 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
```

Сарадник би сада у `master` грани имао последње комитове, а у `project-history/master` грани историјске комитове.

```
$ 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
```

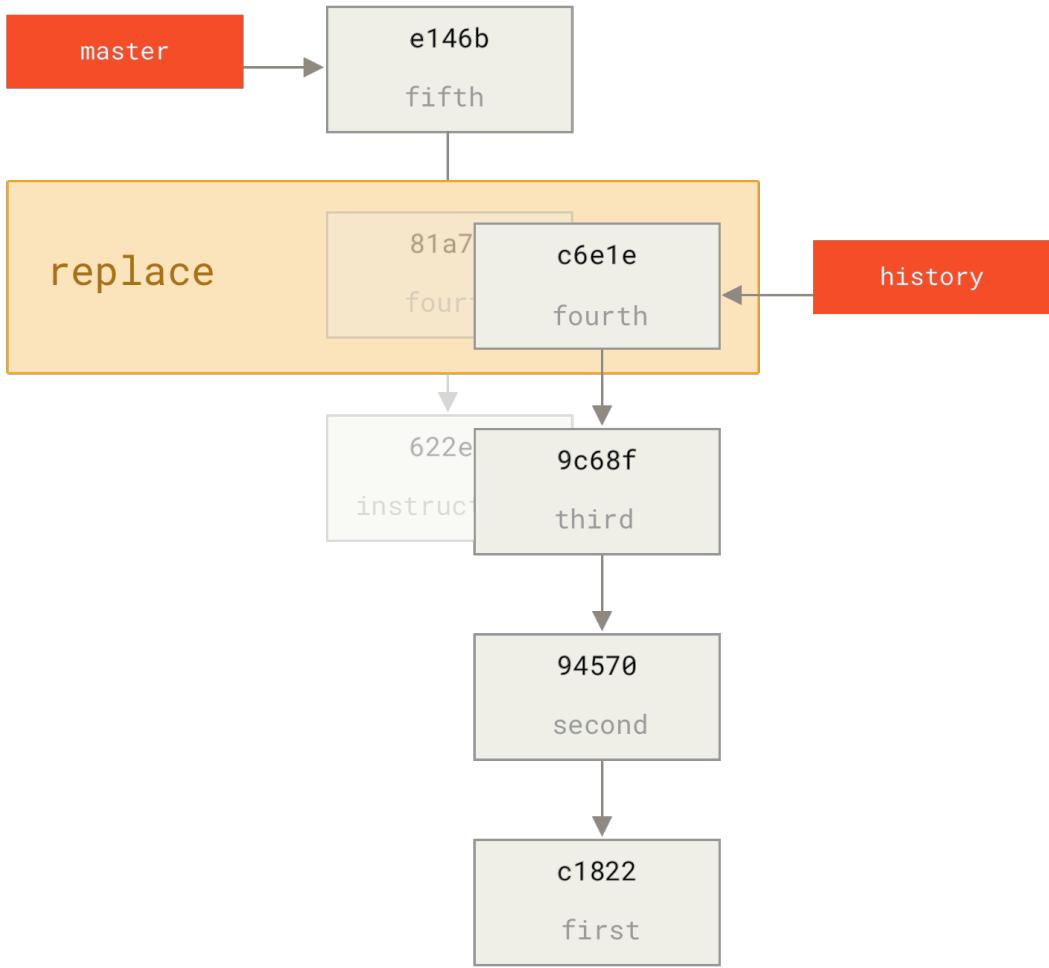
Ако желите да их комбинујете, можете једноставно да позовете `git replace` са комитом који желите замените, па са комитом којим желите да га замените. Тако да „четврти” комит у `master` грани желимо да заменимо са „четвртим” комитом у `project-history/master` грани:

```
$ git replace 81a708d c6e1e95
```

Ако сада погледате историју `master` грane, изгледаће овако:

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Фино, зар не? Били смо у могућности да заменимо један комит у нашој историји потпуно другим комитом, без потребе да мењамо све SHA-1 контролне суме узводно, а сви уобичајени алати (`bisect`, `blame`, итд.) ће радити онако како се и очекује да раде.



Интересантно је да се као SHA-1 сума још увек приказује **81a708d**, мада се уствари користе подаци **c6e1e95** комита којим смо га заменили. Чак и ако извршите команду као што је **cat-file**, она ће вам приказати замењене податке:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eeee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Упамтите да је стварни родитељ комита **81a708d** наш комит чувар места (**622e88e**), а не **9c68fdce** као што овде пише.

Још једна интересантна ствар је да се ови подаци чувају у нашим референцама:

```
$ 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
```

То значи да је дељење наше замене са осталима једноставно јер ово можемо да гурнемо на наш сервер и остали то лако могу да преузму. Ово није од велике помоћи у сценарију калемљења историје који смо овде представили (пошто би се у сваком случају преузимале обе историје, па зашто да се онда раздвајају?), али може бити корисно у неким другим околностима.

Складиште акредитива

Ако за повезивање на удаљене репозиторијуме користите SSH транспорт, могуће је да имате кључ без вишеделне лозинке који вам омогућава да безбедно преносите податке без потребе уноса корисничког имена и лозинке. Међутим, ово није могуће са HTTP протоколима – за сваку везу је неопходно корисничко име и лозинка. Ово постаје још теже за системе са двофакторском аутентификацијом код којих се токен коришћен за лозинку генерише насумично и не може да се изговори.

На срећу, програм Гит поседује систем акредитива који вам помаже у оваквим ситуацијама. Програм Гит нуди неколико утврђених опција:

- Акредитиви се подразумевано не кеширају. Приликом сваког повезивања бићете упитани за своје корисничко име и лозинку.
- „cache” режим чува акредитиве у меморији на одређени период. Ниједна лозинка се никада не чува на диск и уклањају се из кеша након 15 минута.
- „store” режим чува акредитиве у фајлу на диск као чисти текст и никада не истичу. Ово значи да докле год не промените своју лозинку за Гит хост, више нећете морати поново да уносите своје акредитиве. Мана овог приступа је што се нешифриране лозинке чувају у чистом фајлу који се налази у вашем почетном директоријуму.
- Ако користите Мек, програм Гит долази са „osxkeychain” режимом који акредититве кешира у сигурном привеску за кључеве (*keychain*) који је везан са ваш системски налог. Ова метода чува акредитиве на диск и они никада не истичу, али су шифровани истим системом који чува HTTPS сертификате и аутоматска попуњавања за програм Сафари.
- Ако користите Виндоуз, мекОС, или Линукс, можете да инсталирате помоћник под називом „[Git Credential Manager](#)”. Он за контролу осетљивих информација користи складишта података развијена за дату платформу.

Неку од ових метода можете изабрати тако што поставите Гит конфигурациону вредност:

```
$ git config --global credential.helper cache
```

Неки од ових помоћника имају опције. „store” помоћник може да прихвати `--file <путања>` аргумент који прилагајава место чувања фајла са чистим текстом (подразумевана вредност је `~/.git-credentials`). „cache” помоћник прихвата опцију `--timeout <секунди>` која мења време током којег даемон наставља са извршавањем (подразумевана вредност је „900”, или 15 минута). Ево примера како да за „store” помоћник подсетите жељено име фајла:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Програм Гит вам чак дозвољава да подесите и неколико помоћника. Када тражи акредитиве за одређени хост, програм Гит ће питати помоћнике редом и зауставиће се када прими први одговор. Када чува акредитиве, програм Гит ће послати корисничко име и лозинку **свим** помоћницима са листе, па они могу одлучити шта да раде са тим подацима. Ево како би изгледао `.gitconfig` ако имате фајл са акредитивима на флеш драјву, али желите да користите кеш у меморији како не бисте морали да куцате када драјв није у порту:

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 3000
```

Испод хаубе

Како све ово функционише? Основна команда програма Гит за систем помоћника за акредитиве је `git credential` која узима команду као аргумент, па затим још улазних података кроз `stdin`.

Ово се вероватно лакше схвата кроз пример. Речимо да је помоћник за акредитиве конфигурисан и да је помоћник сачувао акредитиве за `mygithost`. Следи сесија која користи команду „fill”, која се позива када програм Гит покушава да пронађе акредитиве за неки 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 Ово је команда која започиње интеракцију.
- ② Git-credential затим чека на улаз са *stdin*. Прослеђујемо му ствари које знамо: протокол и име хоста.
- ③ Празна линија означава да је унос завршен и да би систем акредитива требало да одговори оним што зна.
- ④ Затим git-credential преузима и исписује на *stdout* делиће информације коју је пронашао.
- ⑤ Ако се акредитиви не пронађу, програм Гит тражи од корисника да унесе корисничко име и лозинку, па и г доставља назад позивајућем *stdout* (овде су они прикачени на исту конзолу).

Систем акредитива уствари позива програм који није део самог програма Гит; који је то програм, зависи од вредности конфигурационе променљиве `credential.helper`. Она може узети неколико облика:

| Вредност конф. променљиве | Понашање |
|--|--|
| <code>foo</code> | Покреће <code>git-credential-foo</code> |
| <code>foo -a --opt=bcd</code> | Покреће <code>git-credential-foo -a --opt=bcd</code> |
| <code>/absolute/path/foo -xyz</code> | Покреће <code>/absolute/path/foo -xyz</code> |
| <code>!f() { echo "password=s3cre7"; }; f</code> | Код након <code>!</code> се израчунава у љуски |

Тако да се помоћници описани изнад уствари називају `git-credential-cache`, `git-credential-store` и тако даље, па можемо да их конфигуришемо тако да узимају аргументе из команде. Општи облик за ово је `git-credential-foo [арг] <акција>`. *stdin/stdout* протокол је исти као за `git-credential`, само што користе донекле изменјени скуп акција:

- `get` је захтев за пар корисничко_име/лозинка.
- `store` је захтев за чување скупа акредитива у меморији помоћника.

- `erase` чисти акредитиве за дате особине из меморије помоћника.

Одговор није обавезан за `store` и `erase` акције (програм Гит га ионако игнорише). Међутим, у случају `get` акције, програм Гит је веома заинтересован да чује шта помоћник има да каже. Ако помоћник не зна било шта корисно, он једноставно може да заврши извршавање без икаквог излаза, али ако зна, требало би да достављену информацију допуни са сачуваним подацима. Излаз се третира као низ наредби доделе; све што се достави ће заменити оно што програм Гит већ зна.

Ево ситог примера као малопре, само што прескачамо `git-credential` и прелазимо право на `git-credential-store`:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3crg7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3crg7
```

① Овим кажемо помоћнику `git-credential-store` за сачува неке акредитиве: корисничко име „`bob`” и лозинка „`s3crg7`” би требало да се употребе када се приступа <https://mygithost>.

② Сада ћемо преузети те акредитиве. Достављамо делове везе које већ познајемо (<https://mygithost>), и празну линију.

③ `git-credential-store` одговара са корисничким именом и лозинком коју смо сачували изнад.

Ево како изгледа фајл `~/git.store`:

```
https://bob:s3crg7@mygithost
```

Он је само низ линија од којих свака садржи URL украшен акредитивима. Помоћници `osxkeychain` и `winstore` користе нативни формат својих складишта, док `cache` користи свој сопствени формат у меморији (који ниједан други процес не може да прочита).

Прилагођени кеш акредитива

Како су `git-credential-store` и пријатељи програми одвојени од програма Гит, не треба пуно вештине да се схвати како *било који* програм може бити Гит помоћник за акредитиве. Помоћници које обезбеђује програм Гит покривају многе уобичајене случајеве употребе, али не све. На пример, рецимо да ваш тима има неке акредитиве које дели цео тим, вероватно за постављање. Они се чувају у дељеном директоријуму, али не желите да их копирате у своје складиште акредитива јер се често мењају. Овај случај употребе не покрива ниједан од

постојећих помоћника; па хајде да видимо шта је потребно да напишете сопствени. Постоји неколико кључних могућности које овај програм мора да поседује:

1. Једина акција на коју треба да обратимо пажњу је `get`; `store` и `erase` су операције уписа, па када их примимо просто ћемо на чист начин да завршимо извршавање.
2. Формат дељеног фајла акредитива је исти као онај који користи `git-credential-store`.
3. Локација тог фајла је прилично стандардна, али би кориснику за сваки случај ипак требало да омогућимо прослеђивање прилагођене путање.

И овога пута ћемо проширење написати у језику Руби, мада функционише било који језик све док програм Гит буде у стању да изврши завршени производ. Ево комплетног извornог кода нашег новог помоћника за акредитиве:

```
#!/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'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① Овде парсирамо опције из командне линије чиме кориснику омогућавамо да достави улазни фајл. Подразумевана вредност је `~/.git-credentials`.

② Овај програм враћа одговор само у случају да је акција `get` и да постоји фајл позадински

фајл за чување акредитива.

- ③ Ова петља чита са *stdin* све док се не наиђе на празну линију. Улази се чувају у хешу `known` за каснију употребу.
- ④ Ова петља чита садржај фајл који чува акредитиве и тражи подударања. Ако се протокол и хост из `known` подударају са овом линијом, програм исписује резултате на *stdout* и прекида извршавање.

Наш помоћник ћемо сачувати као `git-credential-read-only` негде унутар `PATH` и обележити га тако да се може извршавати. Ево како изгледа интерактивна сесија:

```
$ git credential-read-only --file=/mnt/shared/creds get  
protocol=https  
host=mygithost  
username=bob  
  
protocol=https  
host=mygithost  
username=bob  
password=s3cr3t
```

Пошто име почиње на „git-”, за вредност конфигурационе променљиве можемо употребити једноставну синтаксу:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

Као што видите, проширивање овог система је прилично једноставно и вама и вашем тиму може решити неке честе проблеме.

Резиме

Видели сте већи број напредних алата који вам омогућавају прецизнији рад са комитовима и стејцом. Када приметите проблеме, требало би да лако можете открити који комит их је изазвао, када и ко га је увео. Ако у свом пројекту желите да користите потпројекте, научили сте како да задовољите ту потребу. Сада би требало да можете урадити већину свакодневних ствари у програму Гит које су вам потребне у командној линији и да их радите удобно.

Прилагођавање програма Гит

Досад смо прешли основе начина на који програм Гит функционише и како се користи, а затим смо представили више алата које програм Гит нуди као помоћ за једноставно и ефикасно коришћење. У овом поглављу ћемо видети како програм Гит можете да прилагодите својим потребама, тако што ћемо увести неколико важних конфигурационих подешавања система кука. Помоћу ових алата, лако је подесити програм Гит тако да ради баш онако како ви, ваша компанија или ваша група то жели.

Конфигурисање програма Гит

Како што сте укратко прочитали у [Почетак](#), конфигурациона подешавања програма Гит можете поставити командом `git config`. Једна од првих ствари које сте подесили су били ваше име и мејл адреса:

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

Сада ћете научити још неколико занимљивих опција које можете подесити на овај начин како бисте којима програм Гит прилагођавате својим потребама.

Најпре қратак резиме: програм Гит користи низ конфигурационих фајлова којима одређује неподразумевано понашање које је по вашој вољи. Прво место на коме програм Гит тражи ове вредности су у фајлу [\[путања\]/etc/gitconfig](#) и он садржи подешавања која се примењују на сваког корисника на систему и на све њихове репозиторијуме. Ако команди `git config` проследите опцију `--system`, програм Гит ће вршити упис и читање из овог фајла.

Следеће место које програм Гит гледа је `~/.gitconfig` (или `~/.config/git/config`) фајл који је посебан за сваког корисника. Прослеђивањем опције `--global`, програм Гит можете приморати да чита и пише одавде.

И на крају, програм Гит гледа конфигурационе вредности у конфигурационом фајлу у Гит директоријуму (`.git/config`) оног репозиторијума који тренутно користите. Ове вредности су одређене само за тај један репозиторијум и на њих реферишете прослеђивањем опције `--local` команди `git config`. Ако не наведете ниво са којим желите да радите, ово је подразумевани.

Сваки од ових „нивоа” (системски, глобални, локални) пише преко вредности из претходног нивоа, што значи да ће, на пример, вредности у `.git/config` преиначити оне из [\[путања\]/etc/gitconfig](#).



Конфигурациони фајлови програма Гит се чувају као обичан текст, што значи да све вредности можете подесити и ручном изменом фајла, водећи рачуна о томе да синтакса буде исправна. Ипак, у општем случају је једноставније извршити команду `git config`.

Основна конфигурација клијента

Конфигурационе опције које програм Гит препознаје се могу сврстати у две категорије: клијентске и серверске. Већина опција је на страни клијента — то су оне које подешавају личних радно окружење. Подржано је много, много конфигурационих опција, али велики део њих је користан само у одређеним крајњим случајевима. Овде ћемо покрити само оне најчешће и најкорисније. Ако желите да погледате листу свих опција које препознаје ваша верзија програма Гит, можете да извршите следећу команду:

```
$ man git-config
```

Ова команда приказује све доступне опције, прилично детаљно. Овај референтни материјал можете наћи и на <https://git-scm.com/docs/git-config>.

core.editor

Са подразумеваним подешавањима, програм Гит користи штагод постављено као подразумевани текст едитор преко једне од променљивих љуске `$VISUAL` или `$EDITOR`, а ако није постављена ниједна, користиће едитор `vi` за креирање и уређивање комит порука и порука ознака. Ако желите да промените то подразумевано подешавање на нешто друго, можете да искористите подешавање `core.editor`:

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

Одсада, штагод да је у љуски подешено као подразумевани едитор, програм Гит ће покренути *Emacs* за измену порука.

commit.template

Ако подесите ово на путању до фајла у свом систему, програм Гит ће га користити као подразумевану почетну поруку када комитујете. Вредност у креирању прилагођеног комит шаблона је што га можете употребити да подсетите себе (или друге) на правilan формат и стил када се креира комит порука.

На пример, рецимо да направите шаблон у фајлу `~/.gitmessage.txt` који изгледа овако:

```
Subject line (try to keep under 50 characters)
```

```
Multi-line description of commit,  
feel free to be detailed.
```

```
[Ticket: X]
```

Приметите како овај комит шаблон подсећа комитера да линију теме држи кратком (зарад излаза команде `git log --oneline`), да испод ње дода још детаља и да се позове на проблем или тикет баг трекера ако постоји.

Ако програму Гит жelite рећи да ово користи као подразумевану поруку која се појављује у едитору када извршите `git commit`, треба да подесите конфигурациону вредност `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt  
$ git commit
```

Када комитујете, ваш едитор ће се отворити и приказати нешто овако на месту комит поруке:

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 ваш тим има полису која управља форматом комит поруке, постављање шаблона за ту полису на систем и конфигурисање програма Гит да га подразумевано користи може помоћи да повећате шансе да се та полиса редовно поштује.

core.pager

Ово подешавање одређује који пагинатор се користи када програм Гит дели излазе као што су `log` и `diff` на странице. Можете да га поставите на `more` или на ваш омиљени пагинатор (подразумевано је `less`), или га можете искључити тако што ћете га поставити на празан стринг:

```
$ git config --global core.pager ''
```

Ако извршите ово, програм Гит ће излаз свих команда исписати одједном, без обзира на то колико су дугачки.

user.signingkey

Ако правите потписане прибележене ознаке (којима смо се бавили у [Потписивање вашег](#)

рада), постављање GPG кључа за потписивање као подешавање у конфигурацији ће учинити процес много једноставнијим. Подесите ID свог кључа на следећи начин:

```
$ git config --global user.signingkey <gpg-key-id>
```

Ознаке сада можете да потпишете без потребе да сваки пут наводите кључ у `git tag` команди:

```
$ git tag -s <име-ознаке>
```

core.excludesfile

У `.gitignore` фајл свог пројекта можете поставити шаблоне тако да их програм Гит не би видео као непраћене фајлове или покушао да их дода на стејџ када извршите `git add` над њима, као што смо видели у [Игнорисање фајлова](#).

Али одређене фајлове понекада желите да игноришете у свим репозиторијума са којима радите. Ако се на вашем компјутеру извршава мекОС, вероватно су вам познати `.DS_Store` фајлови. Ако је ваш омиљени едитор *Emacs* или *Vim*, знате за имена фајлова које се завршавају са `~` или `.swp`.

Ово подешавање вам омогућава да напишете неку врсту глобалног `.gitignore` фајла. Ако креирате фајл `~/.gitignore_global` са следећим садржајем:

```
*~  
.*/.swp  
.DS_Store
```

...и извршите `git config --global core.excludesfile ~/.gitignore_global`, програм Гит вам никад више неће правити проблеме са тим фајловима.

help.autocorrect

Ако погрешно укуцате команду, програм Гит вам показује нешто слично овоме:

```
$ git chekcout master  
git: 'chekcout' is not a git command. See 'git --help'.
```

```
The most similar command is  
checkout
```

Програм Гит се труди да вам помогне тако што покушава да одреди коју сте команду желели да укуцате, али ипак одбија да је изврши. Ако `help.autocorrect` поставите на `1`, програм Гит ће вам аутоматски покренути ову команду:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Обратите пажњу на „0.1 seconds”. **help.autocorrect** је уствари цео број који представља десетинке. Ако га подесите на 50, програм Гит ће вам оставити пет секунди да се предомислите пре него што изврши команду коју је аутоматски исправио.

Боје у програму Гит

Програм Гит у потпуности подржава обојени излаз терминала, што доста помаже у бржем и лакшем визуелном парсирању излаза команде. Постоји велики број опција које ће вам помоћи да боје подесите у складу са својим потребама.

color.ui

Програм Гит аутоматски боји већину свог исписа, али постоји главни прекидач који можете искористити уколико вам се овакав начин приказа не допада. Ако желите да искључите све обојени излаз програма Гит на терминалу, урадите следеће:

```
$ git config --global color.ui false
```

Подразумевано подешавање је **auto** и тада се излаз боји када се исписује директно на терминал, али не садржи контролне кодове боја када се излаз преусмерава у пајп или фајл.

Можете га поставити и на **always** ако желите да се не прави разлика између терминала и пајпова. Ово ћете ретко користити; у већини случајева, ако желите кодове за боје у преусмереном излазу, можете уместо тога Гит команди да проследите заставицу **--color** чиме је приморавате да користи и кодове за боје. Подразумевано подешавање је скоро увек оно што ће вам одговарати.

color.*

Ако желите да будете одређенији о томе које команде ће бити обојене и на који начин, програм Гит нуди посебна подешавања за боје. Свака од ових се може подесити на **true**, **false** или **always**.

```
color.branch
color.diff
color.interactive
color.status
```

Сем тога, свака од ових има под-подешавања које можете употребити да поставите боју за делове излаза, ако желите да преиначите сваку боју. На пример, да у **diff** излазу мета информација буде исписана плавим словима на црној позадини, подебљаним словима, можете да извршите следеће:

```
$ git config --global color.diff.meta "blue black bold"
```

Боју можете да подесите на било коју од следећих вредности: `normal` (како је подешен терминал), `black` (црна), `red` (црвена), `green` (зелена), `yellow` (жути), `blue` (плава), `magenta` (магента), `cyan` (тиркизна), или `white` (бела). Ако желите да додате и атрибут као што је `bold` у претходном примеру, можете да изаберете неки од следећих: `bold` (подебљано), `dim` (пригушено), `ul` (подвучено), `blink` (трепери) и `reverse` (замена боје текста и боје позадине).

Спoљни алати за спајање и приказ разлика

Мада програм Гит има своју унутрашњу имплементацију програма `diff`, коју смо досад показивали у овој књизи, уместо њега можете да подесите и спољни алат. Можете да подесите и графички алат за разрешење конфликта при спајању, уместо да их ручно решавате. Приказаћемо подешавање *Perforce Visual Merge Tool* (P4Merge) за приказ разлика и решавање конфликата при спајању, пошто је добар графички алат и бесплатан је.

Ако желите да пробате ово, *P4Merge* ради на свим већим платформама и требало би да успете да га подесите. Надаље ћемо користити имена путањи које раде на мекОС и Линуксу; за Виндууз ћете морати да промените `/usr/local/bin` на путању до извршног фајла у свом окружењу.

За почетак, [преузмите P4Merge са Perforce](#) сајта. Затим ћете подесити спољне омотач скрипте које ће пократати команде. Користићемо мекОС путању за извршни фајл; у другим системима, он ће бити тамо где је инсталiran `p4merge` бинарни фајл. Подесите скрипту-омотач коју можете назвати `extMerge`, која зове бинарни фајл заједно са свим прослеђеним аргументима:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Омотач за приказ разлике проверава да ли је прослеђено седам аргумената и два од њих прослеђује скрипти за спајање. Програм Гит подразумевано прослеђује следеће аргументе програму за приказ разлика:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Пошто су потребни само аргументи `old-file` и `new-file`, користимо скрипту-омотач да проследимо само оне који нам требају.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Такође је потребно и да дозволимо извршавање ових алата:

```
$ sudo chmod +x /usr/local/bin/extMerge  
$ sudo chmod +x /usr/local/bin/extDiff
```

Сада можете подесити конфигурациони фајл тако да се за решавање конфликта и преглед промена користе алати које сте направили. За то је потребно поставити већи број подешавања: `merge.tool` ће рећи програму Гит коју стратегију да користи, `mergetool.<алат>.cmd` наводи како се команда извршава, `mergetool.<алат>.trustExitCode` ће рећи програму Гит да ли излазни код програма говори о успешном решавању конфликта спајања или не, а `diff.external` ће рећи програму Гит коју команду да покрене за приказ промена. Дакле, можете или да покренете четири команде за конфигурацију:

```
$ 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 да измените фајл `~/.gitconfig` тако да додате следеће линије:

```
[merge]  
  tool = extMerge  
[mergetool "extMerge"]  
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"  
  trustExitCode = false  
[diff]  
  external = extDiff
```

Када све ово подесите, ако извршите команду за преглед разлика као што је ова:

```
$ git diff 32d1776b1^ 32d1776b1
```

Уместо да добијете излаз разлике у командној линији, програм Гит покреће P4Merge, који изгледа отприлике овако:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Date Finder</title>
<meta http-equiv="content-type" content="text/html; c
<link rel="stylesheet" href="/stylesheets/application
<!-- javascript_include_tag 'prototype', 'effects' -->
</head>
<body>

Date Finder

<form onsubmit="return false;">
<#> text_field_tag('date') <#>
</form>

<p id="out">...</p>

<div id="footer">please contact us at support@github.co
<script type="text/javascript">

new Form.Element.Observer(
'date',
0.5,
function(el, value){
new Ajax.Request('/main/chronic/' + value, {
method: 'get',
onSuccess: function(transport){
$('out').innerHTML = transport.responseText;
}
});
}
)
</script>
</body>

```

Слика 142. P4Merge

Ако пробате да спојите две гране и као последицу тога добијете конфликте при спајању, можете извршити команду `git mergetool`; она покреће P4Merge и дозвољава вам да решите конфликте у том ГКИ алату.

Добра ствар у вези ове поставке са омотачем је то што лако можете да промените алате за спајање и приказ разлика. На пример, ако желите да промените алате `extDiff` и `extMerge` и подесите их тако да покрећу `KDiff3`, све што треба да урадите је да уредите `extMerge` фајл:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Програм Гит ће сада користити алат `KDiff3` за приказ промена и решавање конфликата при спајању.

Програм Гит се испоручује са великим бројем већ подешених осталих алата за разрешење конфликата при спајању за које не морате да подешавате конфигурацију из командне линије. Ако желите да видите листу алата који су подржани, пробајте ово:

```
$ 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
tortoisediff
xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

Ако програм *KDiff3* не ћете да користите за приказ разлика, већ само за разрешење конфликтата пти спајању, а команда *kdiff3* се налази на вашиј путањи, можете извршити следеће:

```
$ git config --global merge.tool kdiff3
```

Ако извршите ово уместо да подешавате фајлове *extMerge* и *extDiff*, програм Гит ће користити *KDiff3* за разрешење конфликтата, а за приказ разлика уобичајени интерни Гит алат намењен за то.

Форматирање и празан простор

Проблеми са форматирањем и празним простором су неки од фрустрирајућих и суптилних проблема на које наилазе многи програмери који сарађују, поготово када не раде сви на истој платформи. Веома је лако да закрпе и други производи рада на којем се сарађује доведу то неприметних промена у празном простору белине јер их едитори уводе без обавештавања, а ако вам фајлове икад дотакне Виндоуз систем, можда ће се заменити њихови крајеви линија. Програм Гит има неколико конфигурационих опција које помажу код ових проблема.

core.autocrlf

Ако програмирате на Виндузу и радите са људима који га не користе (или обрнуто), вероватно ћете у неком тренутку наићи на проблеме око крајева линија. Разлог за ово је што Виндуз у својим фајловима за прелом реда користи два карактера: и CR (*carriage-return*) и LF (*linefeed*) док мекОС и Линукс системи користе само карактер LF. Ово је суптилна али изузетно фрустрирајућа чињеница рада на различитим платформама; многи едитори на Виндузу без упозорења замењују постојеће крајеве редова у LF стилу са CRLF, или умећу оба карактера за прелом реда када корисник притисне тастер ентер.

Програм Гит ово може да обради аутоматским конвертовањем CRLF крајева линија у LF када додате фајл у индекс и обрнуто када одјављујете код у свој фајл систем. Ову функционалност можете да укључите `core.autocrlf` подешавањем. Ако сте на Виндуз машини, подесите га на `true` — то ће конвертовати све LF у CRLF када одјављујете код:

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

Ако сте на Линукс или мекОС систему који користи LF крајеве линија, онда не желите да их програм Гит аутоматски конвертује када одјављујете фајлове; међутим, ако се случајно уведе фајл са CRLF крајевима линија, онда бисте можда желели да програм Гит то аутоматски исправи. Програму Гит можете рећи да конвертује CRLF у LF када комитујете, али не и обрнуто тако што ћете опцију `core.autocrlf` подесити на `input`:

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

С оваквим подешавањем ћете имати CRLF крајеве линија у Виндуз одјављивањима, али LF крајеве линија на мекОС и Линукс системима и у репозиторијуму.

Ако сте Виндуз програмер који ради на искључиво Виндуз пројекту, ову функционалност можете да искључите, чувајући и CR карактере у репозиторијуму тако што ћете конфигурациону вредност поставити на `false`:

```
$ git config --global core.autocrlf false
```

core.whitespace

Програм Гит може да открије и исправи неке од проблема са празним простором. Може да пронађе шест основна проблема са празним простором — три су подразумевано укључена и могу се искључити, а три су подразумевано искључена али се могу укључити.

Три која су подразумевано укључена су `blank-at-eol`, што тражи размаке не крају линије; `blank-at-eof`, што примећује празне редове на крају фајла; и `space-before-tab`, што тражи размаке испред табова на почетку линије.

Три подразумевано искључена која се могу укључити су `indent-with-non-tab`, што тражи линије које почињу размацима уместо табовима (и контролише се опцијом `tabwidth`); `tab-in-indent`, који надгледа табове у делу линије у којем је увлачење; и `cr-at-eol`, које говори

програму Гит да су CR карактери на крајевима линија у реду.

Програму Гит наводите које од ових желите да укључите тако што ћете поставите `core.whitespace` на вредности које желите укључене или искључене, одвојене зарезима. Опцију можете искључити тако што додате знак `-` испред вредности, или користите њену подразумевану вредност тако што је уопште ни не наведете у стрингу подешавања. На пример, ако желите да укључите све осим `space-before-tab`, то можете урадити (уз то да је `trailing-space` скраћеница која покрива и `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 можете да наведете само део који прилагођавате:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Програм Гит ће ове проблеме открити када извршите команду `git diff` и пробаће да их обоји тако да их можда можете решити пре него што комитујете. Такође ће користити ове вредности да вам помогне када примењујете закрпе командом `git apply`. Када примењујете закрпе, од програма Гит можете затражити да вас упозори ако примењује закрпе које имају наведене проблеме са празним простором:

```
$ git apply --whitespace=warn <закрпа>
```

Или можете наредити програму Гит да покуша аутоматски да реши проблеме пре него што примени закрпу:

```
$ git apply --whitespace=fix <закрпа>
```

Ове опције се такође примењују и на команду `git rebase`. Ако сте комитовали проблеме са празним простором, али их још нисте гурнули узводно, можете да извршите `git rebase --whitespace=fix` и програм Гит ће аутоматски да исправи проблеме до поново исписује закрпе.

Конфигурација сервера

За серверску страну програма Гит не постоји ни приближно оволико конфигурационих опција, али има неколико занимљивих који бисте можда желели да запамтите.

`receive.fsckObjects`

Програм Гит може обезбедити да се сваки објекат примљен током гурања и даље подудара са својом SHA-1 контролном сумом и показује на важеће објекте. Ипак, он то подразумевано не ради; рачунање контролне суме је прилично скупа операција и може да успори

извршавање, поготово на великим репозиторијума или при гурању велике количине података. Ако желите да програм Гит проверава конзистентност објекта приликом сваког гурања, можете га приморати на то тако што ћете поставити `receive.fsckObjects` на `true`:

```
$ git config --system receive.fsckObjects true
```

Сада ће програм Гит проверити интегритет репозиторијума пре него што се свако гурање промена прихвати да би се обезбедило да неисправни (или злонамерни) клијенти не уносе неисправне податке.

`receive.denyNonFastForwards`

Ако ребазирате комитове које сте већ гурнули и онда покушате поново да их гурнете, или на неки други начин покушате да гурнете комит на удаљену грану који не садржи комит на који тренутно показује удаљена грана, та операција вам неће бити одобрена. Ово је у општем случају добра полиса; али у случају ребазирања, можете одлучити да знate шта радите и својој команди гурања проследити заставицу `-f`, чиме принудно ажурирате удаљену грану.

Ако програму Git желите наложити да одбија принудна гурanja, поставите `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Други начин на који ово можете урадити на страни сервера јесте помоћу кука на серверској страни, што ћемо обрадити ускоро. Такав приступ вам нуди могућност да урадите и неке сложеније ствари као што је одбијање спајања која не користе технику брзог премотавања унапред само одређеном подскупу корисника.

`receive.denyDeletes`

Један од начина да се заобиђе `denyNonFastForwards` полиса је да корисник обрише грану, па да је онда поново гурне узводно са новом референцом. Да бисте и ово онемогућили, поставите `receive.denyDeletes` на `true`.

```
$ git config --system receive.denyDeletes true
```

Ово одбија било какво брисање грана или ознака — то не може да ради ниједан корисник. Ако желите да уклоните удаљене гране, морате ручно да обришете фајлове референци са сервера. Такође постоје и занимљивији начини да се ово уради на нивоу корисника преко ACL листи, као што ћете научити у [Пример полисе коју спроводи програм Гит](#).

Гит атрибути

Нека од ових подешавања могу да се наведу и за путању, тако да их програм Гит примењује само за поддиректоријум или за подскуп фајлова. Ова подешавања специфична за путање

се зову Гит атрибути и налазе се или у фајлу `.gitattributes` у неком од директоријума (обично у корену пројекта), или у фајлу `.git/info/attributes` уколико не желите да фајл са атрибутима комитујете уз пројекат.

Коришћењем атрибута можете урадити ствари као што су навођење посебне стратегије спајања за појединачне фајлове или директоријуме свог пројекта, можете рећи програму Гит како да прикаже разлику фајлова који нису текстуалног типа, или да програм Гит филтрира садржај пре него што га пријавите или одјавите из базе програма Гит. У овом одељку ћете научити неке од атрибута које можете поставити у путање Гит пројекта и видећете неколико примера коришћења ове могућности у пракси.

Бинарни фајлови

Један одличан трик за који можете употребити Гит атрибуте јесте да кажете програму Гит који фајлови су бинарни (у случајевима када то на други начин не може да одреди) и да му задате посебна упутства како да обради те фајлове. На пример, неки текстуални фајлови су можда машински генерисани и немогуће је приказати њихову разлику, а док је могуће приказати разлику неких бинарних фајлова. Видећете како да програму Гит кажете шта је шта.

Идентификовање бинарних фајлова

Неки фајлови изгледају као текстуални, али се за све намере и сврхе третирају као бинарни подаци. На пример, Xcode пројекти на мекОС садрже фајл који се завршава са `.pbxproj` и који је у суштини скуп података у JSON формату (JavaScript формат података представљен чистим текстом) који IDE записује на диск, у којем се бележе подешавања за изградњу и слично. Технички је ово текстуални фајл (јер је у UTF-8), али не бисте желели да га третирате као такав јер се у суштини ради о простој бази података – садржаје не можете да спојите ако га две особе истовремено промене, а разлике међу њима у општем случају нису од помоћи. Фајл је предвиђен да га чита машина. У суштини, желите да га третирате као бинарни фајл.

Да бисте рекли програму Гит да су сви `pbxproj` фајлови заправо бинарни подаци, додајте следећи линију у свој `.gitattributes` фајл:

```
*.pbxproj binary
```

Програм Гит сада неће покушавати да среди проблеме са CRLF; нити ће да покушава да израчуна или прикаже разлику за промене у овом фајлу када у свом пројекту извршите `git show` или `git diff`.

Приказивање разлике бинарних фајлова

Функционалност Гит атрибута можете да искористите и за поређење бинарних фајлова. Ово можете урадити тако што ћете програму Гит рећи како да конвертује бинарни фајл у текстуални формат који може да се упореди уобичајеним `diff` алатом.

Прво ћете искористити ову технику да решите један од најнезгоднијих проблема познатих човечанству: контрола верзија докумената писаних у програму Мајкрософт Ворд. Сви знају

да је Ворд један од најужаснијих едитора који постоје и чудновато је што га сви и даље користе. Ако желите да контролишете верзије Ворд докумената, можете да их убаците у Гит репозиторијум и комитујете с времена на време; али какве користи имате од тога? Ако извршите `git diff` на уобичајен начин, видећете само нешто овако:

```
$ git diff  
diff --git a/chapter1.docx b/chapter1.docx  
index 88839c4..4afcb7c 100644  
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Две верзије не можете директно упоредити осим ако их не одјавите и ручно прегледате, зар не? Испоставља се да ово прилично добро можете урадити помоћу Гит атрибута. Ставите следећу линију у свој `.gitattributes` фајл:

```
*.docx diff=word
```

Ово говори програму Гит да када пробате да погледате разлике између промена за сваки фајл који одговара шаблону (`.docx`) треба да употреби филтер „word”. Али шта је филтер „word”? Морате да га подесите. Овде ћете програм Гит подесити да употреби програм `docx2txt` који конвертује Ворд документе у читљиве текстуалне фајлове, за које ћете онда моћи да прегледате разлике како ваља.

Најпре, мораћете да инсталирате `docx2txt`; можете га преузети са <http://docx2txt.sourceforge.net>. Пратите упутства из фајла `INSTALL` како бисте га инсталерили негде где га љуска може пронаћи. Затим ћете написати скрипту-омотач која ће конвертовати излаз у формат који програм Гит очекује. Креирајте фајл под именом `docx2txt` негде на путањи и уметните у њега следећи садржај:

```
#!/bin/bash  
docx2txt.pl "$1" -
```

Не заборавите да над фајлом извршите и `chmod a+x`. И коначно, конфигуришите програм Гит тако да користи ову скрипту:

```
$ git config diff.word.textconv docx2txt
```

Програм Гит сада зна да, ако покуша да прикаже разлике између два снимка, а било који од фајлова се завршава са `.docx`, те фајлове треба пропусти кроз филтер `word`, који је дефинисан као `docx2txt` програм. Ово у суштини ствара фине текстуалне верзије Ворд фајлова пре него што се проба њихово поређење.

Ево примера: прво поглавље ове књиге је конвертовано у Ворд формат и комитовано у Гит репозиторијум. Онда је додат нови пасус. Ево шта приказује `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.

Програм Гит нам успешно и недвосмислено каже да смо додали стринг „Testing: 1, 2, 3”, што је тачно. Метода није савршена – овде се не појављују измене у форматирању – али несумњиво ради.

Још један занимљив проблем који можете решити на овај начин јесте преглед разлика између слика. Један од начина да урадите ово јесте да пропустите слике кроз филтер који издваја њихове EXIF податке – метаподатке који се бележе у већини формата за слике. Ако преузмете и инсталирати програм [exiftool](#), можете да га искористите да конвертујете слике у текст о метаподацима, како бисте приликом прегледа разлика барем видели текстуалну презентацију промена које су се можда дододиле. Поставите следећу линију у свој [.gitattributes](#) фајл:

```
*.png diff=exif
```

Подесите програм Git тако да користи овај алат:

```
$ git config diff.exif.textconv exiftool
```

Ако у пројекту промените слику и извршите `git diff`, видећете нешто овако:

```
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
```

Лако можете видети да су се промениле и димензије и величина фајла.

Проширење кључних речи

Програмери који су навикли да користе SVN и CVS често захтевају проширење (експанзију) кључних речи. Главни проблем са овиме у програму Гит је што не можете да измените фајл са информацијом о комиту након што сте комитовали, јер програм Гит прво рачуна контролну суму фајла. Ипак, можете убрзати текст у фајл када се одјави и уклонити га непосредно пре него што се дода у у комит. Гит атрибути вам нуде два начина да урадите ово.

Најпре, можете аутоматски да убрзати SHA-1 контролну суму блоба у `Id` поље фајла. Овај атрибут можете подесити за фајл или скуп фајлова, па када следећи пут одјавите ту грану, програм Гит ће заменити то поље са SHA-1 контролном сумом блоба. Важно је приметити да ово није SHA-1 комита, већ самог блоба. Поставите следећу линију у свој `.gitattributes` фајл:

```
*.txt ident
```

Додајте `Id` референцу на тест фајл:

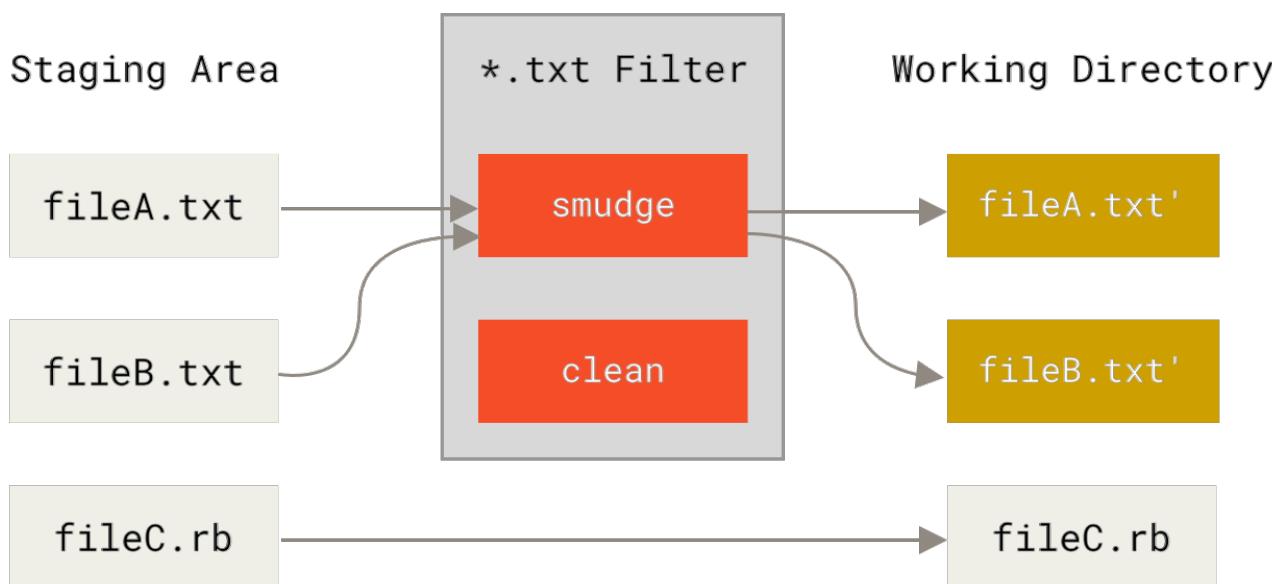
```
$ echo '$Id$' > test.txt
```

Следећи пут када одјавите овај фајл, програм Гит убризгава SHA-1 блоба:

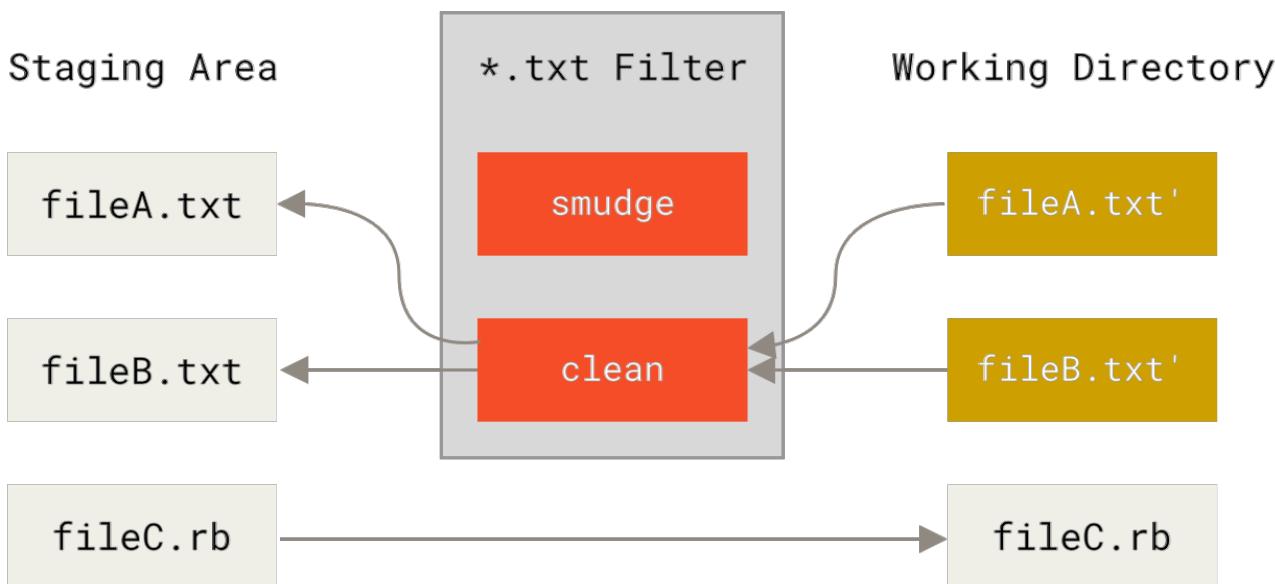
```
$ rm test.txt  
$ git checkout -- test.txt  
$ cat test.txt  
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Међутим, тај резултат има ограничену употребу. Ако сте користили замену кључних речи у програму *CVS* или *Subversion*, могли сте да укључите и печат с датумом — SHA-1 није толико користан јер је поприлично насумичан и само на основу посматрања вредности не можете закључити да ли је неки SHA-1 старији или новији од неког другог.

Испоставља се да можете да напишете своје филтере који ће радити замену у фајловима при комитовању/одјављивању. Ови филтери се зову „чисти” (*clean*) и „замрљани” (*smudge*). У фајлу `.gitattributes` можете поставити филтер за одређене путање па онда подесити скрипте које ће обрадити фајлове непосредно пре него што се одјаве („замрљани”, погледајте „Замрљани” филтер се покреће приликом одјављивања) и непосредно пре него што се поставе на стејџ („чисти”, погледајте "Чисти" филтер се покреће када се фајлови постављају на стејџ). Ови филтери се могу подесити тако да раде гомилу занимљивих ствари.



Слика 143. „Замрљани” филтер се покреће приликом одјављивања



Слика 144. "Чисти" филтер се покреће када се фајлови постављају на стејџ

Првобитна комит порука за ову могућност вам даје једноставан пример пропуштања комплетног изворног кода написаног у језику С кроз програм `indent` пре него што се обави комит. Можете да поставите ово тако што ћете подесити филтер атрибут у фајлу `.gitattributes` тако да филтрира `*.c` фајлове кроз filter `indent`:

```
*.c filter=indent
```

Онда реците програму Гит шта филтер `indent` ради за `smudge` и `clean`:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

У овом случају, када комитујете фајлове које одговарају шаблону `*.c`, Гит ће их пропустити кроз програм за увлачење редова пре него што их постави на стејџ и онда ће их пропустити кроз програм `cat` пре него што их одјавите назад на диск. Програм `cat` у суштини не ради ништа: избацује исте податке које је добио на улазу. Ова комбинација делотворно филтрира сав изворни код на језику С кроз филтер `indent` пре комитовања.

Још један занимљив пример показује проширење кључне речи `$Date$`, у стилу програма RCS. Да бисте ово урадили како треба, требаће вам мала скрипта која узима име фајла, одређује датум последњег комита за овај пројекат, и убацује тај датум у фајл. Ево мале Руби скрипте која ради управо то:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Све што скрипта ради јесте да узме датум последњег комита из излаза команде `git log`, убаци то у било који стринг `$Date$` који види на `stdin` и штампа резултате – требало би да будеовољно једноставно да направите скрипту у било ком другом језику. Овај фајл можете назвати `expand_date` и ставити га на путању. Сада треба да подесите филтер у програму Гит (зваћемо га `dater`) и рећи му да треба да користи ваш филтер `expand_date` да замрља фајлове приликом одјављивања. Користићемо Перл израз да то почистимо приликом комита:

```
$ git config filter.dater.smudge expand_date  
$ git config filter.dater.clean 'perl -pe "s/\$\$Date[^\$\$]*\$\$/\$Date\\\$/"'
```

Овај исечак Перл кода склања све што види у стрингу `$Date$`, како бисте се вратили назад на почетно стање. Сада када је филтер спреман, можете да га тестирате постављањем Гит атрибути за тај фајл који укључује нови филтер и креирањем фајла са `$Date$` кључном речи:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Ако комитујете те промене па онда опет одјавите фајл, видећете да се кључна реч заменила како треба:

```
$ 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 видети колико ова техника може бити моћна у посебним применама. Ипак, морате бити обазриви, јер се фајл `.gitattributes` комитује и прослеђује са пројектом, али не и драјвер (у овом случају `dater`), што значи да неће радити свуда. Када дизајнирате ове филтере, треба да их направите тако да елегантно прекину извршавање, а да пројекат и даље функционише како ваља.

Извоз репозиторијума

Подаци из Гит атрибути вам омогућавају и да радите неке занимљиве ствари када извозите или архивирате свој пројекат.

export-ignore

Можете рећи програму Гит да не извози одређене фајлове или директоријуме када генерише архиву. Ако постоји поддиректоријум или фајл коју не желите да укључите у архиву али желите да су пријављени у пројекат, можете да их задате помоћу атрибути `export-ignore`.

На пример, рецимо да имате неке фајлове за тестирање у поддиректоријуму `test/` и нема смисла укључивати их у *tarball* пројекта. Можете додати следећу линију у свој фајл Гит атрибута:

```
test/ export-ignore
```

Ако сада извршите `git archive` да креирате *tarball* пројекта, тај директоријум неће бити укључен у архиву.

export-subst

Када извозите фајлове за постављање у употребу можете да примените форматирање наредбе `git log` и обраду проширења кључних речи на одређени скуп фајлова који су обележени атрибутом `export-subst`.

На пример, ако желите да у свој пројекат укључите фајл под именом `LAST_COMMIT` и да се метаподаци о последњем комиту аутоматски убрзгају када се изврши `git archive`, можете поставити своје `.gitattributes` и `LAST_COMMIT` фајлове рецимо овако:

```
LAST_COMMIT exportsubst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Када извршите `git archive`, садржај архивираног фајла ће изгледати овако:

```
$ git archive HEAD | tar xf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

Замене могу да укључе, на пример, комит поруке и било какве `git notes`, а `git log` може да уради једноставан прелом редова:

```
$ 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 xf0 - 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.
```

Архива која се добије као резултат је погодна за постављање за употребу, али као и свака друга извезена архива није погодна за даљи рад на развоју.

Стратегије спајања

Гит атрибуте можете искористите и да кажете програму Гит да користи различите стратегије спајања за одређене фајлове у пројекту. Једна веома корисна опција је да програму Гит кажете да не покушава да споји одређене фајлове када имају конфликте, већ да увек користи вашу страну у спајању уместо туђе.

Ово је корисно ако се грана у вашем пројекту разишла или је специјализована, али желите да будете у могућности да промене у њој спојите назад и желите да игноришете одређене фајлове. Рецимо да имате фајл подешавања базе података `database.xml` која се разликује у две гране и желите да своју другу грану спојите, а да се фајл базе података не забрља. Атрибут можете да поставите овако:

```
database.xml merge=ours
```

Па да затим овако дефинишете лажну `ours` стратегију:

```
$ git config --global merge.ours.driver true
```

Ако спојите другу грану, уместо да имате конфликте у фајлу `database.xml`, видећете нешто овако:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

У овом случају, `database.xml` остаје у оној верзији коју сте првобитно имали.

Гит куке

Као и многи други системи за контролу верзије, програм Гит има начина да позове произвољно написане скрипте када се догоде одређене важне акције. Постоје две групе ових кука: оне на клијентској и оне на серверској страни. Куке на клијентској страни се окидају операцијама као што је комитовање и спајање, а куке на серверској страни се покрећу при мрежним операцијама као што је прихватање турнутих комитова. Ове куке можете да користите за разноразне намене.

Инсталација куке

Куке се чувају у `hooks` поддиректоријуму Гит директоријума. У већини пројеката, то је `.git/hooks`. Када командом `git init` иницијализуете нови репозиторијум, програм Гит попуњава директоријум `hooks` гомилом скрипти које служе као примери, од којих су многе саме по себи корисне; али такође и документују улазне вредности сваке скрипте. Сви примери су написано као скрипте љуске, уз мало Перла убаченог ту и тамо, али било која ваљано именована извршна скрипта ће радити посао — можете их писати у Рубију, Пајтону или у било ком језику који познајете. Ако желите да користите скрипте за куке које су већ испоручене уз репозиторијум, мораћете да им промените име; њихова имена се сва завршавају на `.sample`.

Да бисте омогућили скрипту за куку, поставите пригодно назван (без било какве екstenзије) извршни фајл у `hooks` поддиректоријуму вашег .git директоријума. Од тог тренутка па надаље, требало би да се позива. Овде ћемо покрити већину главних имена фајлова за куке.

Куке на страни клијента

Постоји много кука за клијентску страну. Овај одељак их дели на куке који се тичу процеса рада комитовања, скрипте које се тичу процеса рада са имејлом, и све остале.



Важно је приметити да се куке на клијентској страни **не** копирају када клонirate репозиторијум. Ако вам је план да овим скриптама заведете поштовање полисе, вероватно ћете то хтети да урадите на серверској страни; погледајте пример у [Пример полисе коју спроводи програм Гит](#).

Куке у вези процеса комитовања

Прве четири куке имају везе са процесом комитовања.

Кука `pre-commit` се покреће прва, чак и пре него што откуцате комит поруку. Користи се за инспекцију снимка који ће бити комитован, да се провери да ли сте нешто заборавили, да се постара да тестови пролазе, или да се истражи штагод је потребно да истражите у коду. Ако скрипта ове куке на kraју свог извршавања врати вредност различиту од нуле, комитовање се прекида, мада то можете да избегнете командом `git commit --no-verify`. Можете да радите ствари као што су провера стила кода (да покренете `lint` или нешто слично), провера вишке празног простора на kraју редова (подразумевана кука ради управо ово), или да проверите да ли постоји одговарајућа документација за нове методе.

Кука `preargc-commit-msg` се покреће пре него што се покрене едитор за измену комит поруке, али након што се креира подразумевана порука. Дозвољава вам да измените подразумевану поруку пре него што је види аутор комита. Ова кука узима неколико параметара: путању до фајла који садржи тренутну комит поруку, врсту комита и SHA-1 комита ако се ради о комиту који се мења (`amend`). У општем случају ова кука није корисна за обичне комитове, али је добра за комитове где се подразумевана порука аутоматски генерише, као што су шаблонске комит поруке, комитови спајања, згњечени комитови, и изменјени комитови. Можете их користити у спрези са шаблоном за комит да програмски уметнете информације.

Кука `commit-msg` узима један параметар који представља путању до привременог фајла који садржи комит поруку коју је написао програмер. Ако ова скрипта врати излазну вредност различиту од нуле, програм Гит обуставља процес комитовања, тако да је можете користити да потврдите стање пројекта или комит поруку пре него што допустите да се комит обави. У последњем одељку овог поглавља, показаћемо како се овај кука користи за проверу да ли се комит порука подудара са прописаним шаблоном.

Након што се обави цео процес комитовања, покреће се `post-commit` кука. Оне нема никакве параметре, али лако можете добити последњи комит тако што извршите `git log -1 HEAD`. У општем случају, ова скрипта се користи за обавештење или нешто слично.

Куке у вези процеса рада са имејловима

Можете поставити три куке на клијентској страни за процес рада базиран на имејловима. Све се позивају командом `git am`, па ако ту команду не користите у свом процесу рада, можете слободно да прескочите овај одељак. Ако закрпе припремљене командом `git format-patch` добијате путем имејла, онда ће вам неке од ових можда бити корисне.

Прва кука која се покреће је `applypatch-msg`. Узима један аргумент: име привременог фајла који садржи предложену комит поруку. Програм Гит обуставља примену закрпе ако је враћена вредност из скрипте различита од нуле. Можете је користити да обезбедите исправно форматирање комит поруке, или да нормализујете поруку тако што ће је скрипта изменити у ходу.

Следећа кука која се покреће приликом примењивања закрпа са `git am` је `pre-applypatch`. Мало је збуњујуће, али покреће се након што се закрпа примени, али пре него што се направи комит, тако да је можете користити да прегледате снимак пре него што комитујете. Овом скриптом можете покренути тестове или на неки други начин проверити стање радног стабла. Ако нешто недостаје или тестови не пролазе, напуштање скрипте са враћеном вредности различитом од нуле прекида команду `git am` и закрпа се не комитује.

Последња кука која се покреће током извршавања команде `git am` је `post-applypatch`, и она се покреће након што се уради комит. Можете је користити да обавестите групу или аутора закрпе коју сте довукли да сте је успешно применили. Овом скриптом не можете прекинути процес закрпљивања.

Остале куке на клијентској страни

Кука `pre-rebase` се покреће пре него што ребазирате било шта и може да заустави процес ако врати излазну вредност различиту од нуле. Ову куку можете употребити да не дозволите

ребазирање комитова који су већ раније гурнути. `pre-rebase` кука која је инсталрирана као пример уз програм Гит ради управо то, мада претпоставља одређене ствари које се можда не уклапају у ваш процес рада.

Куку `post-rewrite` покрећу команде које врше замену комитова, као што је `git commit --amend` и `git rebase` (мада не и `git filter-branch`). Њен једини аргумент је команда која је окинула поновно писање комита, а листу тих преписивања преузима са `stdin`. Многе примене ове куке су исте као и за `post-checkout` или `post-merge` куке.

Након успешно извршене команде `git checkout`, покреће се `post-checkout` кука; можете је користити да на одговарајући начин припремите радни директоријум тако да одговара окружењу пројекта. Ово може значити пресељење великих бинарних фајлова које не желите да буду део система за контролу верзије, аутоматско генерирање документације, или нешто слично томе.

Кука `post-merge` се покреће после успешно извршене команде `merge`. Можете је користити за враћање података које програм Гит не може да прати у радни директоријум, као што су подаци о дозволама. Ова кука исто тако може да потврди присуство фајлова ван контроле програма Гит које можда желите да копирате у радно стабло када се оно измене.

Кука `pre-push` се покреће уз команду `git push`, након што се ажурирају удаљене референце или пре него што се било који објекат пренесе. Као параметре прима име и локацију удаљене референце, а листу референци које ће бити ажуриране са `stdin`. Можете је користити за проверу исправности скупа референци пре него што се догоди само гурање (ако скрипта врати излазну вредност различиту од нуле, гурање се прекида).

Програм Гит повремено ради уклањање ђубрета као део свог нормалног процеса рада тако што изврши `git gc --auto`. Кука `pre-auto-gc` се покреће непосредно пре почетка процеса скупљања ђубрета и може се користити да вас обавести да се ово дешава, или да се прекине извршење скупљања ђубрета ако сада није добар тренутак за то.

Куке на страни сервера

Уз куке на страни клијента, као системски администратор можете да користите и неколико важних кукса стране сервера како бисте применили скоро било коју врсту полисе на ваш пројекат. Ове скрипте се покрећу пре и после гурања на сервер. Пре-куке у било које време могу да врате излазну вредност различиту од нуле и тиме одбију гурање, као и да клијенту испишу поруку о грешки; полиса гурања може да буде сложена онолико колико ви то желите.

`pre-receive`

Прва скрипта која се покреће када се обрађује гурање са клијентске стране је `pre-receive`. Она са `stdin` узима листу референци које се гурају; ако заврши извршавање и врати вредност различиту од нуле, ниједна референца се не прихвата. Ову куку можете користити да обезбедите да су све ажуриране референце типа брзо премотавање унапред, или да примените контролу приступа за све референце и фајлове које оне мењају гурањем.

update

Скрипта `update` је веома слична скрипти `pre-receive`, сем што се покреће по једном за сваку грану коју клијент који гура покушава да ажурира. Ако клијент који гура покушава да гурне на више грана, `pre-receive` се покреће само једном, док се `update` покреће по једном за сваку грану на коју се гура. Уместо да чита са `stdin`, ова скрипта узима три аргумента: име референце (гране), SHA-1 комита на који је референца показивала пре гурања и SHA-1 комита који клијент покушава да гурне. Ако скрипта `update` заврши извршавање и врати вредност различиту од нуле, само та референца ће бити одбијена; остале и даље могу да се ажурирају.

post-receive

Кука `post-receive` се покреће након завршетка комплетног процеса и може да се употреби за ажурирање осталих сервиса или за обавештавање корисника. Узима исте податке са `stdin` као и `pre-receive` кука. Примери укључују мејлинг листу, обавештавање сервера за континуирану интеграцију, или ажурирање система за праћење тикета – можете чак и да парсирате комит поруку да откријете да ли неки тикети треба да се отворе, измене или затворе. Ова скрипта не може да обустави процес гурања, али веза са клијентом се не прекида све док се скрипта не изврши у целости, зато будите обазриви ако покушавате да урадите нешто што захтева много времена.



Ако пишете скрипту/куку коју ће други да читају, боље је да употребљавате дугачке верзије заставица командне линије; након шест месеци ћете нам бити захвални.

Пример полисе коју спроводи програм Гит

У овом одељку ћете искористити оно што сте до сада научили да успоставите процес рада са програмом Гит који проверава да ли је испуњен одређени формат комит поруке и само одређеним корисницима дозвољава да мењају одређене поддиректоријуме у пројекту. Направићете клијентске скрипте које помажу програмеру да зна да ли ће подаци које жели да гурне бити прихваћени или не и серверску скрипту која заправо спроводи полисе.

Скрипте које ћемо представити су писане у Рубију; делимично због наше интелектуалне инерције, али и због тога што се Руби лако чита, иако не мора да значи да њему нешто умете и да напишете. Међутим, сваки језик ће функционисати — све скрипте које се испоручују уз програм Гит су или на Перлу или на Бешу, па ако их погледате видећете и пуно примера кука на овим језицима.

Кука на серверској страни

Сав рад на серверској страни ће се налазити у фајлу `update` у директоријуму `hooks`. Кука `update` се покреће по једном за сваку грану која се гура и узима три аргумента:

- име референце на коју се гура,
- стара ревизија где се грана раније налазила и
- нова ревизија која се гура.

Сем тога, имате и приступ кориснику који обавља гурање ако се оно обавља преко SSH протокола. Ако сте дозволили свакоме да се повеже као један корисник (нпр. „git“) преко аутентификације јавним кључем, можда ћете морати да том кориснику дате омотач љуске који на основу јавног кључа одређује који се корисник повезао, па да сходно томе, подесите променљиву окружења. Овде ћемо претпоставити да се корисник који се повезује налази у променљивој окружењу `$USER`, тако да скрипта `update` почине тако што прикупља све информације које су вам потребне:

```
#!/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, то су глобалне променљиве. Не осуђујте нас — лакше је показати пример тако.

Справођење одређеног формата за комит поруке

Први изазов је наметнути полису која прописује да свака комит порука мора да буде у одређеном формату. Чисто да бисмо имали неки циљ, претпоставићемо да свака порука мора да садржи стринг који изгледа као „ref: 1234“, јер желите да сваки комит буде повезан са неким тикетом. Морате да прегледате сваки комит који се гура, да испитате да ли се тај стринг налази у комит поруци, па ако недостаје у макар једном комиту, завршите извршавање и вратите вредност различиту од нуле како бисте одбили гурање.

Можете да добити листу SHA-1 вредности свих комитова који се гурају тако што ћете узети вредности `$newref` и `$oldref` и проследити их водоводној команди програма Гит `git rev-list`. Ово је у суштини команда `git log`, али подразумевано исписује само SHA-1 вредности и никакве друге информације. Дакле, да бисте добили листу SHA-1 вредности свих комитова који су уведени између два комита задата својим SHA-1 вредностима, извршите нешто овако:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475
```

Можете да узмете тај излаз, да прођете петљом кроз сваку од ових SHA-1 вредности комита, узмете поруку за њега, и тестирате је регуларним изразом који тражи одговарајући шаблон.

Морате пронаћи начин да добијете комит поруку сваког од ових комитова како бисте је тестирали. За добијање сирових података о комиту, можете употребити још једну водоводну

команду под именом `git cat-file`. У Гит изнутра ћемо детаљније прећи ове водоводне команде; засад, ево шта вам она враћа:

```
$ 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

Једноставан начин да добијете комит поруку из комита када имате његов SHA-1 јесте да одете на прву празну линију и узмете све после тога. То можете урадити командом `sed` на Јуникс системима:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
Change the version number
```

Ту чаролију можете употребити да зграбите комит поруку из сваког комита који покушава да буде гурнут, па да завршите извршавање ако видите нешто што се не уклапа. Да бисте завршили извршавање скрипте и одбили то гурање, вратите излазну вредност различиту од нуле. Цела метода изгледа овако:

```
$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
```

Ако ово поставите у скрипту `update`, биће одбијено гурање података који садрже комитове чије се поруке не задовољавају ваше правило.

Справођење ACL система базiranog на корисницима

Рецимо да желите додати механизам који користи листу за контролу приступа (ACL) која одређује којим корисницима је дозвољено гурање промена на које делове пројекта. Неки људи могу имати потпун приступ, а други само могу да гурају промене одређених поддиректоријума или одређених фајлова. Да бисте спровели овакву полису, та правила

морате да запишете у фајл под именом `acl` који треба да сместите у огњени Гит репозиторијум на серверу. Кука `update` ће читати та правила, видеће које фајлове уводе сви комитови који се гурају и одредиће да ли корисник који врши гурање има приступ да ажурира све те фајлове.

Прва ствар коју треба да урадите јесте да напишете ACL. Овде ћемо користити формат који веома личи на CVS ACL механизам: користи низ линија, где је прво поље `avail` или `unavail`, следеће поље је листа корисника који за које важи правило раздвојена зарезима, док је последње поље путања за коју важи правило (при чему празно значи отворени приступ). Ова поља су раздвојена карактером вертикална црта (`|`).

У овом случају, имате неколико администратора, неке људе задужене за документацију који имају приступ директоријуму `doc` и једног програмера који има приступ само директоријумима `lib` и `tests`, па онда ваш ACL изгледа овако:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Почињете учитавањем ових података у структуру коју можете да искористите. У овом случају, да би пример био једноставан, спровешћете само `avail` директиве. Ево методе која вам има асоцијативни низ у којем је кључ корисничко име, а вредност је низ путања у којима тај корисник има дозволу за писање:

```
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
```

Када се метода `get_acl_access_data` позове над ACL фајлом који сте видели раније, она враћа структуру података која изгледа овако:

```
{"defunkt"=>[nil],  
 "tpw"=>[nil],  
 "nickh"=>[nil],  
 "pjhyett"=>[nil],  
 "schacon"=>["lib", "tests"],  
 "cdickens"=>["doc"],  
 "usinclair"=>["doc"],  
 "ebronte"=>["doc"]}
```

Сада када сте средили питање дозвола, треба да одредите које то путање мењају комитови који се гурају, тако да бисте били сигурни да корисник има приступ свима тим путањама.

Фајлове које мења један комит прилично лако можете да видите користећи опцију `--name-only` команде `git log` (која је кратко поменута у [Основе програма Гит](#)).

```
$ git log -1 --name-only --pretty=format:'' 9f585d  
  
README  
lib/test.rb
```

Ако употребите ACL структуру коју је вратила метода `get_acl_access_data` и упоредите је са фајловима излистаним за сваки од комитова, можете одредити да ли корисник има право да гурне све своје комитове:

```

# дозвољава само одређеним корисницима да измене одређене поддиректоријуме
# у пројекту
def check_directory_perms
    access = get_acl_access_data('acl')

    # провери да ли неко покушава да гурне промене које не сме
    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 # корисник има приступ свему
                    || (path.start_with? access_path) # приступ само овој путањи
                    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

```

Командом `git rev-list` добијате листу нових комитова који се турају на ваш сервер. Онда, за сваки од тих комитова проналазите фајлове који се мењају и постарате се да корисник који тура има приступ свим путањама које се мењају.

Сада ваши корисници не могу да турају комитове са лоше форматираним порукама или са променама које мењају фајлове ван путања које су им додељене.

Тестирање

Ако извршите `chmod u+x .git/hooks/update`, што је фајл у који би требало да поставите сав овај код, па онда пробате да гурнете комит са поруком која не задовољава полису, добићете нешто овако:

```
$ 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'
```

Овде има неколико занимљивих ствари. Најпре, можете видети где се тачно покреће кука.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Присетите се да се то исписали на самом почетку **update** скрипте. Све што ваша скрипта испише на *stdout* се преноси клијенту.

Следећа ствар коју можете уочити јесу поруке о грешкама.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Прву линију сте ви исписали, а остале две програм Гит који вам говори да је скрипта **update** завршила извршавање и вратила вредност различиту од нуле и да је то разлог због чега турање није успело. На крају, ту је и ово:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Видећете **remote rejected** поруку за сваку референцу коју је кука одбила и то вам говори да је ваш захтев одбијен зато што се кука није извршила успешно.

Штавише, ако неко покуша да изменi фајл којем нема приступ, па онда гурне комит који га садржи, видеће нешто слично. На пример, ако аутор документације покуша да гурне комит који мења нешто у директоријуму **lib**, видеће следеће:

```
[POLICY] You do not have access to push to lib/test.rb
```

Одсад па надаље, све док је скрипта `update` на свом месту и може да се изврши, репозиторијум никада неће имати комит поруку која у себи нема ваш шаблон, а ваши корисницима ће приступ деловима пројекта бити прецизно одређен.

Куке на страни клијента

Лоша страна овог приступа су притужбе које ће се несумњиво јавити када комитови ваших корисника не буду прихваћени. Кад се рад у који је уложено пуно труда одбије у последњем тренутку, то зна да буде доста фрустрирајуће и збуњујуће; штавише, мораће да измене своју историју како би исправили проблем, а то није увек посао за оне са слабим срцем.

Одговор на ову дилему је постављање неких кука на страни клијента које корисници могу да покрену и које ће их обавестити да раде нешто што ће сервер вероватно одбити. На тај начин могу да реше проблем пре него што комитују и пре него што ти проблеми постану тежи за решавање. Пошто се неке куке не преносе заједно са клоном пројекта, те скрипте морате дистрибуирати на неки други начин и да објасните корисницима да их сместе у `.git/hooks` директоријум и да их учине извршним. Ове куке можете да дистрибуирате у оквиру пројекта или у посебном пројекту, али програм Гит их неће аутоматски поставити.

За почетак, треба да проверите комит поруку пре него што се сваки комит забележи, како бисте знали да вам сервер неће одбити промене због лоше форматираних комит порука. Да бисте урадили ово, можете додати `commit-msg` куку. Ако је подесите тако да чита поруку из фајла који се проследи као први аргумент, па онда упореди то са шаблоном, програм Гит можете навести да обустави креирање комита ако се не пронађе подударање:

```
#!/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 се та скрипта налази на одговарајућем месту (у `.git/hooks/commit-msg`) и подеси као извршни фајл, а направите комит са поруком која није исправно форматирана, видећете следеће:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

У том случају се комит не довршава. Међутим, ако комит порука садржи одговарајући

шаблон, програм Гит вам дозвољава да креирате комит:

```
$ git commit -am 'test [ref: 132]'  
[master e05c914] test [ref: 132]  
1 file changed, 1 insertions(+), 0 deletions(-)
```

Затим желите да се осигурате да не мењате фајлове који су ван вашег ACL опсега важења. Ако `.git` директоријум ваšег пројекта садржи копију ACL фајла коју сте раније користили, онда ће следећа `pre-commit` скрипта спровести ограничења која су вам наметнута:

```
#!/usr/bin/env ruby  
  
$user      = ENV['USER']  
  
# [ убаците овде методу acl_access_data одозго ]  
  
# дозволи само одређеним корисницима да мењају одређене поддиректоријуме  
пројекта  
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  
end  
  
check_directory_perms
```

Ово је отприлике иста скрипта као и она на серверској страни, али уз две кључне разлике. Прво, ACL фајл се налази на другом месту, јер се ова скрипта покреће из вашег радног директоријума, а не из `.git` директоријума. Морате да промените путању до ACL фајла са:

```
access = get_acl_access_data('acl')
```

на следеће:

```
access = get_acl_access_data('.git/acl')
```

Још једна важна разлика је начин на који добијате листу свих фајлова који су промењени. Пошто метода на серверској страни гледа лог комитова, а у овом тренутку комит још увек није забележен, листу фајлова морате да узмете из стејџа. Уместо:

```
files_modified = 'git log -1 --name-only --pretty=format:'' #{ref}'
```

морате да користите:

```
files_modified = 'git diff-index --cached --name-only HEAD'
```

Али ово су и једине две разлике – сем тога, скрипта ради на потпуно исти начин. Једина зачкољица је у томе што скрипта очекује да се извршава локално као исти корисник који ће обавља гурање на удаљену машину. Ако то није случај, морате да ручно подесите променљиву `$user`.

Још једна ствар коју овде можемо да урадимо јесте да се постарамо да корисник не сме да гурне референце које нису брзо премотавање унапред. Да бисте добили референцу која није премотавање унапред, морате или да ребазирате прошли комит који сте већ гурнули или да покушате гурање различите локалне грани на исту удаљену.

Претпоставља се да је сервер већ конфигурисан са `receive.denyDeletes` и `receive.denyNonFastForwards` које спроводе ову политику, тако да је једина случајна ствар коју можете покушати да ухватите ребазирање комитова који су већ раније гурнути.

Ево примера `pre-base` скрипте која ради управо то. Преузима листу свих комитова које планирате да поново испишете и проверава да ли постоје на било којој од ваших удаљених референци. Ако види барем један до којег може да се стигне из једне од ваших удаљених референци, прекида ребазирање.

```

#!/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

```

Ова скрипта користи синтаксу која није обрађена у [Избор ревизија](#). Овако добијате листу комитова који су већ раније гурнути:

```
'git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}'
```

Синтакса `SHA^@` се разрешава на све родитеље тог комита. Тражите било који комит до којег може да се стигне из последњег комита на удаљеном репозиторијуму и до којег не може да се стигне од било ког родитеља било којих SHA-1 које покушавате да гурнете – што значи да се ради о брзом премотавању унапред.

Главни недостатак овог приступа је што уме да буде веома спор и често непотребан — ако не покушавате принудно гурање заставицом `-f`, сервер ће вас упозорити и неће прихватити гурнуте комитове. Ипак, ово је занимљива вежба и теоретски вам може помоћи да избегнете ребазирање на које ћете касније морати да се вратите и да га исправите.

Резиме

Покрили смо већину главних начина на које можете прилагодити свој Гит клијент и сервер тако да се најбоље уклопе у ваш процес рада и пројекте. Научили сте много тога о разним конфигурационим подешавањима, атрибутима фајлова и догађајима кука, а изградили сте и пример сервера који форсира полису. Сада би требало да можете подесити програм Гит тако да се уклапа у било који процес рада који вам падне на памет.

Гит и остали системи

Свет није савршен. У већини случајева не можете одмах да пребаците сваки пројекат са којим дођете у контакт на Гит. Некад сте заглављени на пројекту који користи неки други VCS, а волели бисте да је то Гит. Први део овог поглавља ћемо посветити учењу начина на које можете да користите Гит као клијент када је пројекат на којем радите хостован на другом систему.

У једном тренутку ћете можда пожелети да конвертујете свој постојећи пројекат у Гит. Други део овог поглавља покрива начин миграције пројекта на Гит из неколико одређених система, као и метода која ће функционисати ако не постоје већ готови алати за увоз.

Гит као клијент

Програм Гит је програмерима толико угодан за коришћење да су многи људи открили како да га користе на својој радној станици, чак и ако остатак тима користи потпуно другачији VCS. Постоји много оваквих адаптера који се називају „мостови”. Овде ћемо покрити неке на које ћете највероватније наићи.

Гит и Subversion

Велики део развојних пројекта отвореног кода и порпиличан број корпоративних пројекта користе Subversion да управља њиховим изворним кодом. Присутан је већ више од десет година и већину тог времена је био *де факто* VCS избор за пројекте отвореног кода. У многим стварима је врло сличан са CVS који је пре тога био татамата света контроле изворног кода.

Једна од одличних могућности програма Гит је двосмерни мост према програму Subversion који се зове `git svn`. Овај алат вам омогућава да програм Гит користите као важећи клијент Subversion сервера, тако да можете користити све локалне могућности програма Гит и да их затим гурнете на Subversion сервер као да се и локално користили Subversion. Ово значи да можете радити локално гранање и спајање, користите стејџ, употребљавате ребазирање и избор (*cherry-picking*) и тако даље, док ваши сарадници настављају да раде на своје мрачне прастаре начине. То је добар начин да провучете програм Гит у корпоративно окружење и помогнете да ваше колеге програмери постану ефикаснији док ви лобирате да се инфраструктура промени тако да у потпуности подржава програм Гит. Subversion мост је уводна дрога у DVCS свет.

`git svn`

Основна команда у програму Гит за све команде премошћавања до Subversion је `git svn`. Постоји поприличан број команди, тако да ћемо приказати најчешће док пролазимо кроз неколико једноставних процеса рада.

Важно је приметити да када користите `git svn`, ви имате интеракцију са програмом Subversion, а то је систем који функционише веома различито у односу на програм Гит. Мада можете да радите локално гранање и спајање, у општем случају је најбоље да историју држите што је могуће више линеарном тако што ребазирујете свој рад и избегавате да радите

ствари као што је истовремена интеракција са удаљеним Гит репозиторијумом.

Немојте поново да исписујете своју историју и покушате поново да је гурнете и немојте да истовремено гурате на паралелни Гит репозиторијум како би сарађивали са колегама који за развој користе Гит. Subversion може да има са једну линеарну историју, и врло је једноставно да га збуњите. Ако радите у тиму где неко користи SVN, а остали користе Гит, обезбедите да сви за сарадњу користе SVN сервер – на тај начин ће вам живот бити једноставнији.

Подешавање

Да бисте пробали ову функционалност, потребан вам је типичан SVN репозиторијум на који има те право уписа. Ако желите да копирате ове примере, мораћете да направите копију мог тест репозиторијума по којој можете да пишете. То једноставно можете урадити алатом који се зове `svnsync` и долази уз програм Subversion. Као подршку овим тестовима, направили смо нови Subversion репозиторијум на Google Code који је представљао делимичну копију пројекта `protobuf`, алата који кодира структуре података тако да могу да се преносе кроз мрежу.

Да бисте успешно пратили, најпре морате да креирате нови локални Subversion репозиторијум:

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

Затим омогућите свим корисницима право да мењају `revprops` – то се на једноставан начин ради тако да додате скрипту `pre-revprop-change` која увек враћа повратну вредност 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change  
#!/bin/sh  
exit 0;  
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Овај пројекат сада можете да синхронизујете на своју локалну машину тако што позовете `svnsync init` са 'из' и 'у' репозиторијумима.

```
$ svnsync init file:///tmp/test-svn \  
http://progit-example.googlecode.com/svn/
```

Ово поставља особине за позив синхронизације. Затим клонирате код извршавајући:

```
$ 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.  
[...]
```

Мада ова операција може да потраје само неколико минута, ако уместо на локални оригинални репозиторијум покушате да копирате на други удаљени репозиторијум, процес ће трајати скоро сат времена, мада постоји мање од 100 комитова. Subversion мора да клонира ревизије једну по једну, па да их затим турне на други репозиторијум – потпуно неефикасно, али је то једини начин да се ово уради.

Први кораци

Сада када имате Subversion репозиторијум у који можете да уписујете, можете да прођете кроз типичан процес рада. Почекете да командом `git svn clone` која увози комплетан Subversion репозиторијум у локални Гит репозиторијум. Упамтите да ако увозите из реалног хостованог Subversion репозиторијума, фајл `file:///tmp/test-svn` приказан овде треба да замените са URL адресом вашег Subversion репозиторијума:

```
$ 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
```

Ово извршава еквивалент две команде – `git svn init` након које следи `git svn fetch` – над URL адресом коју наведете. Може да потраје. Тест пројекат има само око 75 комитова и база кода није толико велика, али програм Гит мора да одјави сваку верзију, једну по једну, па да је засебно комитује. За пројекат са стотинама или хиљадама комитова, ово буквально може да потраје сатима или чак данима док се не заврши.

`-T trunk -b branches -t tags` део говори програму Гит да овај Subversion репозиторијум прати

основне конвенције гранања и означавања. Ако свом стаблу, гранама или ознакама желите да дате другачија имена, промените ове опције. Пошто је ово толико уобичајено, комплетан део можете да замените са `-s`, што значи стандардни распоред и подразумева све те опције. Следећа команда је еквивалентна:

```
$ git svn clone file:///tmp/test-svn -s
```

У овом тренутку би требало да имате исправан Гит репозиторијум које увезао ваше гране и ознаке:

```
$ 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
```

Приметите како овај алат управља Subversion ознакама као удаљеним референцама. Хајде да детаљније погледамо водоводном командом програма Гит `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711fed4 refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Програм Гит ово не ради када клонира са Гит сервера; ево како изгледа репозиторијум након свежег клонирања:

```
$ 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
6dc09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Програм Гит преузима ознаке директно у `refs/tags`, уместо да их третира као удаљене гране.

Комитовање назад на Subversion

Сада када имате функционалан репозиторијум, можете да урадите неки посао на пројекту и гурнете своје комитове узводно, тако што програм Гит ефективно користите као SVN клијент. Ако уредите један од фајлова и комитујете га, имаћете комит који постоји локално у програму Гит и не постоји на Subversion серверу:

```
$ git commit -am 'Adding git-svn instructions to the README'  
[master 4af61fd] Adding git-svn instructions to the README  
 1 file changed, 5 insertions(+)
```

Затим, своју измену морате да гурнете узводно. Приметите како ово мења начин на који радите са програмом Subversion – можете да урадите неколико комитова док ста ван мреже, па да их све одједном гурнете на Subversion сервер. Када желите да гурнете на Subversion сервер, извршите команду `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
```

Ово узима све комитове које сте направили преко кода са Subversion сервера, за сваки од њих врши Subversion комит, па онда поново исписује ваш локални Гит комит тако да укључи јединствени идентификатор. То је важно јер значи да се све SHA-1 контролне суме ваших комитова мењају. Делимично због овог разлога, истовремени рад са удаљеним Гит верзијама вашег пројекта и Subversion сервером није добра идеја. Ако погледате последњи комит, можете видети уметнути нови `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
```

Приметите да је SHA-1 контролна suma када сте раније комитовали почињала са `4af61fd`, а да сада почиње са `95e0222`. Ако желите да гурнете и на Гит и на Subversion сервер, прво морате да гурнете (`dcommit`) на Subversion сервер, јер та акција мења ваше комит податке.

Повлачење нових измена

Ако радите са другим програмерима, онда ће у неком тренутку један од вас гурнути, па ће онда неко други покушати да гурне измену која изазива конфликт. Та измена ће бити одбијена док не спојите њихов рад. У `git svn` то изгледа овако:

```
$ 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.
```

Да бисте разрешили ову ситуацију, можете да извршите `git svn rebase`, која повлачи евентуалне измене на серверу које још увек немате и ребазира ваш рад (ако га има) на врх онога што се налази на серверу:

```
$ 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 65536c6e30d263495c17d781962cff12422693a
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.
```

Сада је сви ваш рад преко онога што се налази на Subversion серверу, тако да можете успешно извршити `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
```

Приметите да за разлику од програма Гит, који од вас захтева да пре него што гурнете спојите сав узводни рад који још увек немате локално, `git svn` од вас то тражи само ако измене изазивају конфликт (слично као што ради и Subversion). Ако неко други гурне измену једног фајла, панда ви гурнете измену неког другог фајла, ваша `dcommit` ќе функционисати без проблема:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M configure.ac
Committed r87
    M autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977fffc61adff7 (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 efa5a59965fb6b5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M autogen.sh
First, rewinding head to replay your work on top of it...
```

Ово је важно да се запамти, је крајњи исход стање проекта које није постојало ни на једном од ваших компјутера када сте гурнули. Ако измене нису компатибилне али не изазивају конфликт, можете имати проблеме који се тешко откривају. То је разлика у односу на употребу Гит сервера – у програму Гит Git стање можете потпуно тестирати на свом клијентском систему пре него што га објавите, док у SVN, никада не можете бити сигурни да су стања непосредно пре и након комита идентична.

Ову команду би такође требало да извршите за повлачење измена са Subversion сервера, чак и ако сами нисте спремни да комитујете. Можете извршити `git svn fetch` да само преузмете нове податке, али `git svn rebase` преузима па ажурира ваше локалне комитове.

```
$ 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.
```

Повремено извршавање `git svn rebase` обезбеђује да је ваш кôд увек у свежем стању. Ипак, да бисте ово извршили, морате бити сигурни да је ваш радни директоријум чист. Ако имате локалне измене, морате прво или да сакријете свој рад, или да га привремено комитујете пре покретања `git svn rebase` – у супротном ће команда прекинути извршавање ако види да ће резултат ребазирања изазвати конфликт при спајању.

Проблеми са Гит гранањем

Када вам Гит процес рада постане удобан, вероватно ћете почети да креирате тематске гране, радите на њима, па их спајате у главну. Ако гурате на Subversion сервер командом `git svn`, уместо да гране спајате могли бисте да свој рад ребазирате на једну грану. Ребазирање је пожељније од спајања јер Subversion има линеарну историју и не носи се са спајањима као што то ради програм Гит, тако да `git svn` следи само првог родитеља када конвертује снимке у Subversion комитове.

Претпоставимо да вам историја изгледа на следећи начин: креирали сте `experiment` грану, направили два комита, па их онда спојили назад у `master`. Када извршите `dcommit`, добијате следећи излаз:

```
$ 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
```

Извршавање `dcommit` на грани са спојеном историјом ради како треба, није поново исписала ниједан од комитова које сте направили на `experiment` грани – уместо тога, све те измене се појављују у SVN верзији једног јединог комита спајања.

Када неко други клонира тај рад, све што види је комит спајања са комплетним радом сабијеним у њега, као да сте извршили `git merge --squash`; не види податке о томе одакле је комит дошао или када је направљен.

Subversion гранање

Гранање у програму Subversion није исто као гранање у програму Гит; вероватно је најбоље ако можете избећи да га често користите. Ипак, са `git svn` можете да креирате и комитујете Subversion гране.

Креирање нове SVN гране

Ако желите да креирате нову Subversion грану, извршите `git svn branch [имегране]`:

```
$ 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)
```

Ово је еквивалентно Subversion команди `svn copy trunk branches/opera` и оперише на Subversion серверу. Важно је приметити да она вас она не одјављује у ту грану; ако у овом тренутку комитујете, так комит иде у `trunk` на серверу, а не у `opera`.

Пребацивање активних грана

Програм Git открива на коју грану иду ваши `dcommit` тако што гледа на врх било које Subversion гране у вашој историји – требало би да имате само једну, а то би у исорији тренутне гране требало да буде последња са `git-svn-id`.

Ако желите истовремено да радите на више од једне гране, команди `dcommit` можете да подесите локалне гране на одређене Subversion гране започињући их као увезени Subversion комит за ту грану. Ако желите грану `opera` на којој можете одвојено да радите, извршите:

```
$ git branch opera remotes/origin/opera
```

Ако сада желите да своју `opera` грану спојите у `trunk` (односно вашу `master` грану), то можете урадити обичном `git merge` командом. Али морате навести описну комит поруку (са `-m`), или ће спајање уместо нечег корисниг да буде „Merge branch `opera`”.

Упамтите да мада за ову операцију користите `git merge` и спајање ће вероватно бити доста једноставније него што би било у програму Subversion (јер ће програм Гит аутоматски за вас да открије погодну базу за спајање), ово није уобичајен Гит комит спајања. Ове податке морате да гурнете назад Subversion серверу који не може да обради комит који прати више од једног родитеља; дакле, након што га гурнете узводно, изгледаће као један једини комит који је сабио сваки рад на другој грани у један комит. Када једну грану спојите у другу, можете једноставно да се вратите назад и наставите да радите на тој грани, као и иначе што бисте у програму Гит. Команда `dcommit` коју извршавате брише било какве информације које кажу која грана је спојена, тако да ће накнадна израчунавања базе спајања вити погрешна – `dcommit` чини да резултат ваше `git merge` команде изгледа као да се извршили `git merge --squash`. Нажалост, не постоји добар начин да се избегне оваква ситуација – Subversion не може да забележи ову информацију, тако да ће вас увек спутавати његова ограничења док год га користите као свој сервер. Да бисте спречили проблеме, када је спојите у `trunk` требало

би да обришете локалну грану (у овом случају, [орега](#)).

Subversion команде

`git svn` скуп алата обезбеђује већи број команди које помажу да се олакша прелаз на Гит нудећи неку функционалност која је слична оној коју имате у програму Subversion. Ево неколико команда које вам пружају оно што је Subversion.

Историја у SVN стилу

Ако сте навикили на Subversion и своју историју желите да видите као излаз у SVN стилу, you можете да извршите `git svn log` која вам приказује SVN форматирану комит историју:

```
$ 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
```

Требало би да знате две важне ствари у вези команде `git svn log`. Прво, она за разлику од праве `svn log` команде која тражи податке од Subversion сервера тражи податке, ради ван мреже. Друго, она вам приказује само оне комитове који су комитовани на Subversion сервер. Локални Fit комитови над којима још нисте извршили `dcommit` се не појављују; као ни комитови које су у међувремену остали доставили на Subversion сервер. Она пре приказује последње познато стање комитова на Subversion серверу.

SVN означавање

На сличан начин на који `git svn log` команда симулира `svn log` команду ван мреже, еквивалент команде `svn annotate` можете добити извршавањем `git svn blame [ФАЈЛ]`. Излаз изгледа овако:

```
$ 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
```

Опет, она не приказује комитове које сте локално направили у програму Гит, или оне који су у међувремену гурнути на Subversion сервер.

Информације о SVN серверу

Исту врсту информација које вам пружа `svn info` можете добити ако извршите `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)
```

Ово је слично са `blame` и `log` у томе што се извршава ван мреже и што је ажурано само до последњег тренутка када сте имали комуникацију са Subversion сервером.

Игнорисање онога што игнорише Subversion

Ако клонирате Subversion репозиторијум који негде има постављене `svn:ignore` особине, вероватно ћете хтети да поставите одговарајуће `.gitignore` фајлове тако да случајно не комитујете фајлове које не би требало. `git svn` има две команде које вам помажу у томе. Прва је `git svn create-ignore`, која уместо вас аутоматски креира одговарајуће `.gitignore` фајлове тако да ваш наредни комит не може да их укључи.

Друга команда је `git svn show-ignore`, која на `stdout` исписује линије које треба поставите у `.gitignore` фајл, тако њен излаз можете да преусмерите у фајл изузетака пројекта:

```
$ git svn show-ignore > .git/info/exclude
```

На тај начин не затрпавате пројекат `.gitignore` фајловима. Ово је добра опција у случају да сте једини Гит корисник у Subversion тиму и ваше колеге из тима не желе `.gitignore` фајлове у пројекту.

Git-Svn резиме

`git svn` алату су корисни ако сте заглављени на Subversion серверу, или у неком развојном окружењу у којем је неопходно покретање Subversion сервера. Ипак, требало би да га сматрате ограниченим програмом Гит, или ћете наићи на проблеме у превођењу коју могу да збуне и вас и ваше сараднике. Покушајте се држати следећих смерница како не бисте упали у невоље:

- Одржавајте линеарну Гит историју која не садржи `git merge` комитове спајања. Ребазирајте сваки рад који радите ван главне гране назад на њу; не спајајте у главну грану.
- Не постављавате и не сарађујте на одвојеном Гит серверу. По могућству имајте један који убрзава клонирања за нове програмере, али не гурајте на њега ништа што нема `git-svn-id` ставку. Чак можете да додате и `pre-receive` куку која проверава сваку комит поруку на постојање `git-svn-id` и одбија гурања које садрже комит поруке без ње.

Ако следите ове препоруке, рад са Subversion сервером може постати подношљив. Међутим, ако постоји могућност, пређите на прави Гит сервер, на тај начин ћете и ви и тим добити још много више.

Гит и Меркуријал

DVCS свет није само Гит. Уствари, постоји много других система од којих сваки има свој угао посматрања на то како се исправно врши дистрибуирана контрола верзија. Осим програма Гит, најпопуларнији је Меркуријал и они су слични у многим аспектима.

Добра вест је, ако вам се свиђа понашање програма Гит на клијентској страни, али радите на пројекту чији изворни код контролише репозиторијум који хостује Меркуријал, постоји начин да програм Гит користите као клијент Меркуријал репозиторијума. Пошто је начин на који програм Гит разговара са серверским репозиторијумима помоћу удаљених референци, не би требало да буде изненађење што је овај мост имплементиран као удаљени помоћник. Име пројекта је `git-remote-hg`, и може да се пронађе на адреси <https://github.com/felipec/git-remote-hg>.

git-remote-hg

Најпре је потребно да инсталирате `git-remote-hg`. То практично значи да поставите његов фајл негде на путању, отприлике овако:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...уз претпоставку да се `~/bin` налази у вредности ваше `$PATH` променљиве. `Git-remote-hg` има још једну зависност: `mercurial` библиотеку за Пајтон. Ако имате инсталацију језика Пајтон, то је просто као пасуљ:

```
$ pip install mercurial
```

Ако још увек немате инсталацију Пајтон, најпре посетите <https://www.python.org/> и набавите га.

Последња ствар која вам је потребна је Меркуријал клијент. Посетите <https://www.mercurial-scm.org/> и инсталирајте га, ако то већ нисте учинили.

Сада можете да пређете на ствар. Потребан вам је само Меркуријал репозиторијум на који можете да гурате. На срећу, сваки Меркуријал репозиторијум може да се понаша на овај начин, тако да ћемо једноставно употребити „hello world” репозиторијум који користи свако да научи Меркуријал:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Први кораци

Сада када имамо погодан „серверски” репозиторијум, можемо да прођемо кроз типични процес рада. Као што можете да видите, ова два система су доволјно слична тако да нема много трења.

Најпре клонирамо, као што је увек и обичај у програму Гит:

```
$ 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
```

Приметићете да рад са Меркуријал репозиторијумом користи стандардну `git clone` команду. То је зато јер `git-remote-hg` функционише на прилично ниском нивоу, користећи сличне механизме којима је имплементиран HTTP/S протокол у програму Гит (удаљени помоћници). Пошто су и Гит и Меркуријал дизајнирани тако да сваки клијент има комплетну копију историје репозиторијума, ова команда креира потпуни клон, укључујући комплетну историју пројекта, а то обавља прилично брзо.

Log команда приказује два комита, од којих на последњи показује читава шума референци. Испоставља се да неке од њих заиста и нису ту. Хајде да погледамо шта се заиста налази у `.git` директоријуму:

```
$ 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 покушава да ствари буду што природније у Гит стилу, али под хаубом он управља концептуалним мапирањем између два незнатно различита система. Директоријум `refs/hg` је место где се чувају стварне референце. На пример, `refs/hg/origin/branches/default` је фајл Гит референце који садржи SHA-1 што почиње на „ac7955c” и то је комит на који показује `master`. Тако да је директоријум `refs/hg` нека врста лажног `refs/remotes/origin`, али уз додато прављење разлике између маркера и грана.

Фајл `notes/hg` је почетна тачка из које *git-remote-hg* мапира Гит комит хешеве у Меркуријал идентификаторе скупа измена. Хајде да истражимо мало:

```
$ 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
```

Дакле, `refs/notes/hg` показује на стабло које у Гит бази података објекта представља листу осталих објекта са именима. `git ls-tree` исписује режим, тип, хеш објекта и име фајла за

ставке унутар стабла. Када дођемо да једне од ставке стабла, видимо да се у њој налази блоб под именом „ac9117f” (SHA-1 хеш комита на који показује `master`), и садржајем „0a04b98” (ID Меркуријал скупа измена на врху `default` гране).

Добра вест је да углавном уопште нема потребе да бринемо о свему овоме. Типичан процес рада се неће много разликовати од рада са Гит удаљеним репозиторијумом.

Постоји још једна ствар коју би требало да обрадимо пре него што наставимо: игнорисања. Меркуријал и Гит за то користе врло сличан механизам, али вероватно не желите да заиста комитујете `.gitignore` фајл у Меркуријал репозиторијум. На сву срећу, програм Гит има начин да игнорише фајлове који су локални репозиторијуми на диску, а Меркуријал формат је компатибилан са Гит форматом, тако да једноставно можете да га прекопирате:

```
$ cp .hgignore .git/info/exclude
```

Фајл `.git/info/exclude` се понаша као `.gitignore`, али није део комитова.

Процес рада

Хајде да претпоставимо да смо урадили неки посао и направили неколико комитова на `master` грану, па сте сада спремни да их турнете на удаљени репозиторијум. Ево како наш репозиторијум сада изгледа:

```
$ 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
```

Наша `master` грана се налази два комита испред `origin/master`, али та два комита постоје само на нашој локалној машини. Хајде да видимо да ли је још неко у исто време радио важан посао:

```
$ 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
```

Пошто смо употребили заставицу `--all`, видимо „notes” референце које *git-remote-hg* интерно користи, али их можемо занемарити. Остatak је оно што смо и очекивали; `origin/master` је напредовала за један комит, па се наша историја сада раздвојила. За разлику од осталих система са којима радимо у овом поглављу, Меркуријал је способан да обради спајања, тако да нема потребе да радимо било шта спектакуларно.

```
$ 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
```

Савршено. Покрећемо тестове и све пролази како треба, тако да смо спремни да свој рад поделимо са остатком тима:

```
$ git push
To hg:::/tmp/hello
 df85e87..0c64627 master -> master
```

И то је то! Ако погледате Меркуријал репозиторијум видећете да је ова команда урадила оно

што смо и очекивали:

```
$ 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
```

Скуп измена под бројем 2 је направио Меркуријал, а скупове измена 3 и 4 *git-remote-hg*, творањем комитова који су креирани програмом Гит.

Гране и маркери

Програм Git има само једну врсту грана: референцу која се помера када се направе комитови. У програму Меркуријал се ова врста референце назива „маркер” (*bookmark*) и понаша се скоро исто као и грана у програму Гит.

Концепт „гране” у програму Меркуријал има много дубље значење. Грана на којој се креира скуп измена се чува *заједно са скупом измена*, што значи да ће увек бити у историји репозиторијума. Ево примера комита који је направљен на `develop` грани:

```
$ 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
```

Приметите линију која почиње са „branch”. Програм Гит ово не може у потпуности да преслика (и нема потребе за тим; оба типа гране се могу представити као Гит референца), али *git-remote-hg* мора да разуме разлику, јер је Меркуријал прави.

Креирање Меркуријал маркера је једноставно као креирање Гит грана. На Гит страни:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

И то је све. На Меркуријал страни, изгледа овако:

```
$ 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
```

Приметите нову ознаку **[featureA]** на ревизији 5. Оне се понашају потпуно исто као Гит гране са стране програма Гит, уз један изузетак: маркер не можете да обришете са стране програма Гит (ово је ограничење помоћника за удаљене).

Такође можете да радите и на „тешким” Меркуријал гранама: једноставно ставите грану у **branches** простор имена:

```
$ 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
```

Ево како то изгледа са Меркуријал стране:

```
$ 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'
[...]
```

Име гране „permanent” је снимљено са скупом измена под ознаком 7.

Гледано са Гит стране, рад са било којим од ова два стила грана је сити: једноставно одјављујете, комитујете, преузимате, спајате, повлачите и гурате као и иначе. Једна ствар коју би требало да знate је да Меркуријал не подржава поновно исписисвање историје, може само да додаје у историју. Ево како наш Меркуријал репозиторијум изгледа након интерактивног ребазирања и наметнутог гурања:

```
$ 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
```

Креирани су скупови измена 8, 9, и 10 и припадају **permanent** грани, али су још увек присутни стари скупови измена. Ово може **тотално** да збуни ваше колеге из тима који користе Меркуријал, па би било добро да то не радите.

Меркуријал резиме

Програми Гит и Меркуријал су довољно слични да је рад са друге стране границе прилично безболан. Ако избегнете мењање историје која је остала на вашој машини (као што се и иначе препоручује), можда нећете ни приметите да је са друге стране Меркуријал.

Гит и Базаар

Још један од чувених DVCS је [Базаар](#). Базаар је слободан и отвореног кода и чини део [GNU пројекта](#). Понаша се потпуно другачије од програма Гит. Понекада, ако желите да урадите исту ствар као у програму Гит, морате да употребите другу кључну реч, а неке заједничке

кључне речи немају исто значење. Уствари, управљање гранама је веома различито и може да унесе забуну, посебно некоме ко долази из Гит света. Без обзира на све, могуће је радити на Базаар репозиторијуму из Гит репозиторијума.

Постоје многи пројекти који вам омогућавају да програм Гит користите као Базаар клијент. Овде ћемо употребити пројекат Фелипеа Контрераса који можете пронаћи на адреси <https://github.com/felipec/git-remote-bzr>. Да бисте га инсталериали, једноставно треба да преузмете фајл *git-remote-bzr* и поставите га у директоријум који је део **\$PATH**:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O  
~/bin/git-remote-bzr  
$ chmod +x ~/bin/git-remote-bzr
```

Такође је потребно да имате инсталiran програм Базаар. И то је све!

Креирање Гит репозиторијума из Базаар репозиторијума

Ово је једноставно. Довољно је да клонирате Базаар репозиторијум тако што испред имена наведете **bzr::**. Пошто и Гит и Базаар раде потпуно клонирање на вашу машину, Гит клон може да се прикачи на Базаар клон, али се то не препоручује. It's much easier to attach your Git clone directly to the same place your Bazaar clone is attached to — the central repository.

Претпоставимо да сте радили са удаљеним репозиторијумом који се налази на адреси **bzr+ssh://developer@mybazaarserver:myproject**. Затим морате да га клонирате на следећи начин:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git  
$ cd myProject-Git
```

У овом тренутку имате креиран Гит репозиторијум, али он није оптимизован за ефикасно искоришћење диска. Зато би требало да пречистите и сажмете ваш Гит репозиторијум, посебно ако је велики:

```
$ git gc --aggressive
```

Базаар гране

Базаар вам дозвољава само да клонирате гране, али репозиторијум може да садржи неколико грана, па **git-remote-bzr** може да клонира и једно и друго. На пример, ако желите да клонирате грану:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

А да клонирате комплетан репозиторијум:

```
$ git clone bzr://bzr.savannah.gnu.org/emacs emacs
```

Друга команда клонира све гране које се налазе у *emacs* репозиторијуму; међутим, ипак је могуће да се укаже на поједине гране:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Неки репозиторијуми вам не дозвољавају да видите њихове гране, па их у том случају морате ручно навести, мада бисте могли да наведете конфигурацију у команди клонирања, можда вам је ово лакше:

```
$ git init emacs
$ git remote add origin bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

Игнорисање онога што се игнорише у *.bzrignore*

Пошто радите на пројекту којим управља Базаар, не би требало да креирате *.gitignore* фајл јер нехотично можете да га поставите под контролу верзија и то може засметати осталим људима који користе Базаар. Решење је да фајл *.git/info/exclude* креирате или као симболички линк или као обичан фајл. Касније ћемо видети како се решава ово питање.

Базаар за игнорисање фајлова користи исти модел као и програм Гит, али има и две могућности које немају свој еквивалент у програму Гит. Комплетан опис се налази у [документацији](#). Могућности су следеће:

1. „!!” вам дозвољава да игноришете одређене фајл шаблоне чак и када су наведени употребом „!” правила.
2. "RE:" на почетку линије вам дозвољава да наведете [Пајтон регуларни израз](#) (програм Гит дозвољава само глобове љуске).

Као последица, постоје две различите ситуације о којима треба да размислите:

1. Ако *.bzrignore* фајл не садржи ниједан од ова два посебна префикса, онда једноставно у репозиторијуму можете направити симболички линк на њега: `ln -s .bzrignore .git/info/exclude`.
2. У супротном, морате да креирате фајл *.git/info/exclude* и да га прилагодите тако да се игноришу потпуно исти фајлови као у *.bzrignore*.

Шта год да је случај, мораћете остати на опрезу у случају било какве измене фајла *.bzrignore* и да осигурате да фајл *.git/info/exclude* увек пресликава *.bzrignore*. Заиста, ако би измена *.bzrignore* фајла садржала једну или више линија које почињу са „!!” или „RE:”, како програм Git не може те линије да интерпретира, морали бисте да преправите свој *.git/info/exclude* фајл тако да игнорише исте фајлове као оне које игнорише *.bzrignore* фајл. Уз то, ако је *.git/info/exclude* фајл уствари симболички линк, прво морате да обришете симболички

линк, копирате `.bzrignore` у `.git/info/exclude` па да онда прилагодите овај други. Међутим, будите опрезни у вези креирања овог фајла, јер је са програмом Гит немогуће поновно укључивање фајла ако је директоријум родитељ тог фајла искључен.

Пријем измена са удаљеног репозиторијума

Да бисте преузели измене са удаљеног, повлачите измене као и обично, употребом команди програма Гит. Под претпоставком да су ваше измене на `master` грани, спајате или ребазирате свој рад на `origin/master` грани:

```
$ git pull --rebase origin
```

Гурање вашег рада на удаљени репозиторијум

Пошто и Базаар има концепт комитова спајања, неће бити проблема ако гурнете комит спајања. Тако да можете радити на грани, спојити измене у `master` и гурнути свој рад. Затим, креирате своје гране, тестирате и комитујете рад као и обично. На крају гурнете свој рад на Базаар репозиторијум:

```
$ git push origin master
```

Ограничења

Радни оквир програма Гит за удаљене помоћнике има нека ограничења која важе. Тачније, следеће команде не функционишу:

- `git push origin :грана-за-брисање` (Базаар не прихвата овакав начин брисања референци)
- `git push origin old:new` (гурнуће се `old`)
- `git push --dry-run origin branch` (извршиће се гурање)

Резиме

Пошто су модели програма Гит и Базаар слични, нема много отпора када радите с друге стране границе. Док год пазите на ограничења и свесни сте да удаљени репозиторијум није природно Гит, биће све у реду.

Гит и Перфорс

Перфорс је веома популаран систем за контролу верзија у корпоративним окружењима. На тржишту је од 1995. године што га чини најстаријим системом који је покривен овим поглављем. Као такав, дизајниран је са ограничењима која су важила тих дана; претпоставља да сте увек повезани на један централни сервер и на локалном диску се чува само једна верзија. Истини за вольу, његове могућности и ограничења су прилагођене неколицини специфичних проблема, али постоји много пројекта који користе Перфорс тамо где би Гит у суштини функционисао боље.

Постоје две опције ако желите да мешате употребу програма Перфорс и Гит. Прва коју ћемо

представите је *Git Fusion* мост који праве творци програма Перфорс и који вам дозвољава да подстабла Перфорс депоа изложите као Гит репозиторијума по којима може да се чита и пише. Друга је *git-p4*, мост на клијентској страни који вам омогућава да програм Гит користите као Перфорс клијент, без потребе било какве реконфигурације Перфорс сервера.

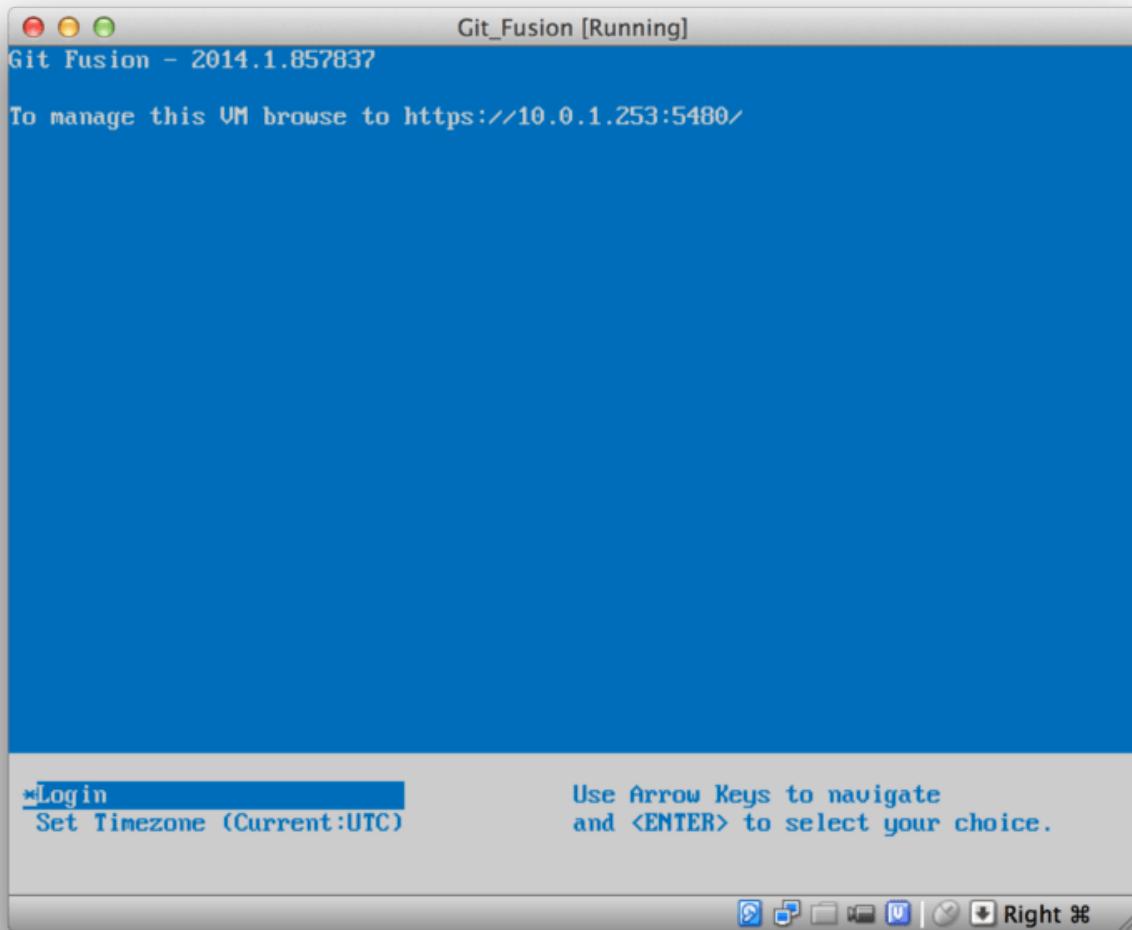
Git Fusion

Перфорс нуди производ под називом *Git Fusion* (доступан на <http://www.perforce.com/git-fusion>), који са серверске стране синхронизује Перфорс сервер са Гит репозиторијумима.

Подешавање

У нашим примерима ћемо користи најлакшу методу инсталације програма *Git Fusion*, која представља преузимање виртуелне машине која извршава Перфорс даемон и *Git Fusion*. Слику виртуелне машине можете преузети са адресе <http://www.perforce.com/downloads/Perforce/20-User>, па када се преузимање заврши, увезете је у свој омиљени софтвер за виртуелизацију (овде ћемо користити *VirtualBox*).

Када по први пут покренете машину, питаће вас да подесите лозинку за три Линукс корисника (`root`, `perforce`, и `git`) и наведете име инстанце, које се користи да се та инсталација разликује од осталих на истој мрежи. Када се све то заврши, видећете следеће:



Слика 145. Екран при покретању *Git Fusion* виртуелне машине

Требало би да прибележите IP адресу која се овде прикаже, касније ћемо је користити. Затим ћемо да креирамо Перфорс корисника. Изаберите опцију „Login” на дну екрана и притисните ентер (или направите SSH везу са машином), и пријавите се као `root`. Затим употребите следеће команде да креирате корисника:

```
$ p4 -p localhost:1666 -u super user -f john  
$ p4 -p localhost:1666 -u john passwd  
$ exit
```

Прва команда ће покренути VI едитор да прилагодите корисника, али можете и прихватити подразумевана подешавања тако што откуцате `:wq` и притиснете ентер. Друга ће од вас захтевати да двапут унесете лозинку. То је све што је потребно да се уради у одзиву љуске, па напустите сесију.

Следећа ствар коју морате урадити је да програму Гит наложите да не проверава SSL сертификате. *Git Fusion* слика долази са сертификатом, али он је за домен који се не подудара са IP адресом ваше виртуелне машине, тако да ће програм Гит одбити да успостави HTTPS везу. Ако ће ово бити стална инсталација, консултујте Перфорс *Git Fusion*

упутство да бисте инсталirали одговарајући сертификат; за наше потребе, следеће ће бити довољно:

```
$ export GIT_SSL_NO_VERIFY=true
```

Сада можете проверити да ли све ради како треба.

```
$ 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.
```

Слика виртуелне машине долази опремљена са примером пројекта који можете да клонирате. Овде ћемо да клонирамо преко HTTPS, користећи *john* корисника којег смо управо креирали изнад; програм Git тражи акредитиве за ову везу, али ће нам кеш акредитива омогућити да онај корак прескочимо у наредним захтевима.

Fusion конфигурација

Када инсталирате програм *Git Fusion*, пожелећете да измените конфигурацију. Ово је у суштини прилично лако употребом вашег омиљеног Перфорс клијента; једноставно мапирајте `//.git-fusion` директоријум на Перфорс серверу у свој радни простор. Структура фајлова изгледа овако:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
|
└── p4gf_config
    ├── repos
    │   └── Talkhouse
    │       └── p4gf_config
    └── users
        └── p4gf_usermap
```

498 directories, 287 files

Директоријум **objects** интерно користи програм *Git Fusion* да мапира Перфорс објекте у Гит и обратно, нема потребе да се тамо петљате са било чиме. У овом директоријуму постоји глобални **p4gf_config** фајл, као и по један са сваки репозиторијум – ово су конфигурациони фајлови који одређују како се програм *Git Fusion* понаша. Хајде да погледамо фајл у корену:

```
[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
```

Овде нећемо улазити у значење ових заставица, али приметите да је ово само текстуални фајл у INI формату, сличан ониме који и програм Гит користи за конфигурацију. Овај фајл наводи глобалне опције које затим могу да се преиначе у конфигурационим фајловима одређеног репозиторијума, као што је [repos/Talkhouse/p4gf_config](#). Ако отворите овај фајл, видећете **[@repo]** одељак са неким подешавањима која се разликују од глобалних подразумеваних. Такође чете видети и одељак који изгледа овако:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Ово је мапирање између Перфорс гране и Гит гране. Име одељка може бити произвољно, докле год је јединствено. **git-branch-name** вам омогућава да конвертујете путању депоа која би била незгодна за употребу у програму Гит на неко згодније име. Поставка **view** контролише како се Перфорс фајлови мапирају у Гит репозиторијум, користећи стандардну синтаксу мапирања погледа. Може да се наведе више од једног мапирања, као у следећем

примеру:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

На овај начин, ако ваше уобичајено мапирање радног простора укључује измене структуре директоријума, то можете пресликати и у Гит репозиторијум.

Последњи фајл о коме ћемо причати је [users/p4gf_usermap](#) који мапира Перфорс кориснике на Гит кориснике и који вам можда никада неће бити ни потребан. Када се Перфорс скуп измена конвертује у Гит комит, подразумевано понашање програма *Git Fusion* је да потражи Перфорс корисника и употреби тамо сачувану имејл адресу и пуно име за поље аутор/комитер у програму Гит. Када се конвертује у другу страну, подразумевано се претражује Перфорс корисник са имејл адресом сачуваном у Гит пољу аутор комита, па се скуп измена подноси као тај корисник (уз примену дозвола). У већини случајева, ово понашање је сасвим у реду, али размотрите следећи фајл мапирања:

```
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"
```

Свака линија је у формату `<корисник> <имејл> "<пуно име>"`, и креира једно мапирање корисника. Прве две линије мапирају две различите имејл адресе у исти Перфорс кориснички налог. Ово је корисно ако сте креирали Гит комитове користећи неколико различитих имејл адреса (или променили имејл адресе), али желите да се све мапирају у истог Перфорс корисника. Када се из Перфорс скупа измена креира Гит комит, прва линија која се подудара са Перфорс корисником ће се употребити за Гит информације о ауторству комита.

Последње две линије маскирају стварна имена и имејл адресе Боба и Џоа у креираним Гит комитовима. Ово је лепо ако желите да отворите код неког интерног пројекта, али не желите да целом свету објавите свој директоријум запослених. Приметите да би имејл адресе и пуна имена требало да буду јединствени, осим ако не желите да се Гит комитови приписују једном фиктивном аутору.

Процес рада

Перфорс *Git Fusion* је двосмерни мост између Перфорс и Гит контроле верзија. Хајде да погледамо како изгледа рад са Гит стране. Претпоставићемо да смо мапирали „Jam“ пројекат користећи конфигурациони фајл као што је показано изнад и који можемо да клонирамо на следећи начин:

```

$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest 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.
[...]

```

Када по први пут урадите ово, може да потраје. Оно што се дешава је да *Git Fusion* конвертује све примењиве скупове измена из Перфорс Историје у Гит комитове. Ово се ради локално на серверу, тако да је релативно брзо, али ако имате доста историје, ипак ће потрајати. Наредна преузимања врше инкременталну конверзију, тако да ће личити на природну Гит брзину.

Као што видите, наш репозиторијум изгледа потпуно исто као и било који други Гит репозиторијум са којим можете да радите. Постоје три гране, и Гит је лепо креирао локалну `master` грану која прати `origin/master`. Хајде да урадимо мало посла и креирати неколико нових комитова:

```

# ...
$ 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.
[...]

```

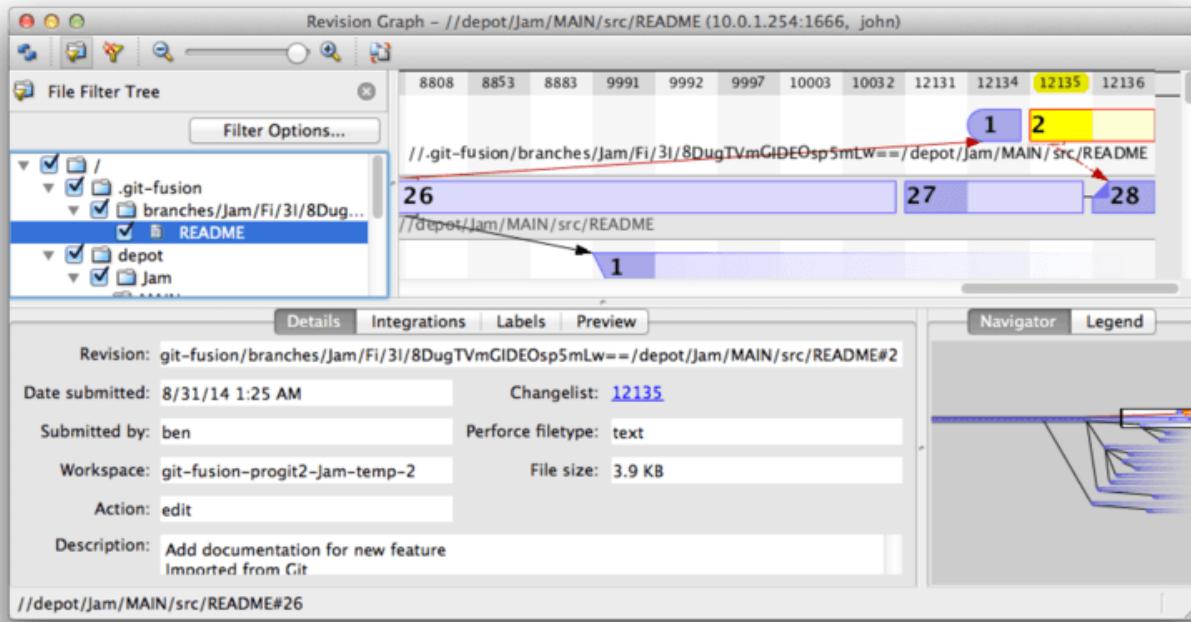
Имамо два нова комита. Хајде сада да видимо да ли је још неко радио:

```
$ 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.
[...]
```

Изгледа да јесте! Не бисте то знали из овог погледа, али **6afeb15** је уствари креиран Перфорс клијентом. Изгледа као још један комит из перспективе програма Гит, што и јесте поента. Хајде да видимо како се Перфорс сервер носи са комитом спајања:

```
$ 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
```

Програм Гит сматра да је све прошло успешно. Хајде да погледамо историју **README** фајла из перспективе програма Перфорс, користећи граф ревизија могућност команде **p4v**:



Слика 146. Перфорс граф ревизија као резултат извршења Гит команде push

Ако овај поглед раније нисте видело, може вас донекле збунити, али он приказује исти концепт као и графички приказ Гит историје. Посматрамо историју README фајла, тако да стабло директоријума горе лево приказује само тај фајл како се појављује у различитим гранама. Горе десно имамо визуелни граф веза између различитих ревизија, а доле десно приказ овог графа у целости. Остатак погледа приказује детаље изабране ревизије (у овом случају 2).

Једна ствар коју треба приметити је да граф изгледа потпуно исто као онај из Гит историје. Перфорс није имао именоване гране у које да смести комитове 1 и 2, па је креирао грану „anonymous” у .git-fusion директоријуму и сместио их у њу. Ово ће се десити и именованим Гит гранама које не одговарају именованој Перфорс грани (а касније можете да их мапирате у Перфорс грани помоћу конфигурационог фајла).

Већина овога се догађа у позадини, али крајњи резултат је да једна особа у тиму може да користи Гит, друга може да користи Перфорс, а ниједна од њих неће знати шта је избор оне друге.

Git-Fusion резиме

Ако имате (или можете добити) приступ свом Перфорс серверу, *Git Fusion* је одличан начин да Гит и Перфорс разговарају међусобно. Потребно је мало конфигурисања, али крича учења није много стрма. Ово је један од малобројних одељака у овом поглављу где се упозорења о употреби пуне снаке програма Гит не појављују. Овим не желимо да кажемо како ће Перфорс бити срећан са свим што му баците – ако покушате да поново испишете историју која је већ гурнута, *Git Fusion* ће одбити – али *Git Fusion* се заиста труди да осећај коришћења буде природан. Можете чак да користите и Гит подмодуле (који ће Перфорс корисницима изгледати чудно) и да спајате гране (ово ће са Перфорс стране бити забележено као интеграција).

Ако администратора свог сервера не можете убедити да постави *Git Fusion*, и даље постоји

начин да ова два алата користите заједно.

Git-p4

Git-p4 је двосмерни мост између програма Гит и Перфорс. Потпуно се извршава у вашем Гит репозиторијуму, тако да вам не треба никакав приступ Перфорс серверу (осим корисничких акредитива, наравно). *Git-p4* није тако флексибилно или комплетно решење као *Git Fusion*, али омогућава да урадите већину онога што бисте желели да урадите, а да не залазите у окружење сервера.



Да бисте радили са *git-p4*, потребно је да негде на **PATH** имате алат **p4**. У време писања, он је слободно доступан на адреси <http://www.perforce.com/downloads/Perforce/20-User>.

Подешавање

У сврху примера, извршаваћемо Перфорс сервер из *Git Fusion* OVA (виртуелне машине) као што је приказано изнад, али ћемо прескочити *Git Fusion* сервер и директно прећи на Перфорс контролу верзија.

Да бисте користили **p4** клијента из командне линије (од којег *git-p4* зависи), потребно је да поставите неколико променљивих окружења:

```
$ export P4PORT=10.0.1.254:1666  
$ export P4USER=john
```

Први кораци

Као што је и обичај у програму Гит, прва команда је клонирање:

```
$ 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
```

Ово креира оно што се у Гит терминологији назива „shallow” (плитки) клон; у Гит се увози само најновија Перфорс ревизија; упамтите, програм Перфорс није дизајниран тако да сваком кориснику достави сваку ревизију. Ово је довољно да се програм Гит користи као Перфорс клијент, али за остале сврхе није довољно.

Када се заврши, имамо потпуно функционални Гит репозиторијум:

```
$ 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
```

Приметите да за Перфорс сервер постоји „р4” удаљени репозиторијум, али све остало изгледа као стандардни клон. То уствари и није тачно, јер тамо заправо нема никаквог удаљеног репозиторијума.

```
$ git remote -v
```

У овом репозиторијуму нема ниједног удаљеног репозиторијума. *Git-p4* је креирао неке референце које представљају стање сервера и команди `git log` оне изгледају као удаљене референце, али њима не управља сам програм Гит и не можете да гурнете на њих.

Процес рада

У реду, хајде да одрадимо неки посао. Претпоставимо да сте постигли неки напредак на веома важној могућности и спремни сте да их покажете остатку свог тима.

```
$ 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
```

Направили смо два нова комита које желимо да проследимо Перфорс серверу. Хајде да проверимо да ли је још неко радио данас:

```
$ 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
```

Изгледа као да се се измене и `master` и `p4/master` разишли. Систем грана програма Перфорс *упуште* не личи на систем програма Гит, тако да достављање комитова спајања нема никаквог смисла. *Git-p4* препоручује да ребазирате своје комитове, па чак обезбеђује и пречицу за то:

```
$ 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(-)
```

Вероватно можете видети у излазу, `git p4 rebase` је пречица за `git p4 sync` иза које следи `git rebase p4/master`. У суштини је мало паметније од тога, посебно када се ради са више грана, али ово је добра апроксимација.

Сада је наша историја поново линеарна, па смо спремни да измене допринесемо назад на Перфорс. Команда `git p4 submit` ће покушати да креира нову Перфорс ревизију за сваки Гит комит између `p4/master` и `master`. Када је покренете прелазите у свој омиљени едитор и садржај фајла који вам се приказује изгледа отприлике овако:

```
# 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

-

+

Jam/MR,

a software build tool.

</td>

Ово је углавном исти садржај који бисте видели и да извршите **p4 submit**, осим дела на крају који је *git-p4* згодно убацио. Када треба да достави име за комит или скуп измена *Git-p4*

покушава да поштује појединачно ваша Гит и Перфорс подешавања, али у неким случајевима бисте то пожелели да преиначите. На пример, ако је Гит комит који увозите написао сарадник који нема Перфорс кориснички налог, ипак желите да крајњи скуп измена изгледа тако да га је написала та особа (а не ви).

Git-p4 је корисно увезао поруку из Гит комита као садржај овог Перфорс скупа измена, па све што је потребно да се уради јесте да фајл сачувамо и напустимо едитор, двапут (по једном за сваки комит). Коначни излаз у љуски ће изгледати слично следћем:

```
$ 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
```

Резултат је исти као да смо само извршили `git push`, што и јесте најприближнија аналогија ономе што се заиста одиграло.

Приметите да се током овог процеса сваки Гит комит претворио у Перфорс скуп измена; ако желите да их згњечите у један једини скуп измена, урадите то интерактивним ребазирањем пре него што извршите `git p4 submit`. Такође приметите да су се SHA-1 хешеви свих комитова који су достављени као скупови измена променили; то је зато што *git-p4* додаје линију на крај сваког комита који конвертује:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3eb6b9aa53145
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]
```

Шта се дешава ако покушате да доставите комит спајања? Хајде да пробамо. Ево ситуације у којој смо се нашли:

```
$ 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
```

Гит и Перфорс историје су се разишли након 775a46f. Гит страна има два комита, па затим комит спајања са Перфорс главом, затим још један комит. Покушаћемо да их доставимо као један једини скуп измена на Перфорс страну. Хајде да видимо шта би се догодило ако сада покушамо да доставимо:

```
$ 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
```

Заставица **-n** је скраћеница за **--dry-run**, а то покушава да извести шта би се догодило ако би се команда достављања заиста извршила. У овом случају изгледа да би се креирала три Перфорс скупа измена, што се подудара са три комитова који нису спајање и који још увек не постоје на Перфорс серверу. То звучи управо као оно што желимо, па хајде да видимо како ће да испадне:

```
$ git p4 submit
[...]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Наша историја је постала линеарна, као да смо је ребазирали пре подношења (што се у суштини заиста и десило). Ово значи да на Гит страни слободно можете да креирате гране, радите на њима, бришете их и спајате без страха да ће ваша историја на неки начин постати некомпабилна са програмом Перфорс. Ако можете да је ребазирате, можете и да је пошаљете на Перфорс сервер.

Гранање

Ако ваш Перфорс пројекат има више грана, ипак имате среће; *git-p4* то може да обради на начин да којим се чини као да је природно у програму Гит. Рецимо да је ваш Перфорс депо постављен на следећи начин:

```
//depot
  └── project
      ├── main
      └── dev
```

И рецимо да имате **dev** грану која поседује следећу спецификацију погледа:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 може аутоматски да детектује такву ситуацију и уради одговарајућу ствар:

```
$ 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
| * a43cfaf Populate //depot/project/main/... //depot/project/dev/....
|
* 2b83451 Project init
```

Приметите спецификатор „@all” у путањи депоа; он говори *git-p4* да клонира не само последњи скуп измена за то подстабло, већ и све скупове измена које су икада дотакле те путање. То је приближније концепту клонирања у програму Гит, али ако радите на пројекту са дугачком историјом, могло би да потраје.

Заставица **--detect-branches** налаже да *git-p4* искористи Перфорс спецификације гране за мапирање грана у Гит референце. Ако се ова мапирања не налазе на Перфорс серверу (што је савршено исправан начин да се користи Перфорс), можете сами навести *git-p4* мапирања грана, па добијате исти 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 .
```

Постављање **git-p4.branchList** конфигурационе променљиве на **main:dev** говори команди *git-p4* да су и „main” и „dev” гране, као и да је друга дете прве.

Ако сада извршимо **git checkout -b dev p4/project/dev** и направимо неколико комитова, *git-p4* је доволно паметна да циља на праву грану када извршимо **git p4 submit**. Нажалост, *git-p4* не може да меша плитке клонове и вишеструке гране; ако имате огроман пројекат и желите да радите на више од једне гране, мораћете да извршите **git p4 clone** по једном за сваку грану на коју желите да достављате измене.

За креирање или интегрисање грана мораћете да користите Перфорс клијент. *Git-p4* може само да синхронизује и доставља на постојеће гране и то само као један линеарни скуп измена одједном. Ако у програму Гит спојите две гране и покушате да доставите нови скуп измена, све што ће се забележити биће гомила измена над фајловима; изгубиће се сви метаподаци о гранама које су умешане у интеграцију.

Гит и Перфорс резиме

Git-p4 омогућава да се са Перфорс сервером употребљава Гит процес рада, и прилично је добра у томе. Међутим, важно је упамтити да је Перфорс управља извornим кодом, а ви програм Гит користите само за рад у локалу. Само будите врло пажљиви са дељењем Гит комитова; ако имате удаљени репозиторијум који користе и други људи, не гурајте било које комитове који већ нису достављени Перфорс серверу.

Ако желите да слободно мешате употребу програма Перфорс и програма Гит као клијената за контролу извornог кода, а администратора сервера можете убедити да га инсталира, *Git Fusion* чини да Гит постане првокласни клијент Перфорс сервера за контролу верзија.

Миграње на Гит

Ако имате постојећу базу кода у неком другом VCS или сте одлучили да почнете користити програм Гит, морате да миграјете пројекат на један или други начин. Овај одељак представља неке увознике за уобичајене системе, па онда приказује како можете да израдите свој сопствени. Научићете како да увезете податке из неколико других већих SCM који се професионално користе, зато што њих користи већина корисника који желе да начине промену и због тога што је једноставно доћи до високо квалитетних алата за њих.

Subversion

Ако сте читали претходни одељак о коришћењу `git svn`, та упутства можете лако да следите и направите `git svn clone` репозиторијума; па затим престанете да користите Subversion сервер, гурнете на нови Гит сервер и почнете то да користите. Ако вам је потребна историја, добићете је онолико брзо колико је потребно да повучете податке са Subversion сервера (што може да потраје).

Међутим, увоз није савршен; а пошто ће потрајати прилично дugo, такође можете да га урадите и како треба. Први проблем су информације о аутору. У програму Subversion, свака особа која комитује има корисничко име на систему које се бележи у комит информацијама. Примери у претходном одељку приказују на неким местима `schacon`, као што је излаз команди `blame` и `git svn log`. Ако ово желите да мапирате у боље Гит податке о аутору, потребно вам је мапирање из Subversion корисника на Гит ауторе. Креирајте фајл под именом `users.txt` који ово мапирање има у следећем формату:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Да бисте добили листу имена аутора које користи SVN, извршите следеће:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*/>(.*)<.*/$1 = /'
```

Ово генерише лог излаз у XML формату, затим задржава само линије са информацијама о аутору, одбацује дупликате и уклања XML ознаке. Очигледно је да функционише само на

машини са инсталлираним `grep`, `sort`, и `perl`. Затим, преусмерава узлаз у ваш `users.txt` фајл тако да уз сваку ставку можете додати податке еквивалентног Гит корисника.



Ако ово пробате на Виндоуз машини, на овом месту ћете наићи на проблем.

Мајкрософт је понудио неколико добрих савета и примера на адреси <https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git>.

Овај фајл можете доставити команди `git svn` и помогнете јој да прецизније мапира податке о аутору. `git svn` такође можете наложити да не укључи метаподатке које Subversion обично увози, ако команди `clone` или `init` проследите `--no-metadata`. Метаподаци садрже `git-svn-id` у свакој комит поруци коју ће програм Гит генерисати током увоза. То ће да претрпа ваш Гит лог, па може постати донекле нејасан.



Ако комитове направљене у Гит репозиторијуму желите да пресликате назад у оригинални SVN репозиторијум, мораћете да очувате метаподатке.

Ако у свом логу комитова не желите синхронизацију, слободно изоставите параметар `--no-metadata`.

Тако да ваша `import` команда изгледа овако:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
  --authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

Сада би у `my_project` директоријуму требало да имате лепше увезене Subversion податке. Уместо да комитови изгледају на следећи начин:

```
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
```

они изгледају овако:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:  Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk
```

Не само да поље *Author* изгледа много боље, већ се ту више не налази ни `git-svn-id`.

Такође би требало мало и да почистите након увоза. Било би добро да почистите чудне референце које је поставила `git svn`. Најпре ћете преместити ознаке тако заиста постану ознаке, е не чудне удаљене гране, па ћете затим преместити остatak грана тако да постану локалне.

Да би ознаке постале прописне Гит ознаке, извршите:

```
$ 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
```

Ово узима референце које су биле удаљене гране, а почињу на `remotes/origin/tags/` и претвара их у праве (просте) ознаке.

Затим, преместите остatak референци под `refs/remotes` тако да постану локалне гране:

```
$ 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 се додати да видите неке додатне гране које имају `@xxx` суфикс (где је xxx број), док у Subversion видите само једну грану. Ово је заправо Subversion могућност под називом „рег-revisions” и програм Гит за то једноставно нема одговарајући синтаксни елемент. Стога, `git svn` просто додаје број svn верзије имену гране на исти начин као што бисте ви и ви урадили у svn како би се обратили рег-ревизији те гране. Ако више не водите рачуна о рег-ревизијама, једноставно их уклоните:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D $p; done
```

Сада су све старе гране праве Гит гране и све старе ознаке праве Гит ознаке.

Постоји још једна ствар која треба да се почиши. Нажалост, `git svn` креира додатну грану под називом `trunk` и која се мапира на подразумевану грану у Subversion, али `trunk` референца показује на исто место као и `master`. Пошто `master` више одговара Гит стилу, ево како да уклоните додатну грану:

```
$ git branch -d trunk
```

Последња ствар који треба да урадите је да свој нови Гит сервер додате као удаљени репозиторијум и да турнете на њега. Ево примера како да додате свој сервер као удаљени репозиторијум:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Пошто желите да све ваше гране и ознаке оду узводно, требало би да извршите следеће:

```
$ git push origin --all  
$ git push origin --tags
```

Сада би све ваше гране и ознаке требало да се налазе на Гит серверу након чистог увоза.

Меркуријал

Пошто Меркуријал и Гит имају прилично сличне моделе за представљање верзија, а како је програм Гит мало флексибилнији, конвертовање репозиторијума из Меркуријал у Гит је прилично лако, употребом алата под називом *hg-fast-export*, који прво треба да преузмете:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

Први корак конверзије је да преузмете потпуни клон Меркуријал репозиторијума који желите да конвертујете:

```
$ hg clone <URL адреса удаљеног репозиторијума> /tmp/hg-геро
```

Наредни корак је креирање фајла за мапирање аутора. Меркуријал није толико стриктан као Гит у вези тога шта ће ставити у поље аутора скупа измена, тако да је ово добар тренутак да мало поспремите кућу. Генерирање овог фајла је једнолинијска команда у *bash* љуски:

```
$ cd /tmp/hg-геро  
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Ово ће потрајати неколико секунди, зависно од величине историје пројекта, па ће након тога фајл */tmp/authors* изгледати отприлике овако:

```
bob  
bob@localhost  
bob <bob@company.com>  
bob jones <bob <AT> company <DOT> com>  
Bob Jones <bob@company.com>  
Joe Smith <joe@company.com>
```

У овом примеру, иста особа (Bob) је креирала скупове измена користећи четири различита имена, од којих једно изгледа коректно, а једно би било потпуне неисправно за Гит комит. *Hg-fast-export* вам омогућава да ово исправите тако што сваку линију претворите у правило: "*<указ>=<изказ>*", које мапира *<указ>* у *<изказ>*. Унутар стрингова *<указ>* и *<изказ>*, подржавају се сви означени низови које препознаје Пајтон *string_escape* кодирање. Ако фајл мапирања аутора не садржи одговарајући *<указ>*, онда се тај аутор шаље програму Гит неизмењен. Ако сва корисничка имена изгледају како треба, нема потребе да се овај фајл

исправља. У овом примеру, желимо да наш фајл изгледа на следећи начин:

```
"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>"
```

Наредни корак је да креирамо наш нови Гит репозиторијум и да извршимо скрипту за извоз:

```
$ git init /tmp/converted  
$ cd /tmp/converted  
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Заставица **-r** говори *hg-fast-export* где да пронађе Меркуријал репозиторијум који желимо да конвертујемо, а заставица **-A** говори где да пронађе фајл са мапирањима аутора. Скрипта парсира Меркуријал скупове измена и конвертује их у скрипту за „fast-import” могућност програма Гит (коју ћемо детаљно приказати мало касније). Ово ће потрајати (мада је *много* брже него што би било преко мреже), а излаз је прилично детаљан:

```

$ /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

```

И то је скоро све у вези конверзије. Све Меркуријал ознаке су конвертоване у Гит ознаке, а Меркуријал гране и маркери су конвертовани у Гит гране. Сада сте спремни да гурнете репозиторијум узводно на његово ново место на серверу:

```
$ git remote add origin git@my-git-server:myrepository.git  
$ git push origin --all
```

Базаар

Базаар је DVCS алат који веома личи на Гит, па је стога прилично једноставно конвертовати Базаар репозиторијум у Гит. Да бисте то постигли, потребан вам је **bzr-fastimport** додатак.

Пријемање **bzr-fastimport** додатка

Процедура за инсталацију *fastimport* додатка се разликује на Виндууз и системима сличним Јуниксу. У другом случају је најједноставније да инсталирате **bzr-fastimport** пакет који ће инсталирати и све неопходне зависности.

На пример, на Дебијан и системима изведеним из њега, урадили бисте следеће:

```
$ sudo apt-get install bzr-fastimport
```

На RHEL, урадили бисте следеће:

```
$ sudo yum install bzr-fastimport
```

На Федори, почевши од издања 22, нови менаџер пакета је *dnf*:

```
$ sudo dnf install bzr-fastimport
```

Ако пакет није доступан, можете да га инсталирате као додатак:

```
$ mkdir --parents ~/.bazaar/plugins      # creates the necessary folders for the  
plugins  
$ cd ~/.bazaar/plugins  
$ bzr branch lp:bzr-fastimport fastimport  # imports the fastimport plugin  
$ cd fastimport  
$ sudo python setup.py install --record=files.txt  # installs the plugin
```

Да би овај додатак радио, такође вам је потребан и **fastimport** Пајтон модул. Следећим командама можете проверити да ли је присутан или не и инсталирати га ако треба:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

Ако није доступан, преузмите га са адресе <https://pypi.python.org/pypi/fastimport/>.

У првом случају (на Виндоуз систему), **bzr-fastimport** се автоматски инсталира са самосталном верзијом и подразумеваном инсталацијом (оставите штиклирана сва поља). Тако да у овом случају није потребно да било шта урадите.

Сада се начин увоза Базаар репозиторијума разликује у зависности од тога да ли имате једну грану или радите са репозиторијумом који има неколико грана.

Пројекти са једном граном

Извршите **cd** у директоријум који садржи ваш Базаар репозиторијум и иницијализујте Гит репозиторијум:

```
$ cd /path/to/the/bzr/repository
$ git init
```

Сада једноставно извезете свој Базаар репозиторијум и конвертујете га у Гит репозиторијум следећом командом:

```
$ bzr fast-export --plain . | git fast-import
```

Зависно од величине пројекта, ваш Гит репозиторијум се изграђује у року од неколико секунди до неколико минута.

Случај пројекта са главном граном и радним гранама

Такође можете да уvezете и Базаар репозиторијум који садржи гране. Претпоставимо да имате две гране: једна представља главну грану (`myProject/trunk`), а друга је радна грана (`myProject/work`).

```
$ ls
myProject/trunk myProject/work
```

Креирајте Гит репозиторијум и извршите **cd** у њега:

```
$ git init git-repo
$ cd git-repo
```

Повуците `master` грану у Гит:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject/trunk | \
git fast-import --export-marks=../marks.git
```

Повуците радну грану у Гит:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

Сада вам команда `git branch` приказује `master` грану као и `work` грану. Проверите логове да потврдите јесу ли комплетни и решите се `marks.bzr` и `marks.git` фајлова.

Синхронизација стејџа

Колико год грана да сте имали и који год метод за увоз употребили, ваш стејџ се не синхронизује са `HEAD`, а увозом неколико грана се чак ни ваш радни директоријум не синхронизује. Ситуација се једноставно решава следећом командом:

```
$ git reset --hard HEAD
```

Игнорисање фајлова који су игнорисани са `.bzrignore`

Хајде сада да погледамо како се игноришу фајлови. Прва ствар коју треба да урадите је да преименујете `.bzrignore` у `.gitignore`. Ако `.bzrignore` фајл садржи једну или неколико линија које почињу са „`!!`” или „`RE:`”, мораћете да измените и можда креирате и неколико `.gitignore` фајлова како би се игнорисали потпуно исти фајлови које је игнорисао и програм Базаар.

Коначно, мораћете да креирате комит који садржи ову измену за миграцију:

```
$ git mv .bzrignore .gitignore
$ # modify .gitignore if needed
$ git commit -am 'Migration from Bazaar to Git'
```

Слање вашег репозиторијума на сервер

Ево нас! Сада репозиторијум можете да гурнете на његов нови сервер:

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

Ваш Гит репозиторијум је спреман за употребу.

Перфорс

Наредни систем за који ћете видети процедуру увоза је Перфорс. Као што смо разматрали изнад, постоји два начина да Гит и Перфорс комуницирају међусобно: *git-p4* и *Perforce Git Fusion*.

Perforce Git Fusion

"Git Fusion" чини овај процес прилично безболним. Једноставно конфигуришете поставке вашег пројекта, мапирања корисника и гране користећи конфигурациони фајл (као што је разматрано у [Git Fusion](#)), и клонирате репозиторијум. *Git Fusion* вам оставља оно што изгледа као природни Гит репозиторијум, који затим можете да гурнете на природни Гит хост ако желите. Могли бисте чак да користите и Перфорс као свој Гит гост ако то желите.

Git-p4

Git-p4 такође може да се понаша као алат за увоз. Као пример, извршићемо увоз *Jam* пројекта из Перфорс јавног депоа. Да бисте подесили свој клијент, прво морате да извезете променљиву окружења P4PORT тако да показује на Перфорс депо:

```
$ export P4PORT=public.perforce.com:1666
```



Да бисте могли пратити пример, потребан вам је Перфорс депо на који можете да се повежете. У нашим примерима ћемо користити јавни депо на адреси public.perforce.com, али можете да користите било који депо на које имате приступ.

Извршите команду `git p4 clone` да уvezете *Jam* пројекат са Перфорс сервера, наводећи депо и путању пројекта, као и путању у коју желите да уvezете пројекат:

```
$ 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%)
```

Овај пројекат има само једну грану, али ако имате гране конфигурисане да погледима на гране (или само скуп директоријума), можете да употребите заставицу `--detect-branches` уз команду `git p4 clone` да такође уvezете и све гране пројекта. За мало више детаља у вези овога, погледајте [Гранање](#).

У овом тренутку сте скоро готови. Ако одете у `p4import` директоријум и извршите `git log`, видећете ваш уvezени рад:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

Correction to line 355; change to .

```
[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]
```

Можете видети да је **git-p4** оставила идентификатор у свакој комит поруци. Нема проблема да се тај идентификатор тамо задржи, у случају да вам је касније потребна референца на Перфорс број измене. Међутим, ако желите да уклоните идентификатор, сада је право време за то – пре него што почнете рад на репозиторијуму. За групно уклањање идентификатора можете употребити команду **git filter-branch**:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Ако извршите **git log** видећете да се измениле све SHA-1 контролне суме комитова, али да се **git-p4** стрингови више не налазе у комит порукама:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

Correction to line 355; change to .

```
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).

Ваш увоз је спреман да се гурне на нови Гит сервер.

Прилагодљиви увозник

Ако ваш систем није ниједан од горе наведених, требало би да на мрежи потражите алат за увоз – доступни су квалитетни алати за увоз из многих других система, укључујући CVS, Clear Case, Visual Source Safe, чак и директоријум архива. Ако ниједан од њих не ради у вашем случају, имате прилично непознат алат, или вам је потребан још више прилагођени процес увоза, требало би да искористите `git fast-import`. Ова команда чита једноставне инструкције са `stdin` за уписивање одређених Гит података. На овај начин је много једноставније креирати Гит објекте него да извршавате сирове Гит команде, или да покушате уписивање сирових објеката (за више информација, погледајте [Гит изнутра](#)). На овај начин можете написати скрипту за увоз која чита неопходне информације из система који увозите и пише јасне инструкције на `stdout`. Затим можете да покренете овај програм и спроведете његов излаз кроз `git fast-import`.

Да бисмо брзо показали, написаћете једноставан алат за увоз. Претпоставимо да радите у `curren`, повремено правите резервну копију свог пројекта копирањем директоријума у временски обележен `back_YYYY_MM_DD` директоријум који чува резерве и желите да то увезете у програм Гит. Структура ваших директоријума изгледа овако:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
curren
```

Да бисте увезли Гит директоријум, морате знати како програм Гит чува своје податке. Вероватно се сећате, Гит је у основи увезана листа комит објеката који показују на снимак садржаја. Све што треба да урадите је да команди `fast-import` кажете шта су снимци садржаја, који комит подаци указују на њих, и у ком редоследу се јављају. Ваша стратегија ће бити да пролазите кроз снимке један по један и да креирате комитове са садржајем сваког директоријум, повезујући сваки комит са претходним.

Као што смо урадили у [Пример полисе коју спроводи програм Гит](#), написаћемо то у Рубију, јер у општем случају радимо на том језику и углавном може лако да се чита. Овај пример прилично лако можете да напишете на било ком језику који вам је близак – он само мора да испиши одговарајуће информације на `stdout`. А ако извршавате на Виндоуз систему, то значи да посебно морате пазити да не уведете *carriage returns* карактере на крај редове – `git fast-import` стриктно жели само *line feed (LF)* карактере, а не *carriage return line feed (CRLF)* комбинацију коју користи Виндоуз.

За почетак, прећи ћете у циљни директоријум и идентификовачете свако поддиректоријум, од којих је сваки снимак који желите да увезете као комит. Ући ћете у свако поддиректоријум и исписаћете команде које су неопходне да се извезе. Ваша основна главна петља изгледа овако:

```
last_mark = nil

# прођи кроз све директоријуме
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # уђи у циљни директоријум
        Dir.chdir(dir) do
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

Унутар сваког директоријума сте извршили функцију `print_export`, која узима манифест и маркер претходног снимка и враћа манифест и маркер тренутног; на тај начин можете исправно да их увежете. „Маркер” је термин команде `fast-import` за идентификатор који постављате сваком комиту; док креирате комитове, сваком додељујете маркер који употребљавате да до њега дођете од осталих комитова. Дакле, прва ствар коју треба да урадите у својој `print_export` методи је да из имена директоријума генеришете маркер:

```
mark = convert_dir_to_mark(dir)
```

Ово ћете урадити тако што ћете направити низ директоријума и употребити вредност индекса као маркер, јер маркер мора бити целобројна вредност. Ваша метода изгледа овако:

```
$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir) + 1).to_s
end
```

Сада када имате целобројну представу вашег комита, потребан вам је датум за метаподатке комита. Пошто је датум део имена директоријум, парсираћете га из њега. Следећа линија у вашем `print_export` фајлу је:

```
date = convert_dir_to_date(dir)
```

где је `convert_dir_to_date` дефинисана као:

```
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
```

Ово враћа целобројну вредност за датум сваког директоријума. Последњи део метаподатака који вам је потребан за сваки комит су подаци о комитеру, које једноставно фиксирате у глобалној променљивој:

```
$author = 'John Doe <john@example.com>'
```

Сада сте спремни да почнете испис комит података вашем алату за увоз. Почетна информација наводи дефинишете комит објекат и грану на којој се налази, иза чега следи маркер који сте генерисали, информације о комитеру и комит порука, па затим претходни комит, ако постоји. Код изгледа овако:

```
# 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
```

Временску зону фиксирате (-0700) јер је тако једноставно да се изведе. Ако увозите из неког другог система, временску зону морате да наведете као померај (у односу на зону у којој је ваш систем). Комит порука мора да се наведе у посебном формату:

```
data (size)\n(contents)
```

Формат се састоји из речи *data*, величине података који треба да се прочитају, прелома реда и коначно самих података. Пошто је касније потребно да исти формат употребите за навођење фајла са садржајем, креираћете помоћну методу, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Преостало је само да за сваки снимак наведете садржај. То је једноставно јер се сваки налази у посебном директоријуму – можете да га испишете `deleteall` командом након које следи

садржај сваког фајла у директоријуму. Програм Гит онда на одговарајући начин бележи сваки снимак:

```
puts 'deleteall'  
Dir.glob("**/*").each do |file|  
  next if !File.file?(file)  
  inline_data(file)  
end
```

Напомена: пошто многи системи на своје ревизије гледају као на промене од једног на други комит, **fast-import** такође може да прими и команде уз сваки комит које наводе који фајлови су додати, уклоњени или изменењени, као шта је нови садржај. Израчунали бисте разлике између снимака и доставили само те податке, али то је много компликованије – исто тако можете програму Гит да доставите све податке, па да њему препустите да одреди шта и како. Ако је тако погодније за ваше податке, проверите **fast-import** ман страницу у вези детаља о томе како да податке доставите на овај начин.

Формат за испис садржаја новог фајла или за навођење изменењеног фајла са новим садржајем је као што следи:

```
M 644 inline путања/до/фајла  
data (величина)  
(садржај фајла)
```

Овде 644 представља режим (ако имате извршни фајл, морате то да откријете и да уместо овога наведете 755), а *inline* наводи да садржај следи непосредно након ове линије. Ваша **inline_data** метода изгледа на следећи начин:

```
def inline_data(file, code = 'M', mode = '644')  
  content = File.read(file)  
  puts "#{code} #{mode} inline #{file}"  
  export_data(content)  
end
```

Поново искоришћавате **export_data** методу коју сте дефинисали раније, јер је начин исти као онај на који сте навели податке комит поруке.

Последње што треба да урадите је да вратите текући маркер, тако да се може проследити у наредну итерацију:

```
return mark
```



Ако користите Виндоуз биће вам потребан још један корак. Као што смо поменули раније, Виндоуз користи CRLF карактере као прелом реда, док команда `git fast-import` очекује само LF карактер. Да бисте решили овај проблем и усређили команду `git fast-import`, потребно је да Рубију наложите да уместо CRLF користи LF:

```
$stdout.binmode
```

То је то. Ево како изгледа комплетна скрипта:

```
#!/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 извршите ову скрипту, добићете садржај који отприлике изгледа овако:

```
$ 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
(...)
```

Да бисте покренули алат за увоз, проследите овај излаз команди `git fast-import` док се налазите у Гит директоријуму у који желите да увезете податке. Можете да креирате нови директоријум, па да у њему као први корак извршите `git init`, а затим покренете своју скрипту:

```

$ 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
-----
```

Као што можете видети, када се процес успешно заврши, враћа вам гомилу статистичких података о ономе што је урађено. У овом случају, увезли сте укупно 13 објеката за 4 комита у 1 грану. Сада можете да извршите `git log` и погледате своју нову историју:

```

$ 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
```

Ево финог чистог Гит репозиторијума. Важно је приметити да се ништа не одјављује – у почетку немате ниједан фајл у свом радном директоријуму. Да бисте их добили, своју грану најпре морате да ресетујете на место на којем се сада налази `master`:

```
$ ls  
$ git reset --hard master  
HEAD is now at 3caa046 imported from current  
$ ls  
README.md main.rb
```

Алатом `fast-import` можете да урадите још много тога – обрађујете различите режиме, бинарне податке, вишеструке гране и спајања, ознаке, индикаторе напретка процеса и још тога. Већи број примера компликованијих сценарија можете да погледате у `contrib/fast-import` директоријуму извornog кода програма Гит.

Резиме

Требало би да сада можете комотно да користите Гит као клијент других система за контролу верзије, или да уvezете скоро сваки постојећи репозиторијум у Гит а да не изгубите податке. У следећем поглављу ћемо погледати неке сировине од којих је Гит изграђен како бисте могли да баратате сваким бајтом, ако дође до потребе за тим.

Гит изнутра

Можда се скочили директно на ово поглавље са много ранијег, или сте можда стигли овде након што сте прочитали цelu књигу—у сваком случају, овде ћемо да погледамо како програм Гит ради изнутра и како је имплементиран. Мислимо да је учење ових аспеката програма Гит од суштинске важности за разумевање колико је програм Гит користан и моћан алат, али неки су нам предочили да почетнике могу поприлично да збуне и да су им уводе непотребну сложеност. Зато смо нашли компромис и одлучили да ово буде последње поглавље у књизи како би сте могли да га прочитате пре или касније током процеса учења. На вама је да одлучите.

Сада када сте овде, време је да кренемо. Прво, ако ово још увек није јасно, програм Гит је у основи фајл систем који се адресира по садржају преко којег је написан VCS кориснички интерфејс. Ускоро ћете научити детаљније на шта се мисли под овиме.

У раним данима програма Гит (углавном пре верзије 1.5), кориснички интерфејс је био много сложенији јер је наглашавао фајл систем а не дотеран VCS. У последњих неколико година, КИ је толико унапређен тако да је његово коришћење постало чисто и једноставно као и коришћење било ког другог система; мада често и даље кружи стереотип из раних дана КИ програма Гит о томе како је превише сложен и тежак за учење.

Слој фајл система који се адресира садржајем је невероватно кул, па ћемо га првог обрадити у овом поглављу; затим ћете научити нешто о преносним механизима и задацима везаним за одржавање репозиторијума у које ћете вероватно на kraју крајева морати да се упустите.

Водовод и порцелан

Ова књига првенствено покрива како се програм Гит користи помоћу неких тридесетак подкоманди као што су `checkout`, `branch`, `remote`, и тако даље. Али пошто је програм Гит првобитно био алат система за котролу верзије, а не потпуни VCS једноставан за употребу, он има много других подкоманди који раде ствари на ниском нивоу и које су дизајниране тако да се уланчавају у Јуникс стилу, или да се позивају из скрипти. Ове команде се обично зову „водоводне команде”, а команде које су више пријатељски настројене корисницима се зову „порцеланске команде”.

Као што сте дали сада видели, првих девет поглавља књиге бавило се скоро искључиво порцеланским командама. Али у овом поглављу ћемо се бавити углавном водоводним командама ниског нивоа, јер вам омогућавају приступ унутрашњој страни програма Гит и помажу да се покаже како и зашто програм Гит ради оно што ради. Многе од ових команди нису предвиђене да се користе директно из командне линије, већ да се користе као градивни блокови нових алата и прилагођених скрипти.

Када извршите команду `git init` у новом или постојећем директоријуму, програм Гит креира директоријум `.git` и њему се налази све што програм Гит чува и чиме манипулише. Ако желите да сачувате резервну копију репозиторијума или да га клонирате, копирање овог једног директоријума на неко друго место је скоро све што вам је потребно. Цело ово поглавље се у суштини бави стварима које се налазе у овом директоријуму. Ево како изгледа

свеже иницијализован `.git` директоријум.

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

У зависности од верзије програма Гит коју користите, можда ћете овде видети и неки други садржај, али ово је свеж `git init` репозиторијум — подразумевано би требало ово да видите. `description` фајл користи само програм *GitWeb*, тако да нема потребе да бринете о њој. Фајл `config` садржи конфигурационе опције специфичне за пројекат, а директоријум `info` чува глобални `exclude` фајл са шаблонима за игнорисање које не желите да пратите у `.gitignore` фајлу. Директоријум `hooks` садржи скрипте кука за клијентску и серверску страну, о којима се детаљно говори у [Гит куке](#).

Стога нам преостају још четири битне ставке: фајлови `HEAD` и `index` (који ће тек бити креиран), као и директоријуми `objects` и `refs`. Они представљају срж програма Гит. Директоријум `objects` чува сав садржај базе података, директоријум `refs` чува показиваче на комит објекте из тих података (гране, ознаке, удаљени и још нешто), фајл `HEAD` показује на грану која је тренутно одјављена, а фајл `index` је мести у коме програм Гит чува информације о стејџу. Сада ћемо детаљно да погледамо сваки од ових делова, како бисмо разумели начин на који програм Гит ради.

Гит објекти

Гит је фајл систем који се адресира садржајем. Одлично. Али шта то значи? Значи да је у срцу програма Гит једноставно складиште парова кључ-вредност. Можете да убаците било какав садржај у Гит репозиторијум и он ће вам вратити јединствени кључ који касније можете употребити да дођете до тог садржаја.

Да бисте ово показали, хајде да видимо водоводну команду `hash-object` која узима неке податке, чува их у директоријуму `.git/objects` (у бази података објекта) и враћа вам јединствени кључ који од сада показује на тај објекат података.

Најпре иницијализујете нови Гит репозиторијум и да се уверите да у директоријуму `objects` нема ничега (како се и очекује).

```
$ 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
```

Програм Гит је иницијализовао директоријум `objects` и поддиректоријуме `pack` и `info`, али нема никаквих обичних фајлова. Сада, хайде да употребимо `git hash-object`, креирамо нови објекат података и ручно га сместимо у нову Гит базу података:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

У својом најједноставнијем облику, команда `git hash-object` би преузела садржај који јој проследите и само вратила јединствени кључ који би се користио да се он сачува у вашу Гит базу података. Опција `-w` говори команди `hash-object` да поред тога што врати кључ, сачува и објекат у базу података. Коначно, опција `--stdin` говори команди `git hash-object` да садржај за обраду преузме са стандардног улаза; у супротном команда очекује име фајла у којем се налази садржај као последњи аргумент у командној линији.

Излаз из команде је хеш контролна сума дужине четрдесет карактера. Ово је SHA-1 хеш—контролна сума садржаја који чувате, плус заглавља, о коме ћете научити више мало касније. Сада можете да видите како је програм Гит сачувао податке:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Ако поново погледате свој `objects` директоријум, видећете да се у њему сада налази фајл за тај нови садржај. Ово је начин на који програм Гит иницијално смешта садржај — као један фајл за један део садржаја, чије име је SHA-1 контролна сума садржаја и његовог заглавља. Поддиректоријум добија име по прва два карактера SHA-1 контролне суме, а име фајла је преосталих 38 карактера.

Када се садржај нађе у вашој бази података објекта, можете га испитати командом `cat-file`. Ова команда је нешто као швајцарски војнички нож за инспекцију Гит објекта. Када јој проследите опцију `-p`, наређујете јој да најпре открије врсту садржаја о ком се ради, па да га прикаже на одговарајући начин:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Сада знате како да у програм Гит додате садржај у Гит и да га поново извучете назад. Ово

такође можете да радите и са садржајем у фајловима. На пример, можете да обавите једноставну контролу верзије над фајлом. Најпре креирајте нови фајл и сачувајте његов садржај у базу података:

```
$ echo 'version 1' > test.txt  
$ git hash-object -w test.txt  
83baae61804e65cc73a7201a7252750c76066a30
```

Затим допишите неки нов садржај у фајл, па га поново сачувайте:

```
$ echo 'version 2' > test.txt  
$ git hash-object -w test.txt  
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Ваша база података објекта сада садржи две нове верзије фајла (као и први садржај који сте у њу сачували):

```
$ find .git/objects -type f  
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

У овом тренутку можете обрисати локалну копију тог `test.txt` фајла, а затим да употребите програм Гит да из базе података објекта вратите или прву верзију коју сте сачували:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt  
$ cat test.txt  
version 1
```

или другу верзију:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt  
$ cat test.txt  
version 2
```

Али памћење SHA-1 кључа за сваку верзију фајла није практично; сем тога, у фајл систему не чувате име фајла, већ само његов садржај. Овај тип објекта се зове **блоб**. Програм Гит може да вам каже тип објекта било ког објекта у репозиторијуму, ако му уз команду `cat-file -t` задате SHA-1 кључ.

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a  
blob
```

Објекти стабла

Следећи тип Гит објекта који ћемо испитати је *стабло*, које решава проблем чувања имена фајлова, а поред тога вам дозвољава и да заједно ускладиштите групу фајлова. Програм Гит садржај чува на сличан начин као Јуникс фајл систем, али нешто једноставније. Сав садржај се састоји од објекта стабала и блобова, при чему стабла одговарају Јуникс директоријумима, а блобови су мање-више пандан и-чворовима или садржају фајлова. Један објекат стабла садржи једну или више ставки, од којих свака садржи SHA-1 хеш блоба или подстабла, уз информацију о придруженом режиму, врсти и имену фајла. На пример, рецимо да имате пројекат у којем најновије стабло изгледа некако овако:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

Синтакса `master^{tree}` наводи објекат стабла на које показује последњи комит на `master` грани. Обратите пажњу на то да поддиректоријум `lib` није блоб већ показивач на друго стабло:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

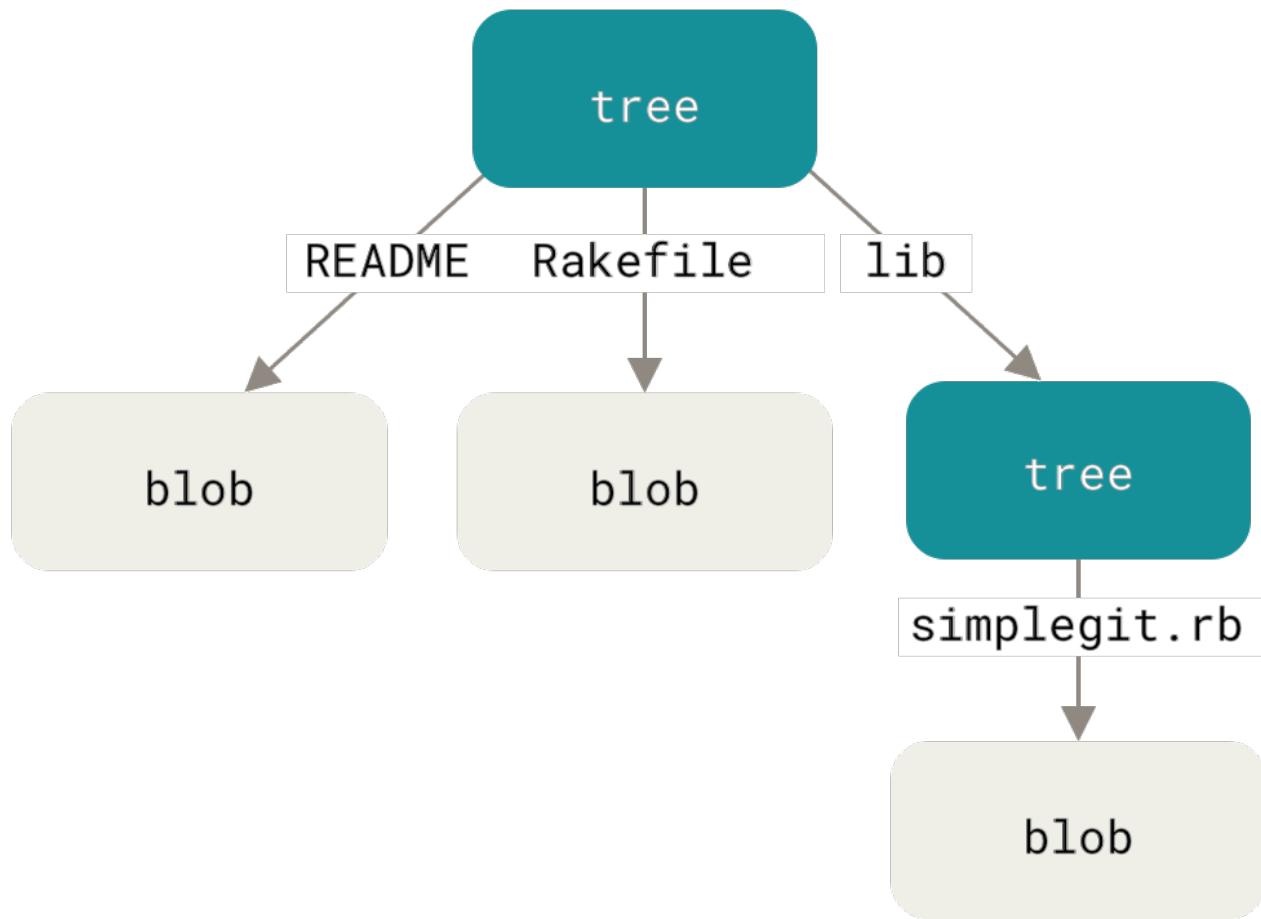
Зависно од љуске коју користите, можете наићи на проблеме при употреби `master^{tree}` синтаксе.



У *CMD* на Виндоуз систему, карактер `^` се користи за означавање, тако да га морате удвојити ако желите да избегнете проблем: `git cat-file -p master^{tree}`. Када се користи *PowerShell*, параметри који садрже `{}` карактере морају да се цитирају, јер се тако спречава погрешно парсирање параметра: `git cat-file -p 'master^{tree}'`.

Ако користите *ZSH*, карактер `^` се употребљава за *globbing* (генерисање имена фајлова и директоријума регуларним изразима), тако да комплетан израз морате да поставите унутар знакова навода: `git cat-file -p "master^{tree}"`.

Концептуално, подаци које чува програм Гит изгледају отприлике на следећи начин:



Слика 147. Једноставна верзија модела података програма Гит

Прилично лако можете направити своје сопствено стабло. Програм Гит обично креира стабло тако што узима стање стејца или индекса и на основу њега исписује низ објеката стабла. Дакле, да бисте креирали стабло, прво треба да поставите индекс тако што стејџујете неке фајлове. Да бисте креирали индекс са само једном ставком — првом верзијом фајла `test.txt` — можете да употребите водоводну команду `update-index`. Ова команду можете искористити да на стејџ вештачки додате старију верзију фајла `test.txt`. Морате да јој проследите опцију `--add` јер фајл још увек не постоји на стејџу (у овом тренутку немате ни стејџ) и `--cacheinfo` јер се фајл који додајете не налази у вашем директоријуму, већ у бази података. Затим наведете режим, SHA-1 и име фајла:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

У овом случају наводите режим `100644`, што значи да се ради о обичном фајлу. Остале опције су `100755`, што значи да се ради о извршном фајлу; и `120000`, што наводи симболички линк. Режим је преузет од уобичајених Јуникс режима али је много мање флексибилан — за фајлове (блобове) у програму Гит важе само ова три режима (мада се други режими користе за поддиректоријуме и подмодуле).

Сада можете да искористите команду `write-tree` и испишете стејџ у објекат стабла. Није потребно да се наведе опција `-w` — позив ове команде аутоматски креира објекат стабла из стања индекса ако такво стабло још увек не постоји:

```
$ git write-tree  
d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Такође можете проверити да је ово објекат стабла употребом исте `git cat-file` команде коју сте видели раније.

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
tree
```

Сада ћете креирати ново стабло са другом верзијом фајла `test.txt`, као и са новим фајлом:

```
$ echo 'new file' > new.txt  
$ git update-index test.txt  
$ git update-index --add new.txt
```

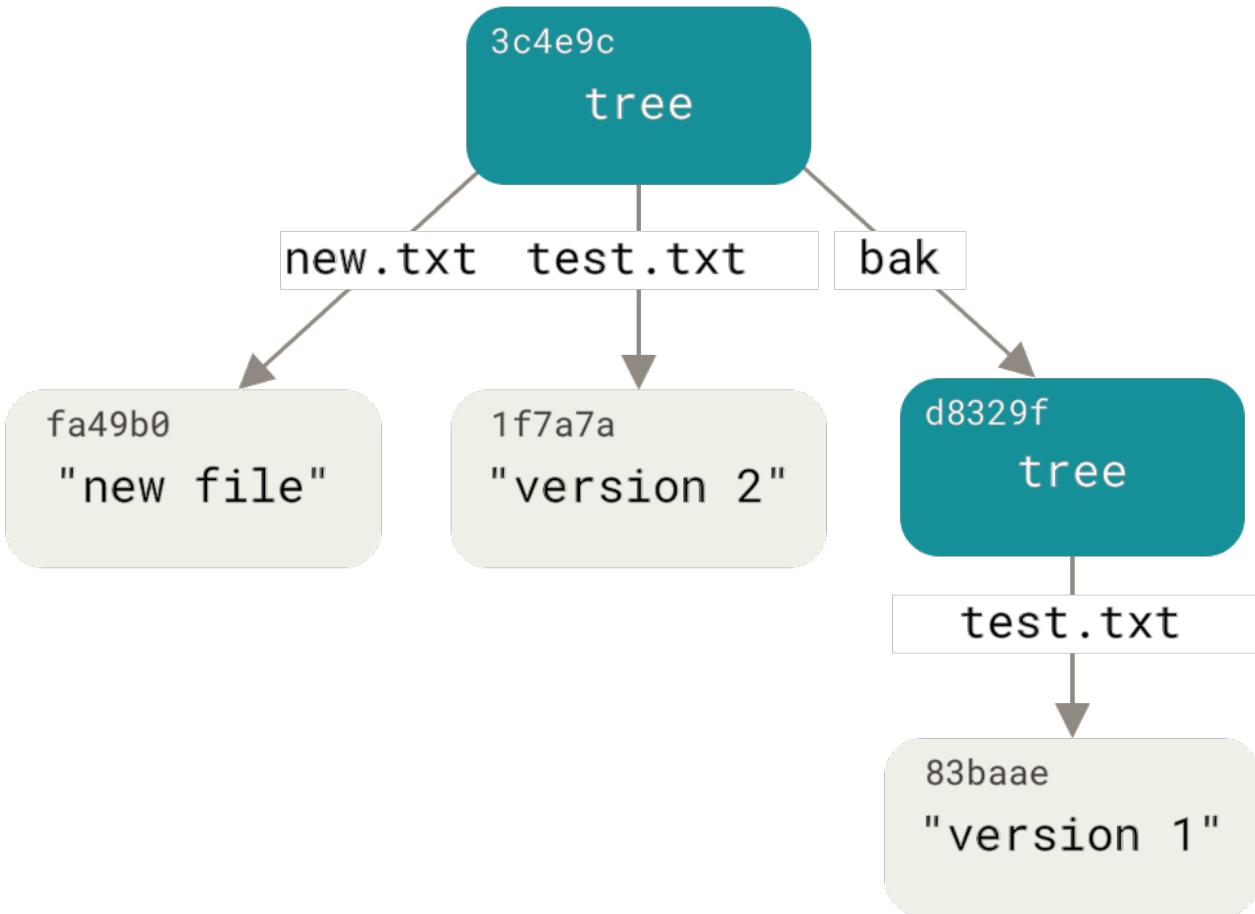
Сада ваш стејџ има нову верзију фајла `test.txt`, као и нов фајл `new.txt`. Испишите то стабло (притом бележећи стање стејџа или индекса у објекат стабла) и погледајте како изгледа:

```
$ git write-tree  
0155eb4229851634a0f03eb265b69f5a2d56f341  
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Обратите пажњу на то да ово стабло има по ставку за оба фајла и да је SHA-1 фајла `test.txt` онај за „верзију 2“ од раније ([1f7a7a](#)). Забаве ради, додаћете прво стабло као поддиректоријум у овом. Стабла можете учитати на свој стејџ командом `git read-tree`. У овом случају, учитавате постојеће стабло на стејџ као подстабло користећи опцију `--prefix` уз команду `read-tree`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git write-tree  
3c4e9cd789d88d8d89c1073707c3585e41b0e614  
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614  
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Ако радни директоријум креирате из новог стабла које сте управо исписали, добићете два фајла на највишем нивоу радног директоријума и поддиректоријум под именом `bak` који садржи прву верзију фајла `test.txt`. Податке које Гит чува за ове структуре можете да замислите овако:



Слика 148. Структура садржаја ваших текућих података у програму Гит

Комит објекти

Ако сте урадили све претходно наведено, сада имате три стабла који представљају различите снимке пројекта који желите да пратите, али остаје проблем од раније: да бисте вратили ранији снимак, морате да упамтите све три SHA-1 вредности. Поред тога немате ни информацију о томе ко је сачувао снимак, када је то учињено, нити зашто. Ово су основни подаци које се бележе у комит објекту.

Да бисте креирали комит објекат, позовите `commit-tree`, наведите SHA-1 једног стабла и ако их уопште има, који комит објекти непосредно претходе овом који креirate. Почните са првим стаблом које сте записали:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

i Добићете другачију хеш вредност јер се подаци о времену креирања и аутору разликују. Уз то, мада се било који комит објекат може прецизно репродуковати помоћу ових података, историјски детаљи конструкције ове књиге значе да одштампани комит хешеви можда не одговарају датим комитовима. У наставку овог поглавља, замените наведене комит хешеве и хешеве ознака са вредностима које ви добијете.

Сада командом `cat-file` можете да погледате свој нови комит објекат:

```
$ 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

Формат комит објекта је једноставан: наводи вршно стабло за снимак пројекта у том тренутку; комитове родитеље, ако постоје (комит објекат приказан изнад нема ниједног родитеља); информације о аутору/комитеру (што користи `user.name` и `user.email` конфигурациона подешавања и временску ознаку); празну линију, па затим комит поруку.

Сада ћете записати још два комит објекта, при чему сваки показује на комит који је дошао непосредно пре њега:

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'Third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Сваки од три комит објекта показује на један од три стабла снимака које сте креирали. Зачудо, сада имате праву Гит историју коју можете да погледате командом `git log`, ако је покренете са SHA-1 контролном сумом последњег комита.

```
$ 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
```

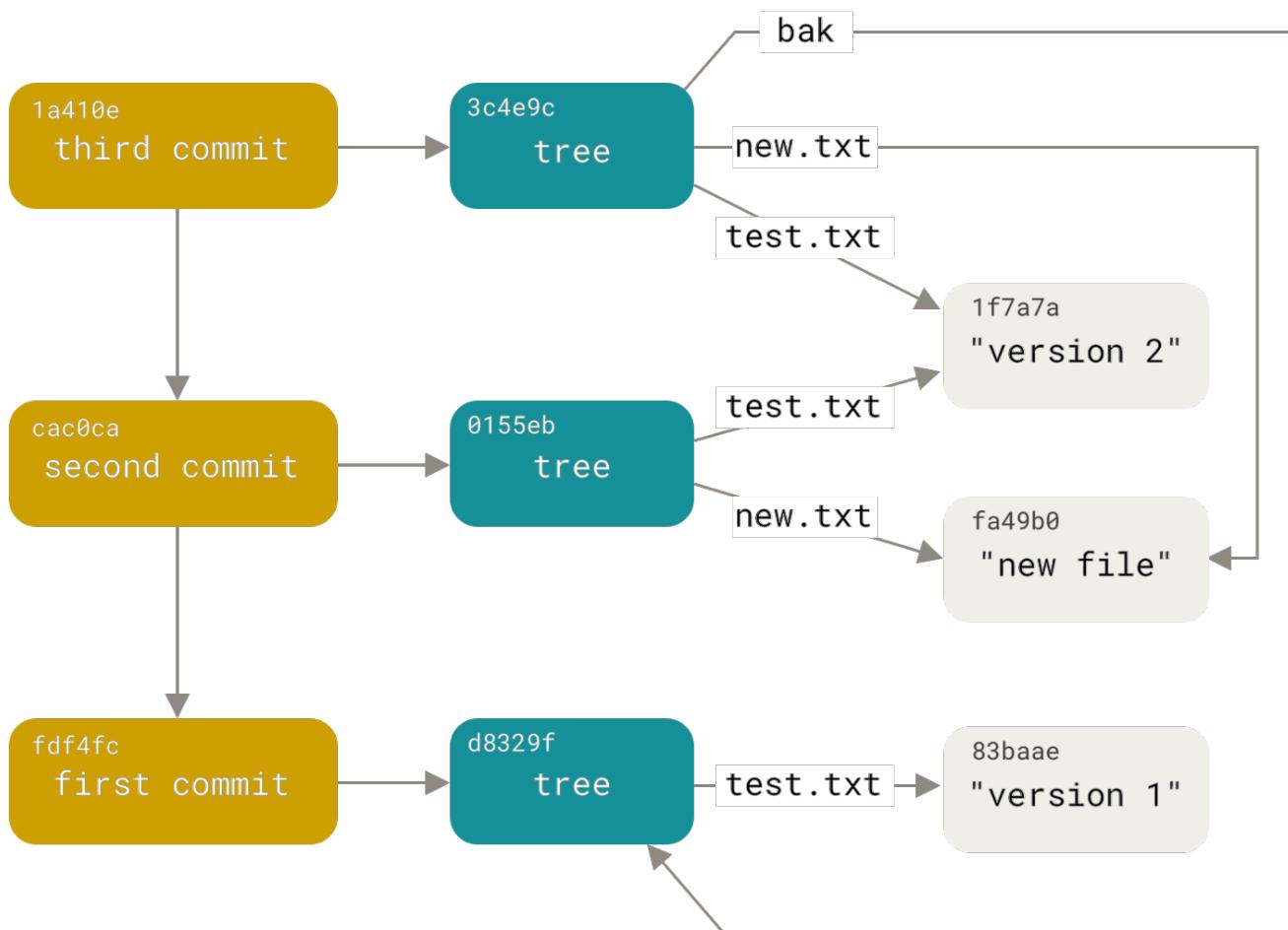
First commit

```
test.txt | 1 +
1 file changed, 1 insertion(+)
```

Невероватно. Управо сте операцијама ниског нивоа изградили Гит историју, не користећи ниједну команду корисничког интерфејса. У суштини, ово је оно што програм Гит ради када покренете команде `git add` и `git commit`—чува блобове за фајлове који су се променили, ажурира индекс, исписује стабла и комит објекте који указују на вршна стабла и комитове који им непосредно претходе. Ова три главна Гит објекта—блоб, стабло и комит—се иницијално чувају као посебни фајлови у директоријуму `.git/objects`. Ево свих објеката из тренутног стања нашег директоријума који користимо за овај пример, уз коментар о томе шта чувају:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cf9 # 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
```

Ako ispratite sve unutrašnje pokazivače, dobićete graf objekata koji izgleda nekako ovako:



Слика 149. Сви доступни објекти из Гит директоријума

Складиште објеката

Раније смо поменули да постоји заглавље које се чува заједно са сваким објектом који комитујете у Гит базу података објеката. Погледајмо накратко како Гит чува своје објекте. Видећете како да сачувате блоб објекат—у овом случају, стринг „what is up, doc?”. — интерактивно у језику Руби.

Интерактивни Руби режим покрећете командом `irb`.

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Програм Гит конструише заглавље које почиње навођењем врсте објекта — у овом случају је то блоб. Затим му додаје размак иза којег следи величина у байтовима, па на крају *null* бајт:

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\0000"
```

Програм Гит надовезује на заглавље оригинални садржај, па онда рачуна SHA-1 контролну суму тог новог садржаја. SHA-1 вредност стринга у Рубију можете да израчунате укључивањем SHA-1 *digest* библиотеке командом `require`, а онда позивањем `Digest::SHA1hexdigest()` за стринг.

```
>> store = header + content
=> "blob 16\0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Хајде да ово упоредимо са излазом команде `git hash-object`. Овде користимо `echo -n` да спречимо додавање прелома реда у улаз.

```
$ echo -n "what is up, doc?" | git hash-object --stdin
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Програм Гит компресује нови садржај користећи *zlib*, што можете да урадите у Рубију користећи библиотеку *zlib*. Прво треба да захтевате библиотеку, па да покренете `Zlib::Deflate.deflate()` над садржајем:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9C\xCA\xC90R04c(\xFCH,Q\xC8,V(-\xD0QH\xC90\xB6\@\\x00_\x1C\@\\x9D"
```

Конечно, записаћете *zlib* компресовани садржај на диск. Одредићете путању објекта који желите да запишете (прва два карактера SHA-1 вредности су име поддиректоријума, а последњих 38 је име фајла унутар тог директоријума). За креирање поддиректоријума, ако већ не постоји, у Рубију можете да искористите функцију `FileUtils.mkdir_p()`. Затим са `File.open()` отворите фајл и позивом методе `write()` над враћеном ручком фајла испишете претходно *zlib*-компресован садржај у фајл:

```
>> 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
```

Хајде да проверимо садржај објекта командом `git cat-file`:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
---
```

И то је све — kreirali ste важећи Git blob objekat.

Сви Гит објекти се чувају на исти начин, само са другачијим типовима — уместо стринга *blob*, заглавље ће почети са *commit* или *tree*. Сем тога, мада садржај блоба може да буде скоро све, садржај комитова и стабала су форматирани на прецизно дефинисан начин.

Гит референце

Ако вас интересује да погледате историју која је доступна из комита, рецимо `1a410e`, могли бисте да извршите нешто као што је `git log 1a410e` што ће вам приказати ту историју, али ипак бисте морали да запамтите да је `1a410e` комит који желите користити као почетну тачку за ту историју. Уместо тога би било лакше да имате фајл у који би могли да сачувате ту SHA-1 вредност под једноставним именом тако уместо сирове SHA-1 вредности можете употребити то просто име.

У програму Гит, ти објекти се називају „референце” или „рефови”; у директоријуму `.git/refs` можете да пронађете фајлове које садрже те SHA-1 вредности. У тренутном пројекту, овај директоријум нема фајлове, али садржи једноставну структуру:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Ако желите да креирате нову референцу која ће вам помоћи да запамтите где се налази последњи комит, технички можете да урадите нешто једноставно као што је ово:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Сада у Гит командама уместо SHA-1 вредности можете да употребите референцу коју сте управо:

```
$ git log --pretty=oneline master  
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit  
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Не саветује се да директно мењате фајлове референци; уместо тога програм Гит нуди сигурнију команду `git update-ref` која, у случају да желите ажурирати референцу, ради управо то:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

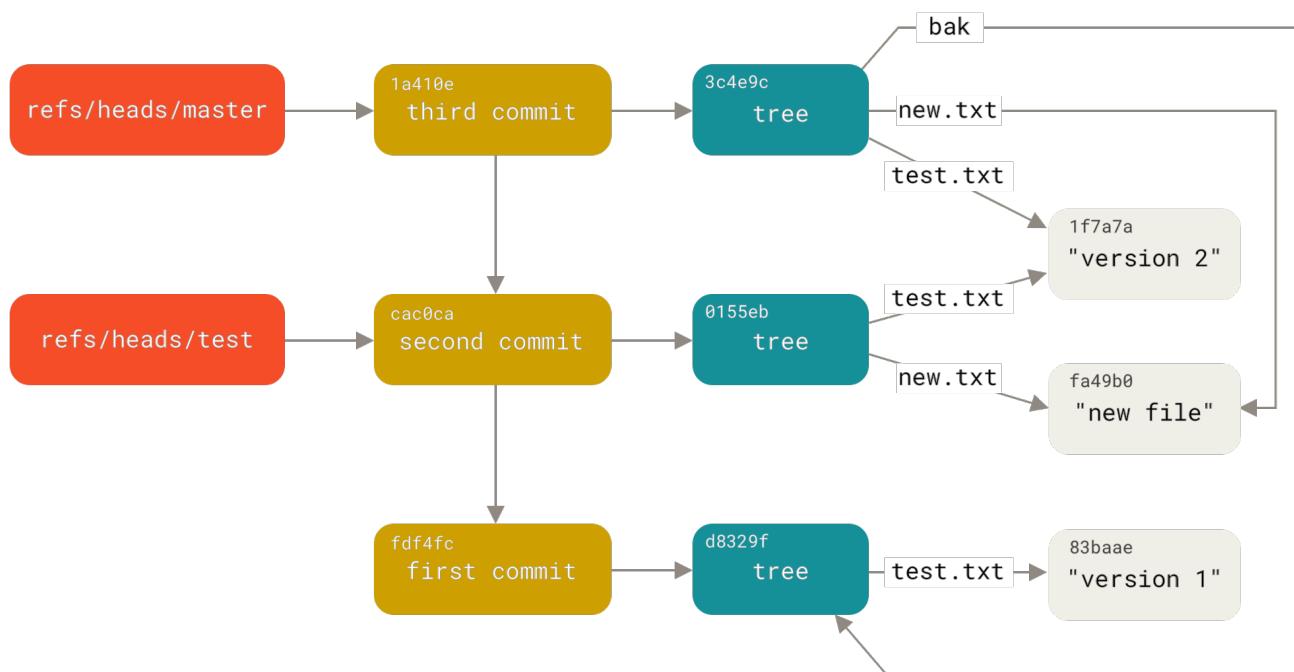
Ово је у суштини оно што у програму Гит представља грана: једноставан показивач на главу неке линије рада. Да бисте креирали грану на другом комиту, можете да урадите ово:

```
$ git update-ref refs/heads/test cac0ca
```

Ваша грана ће садржати само рад почев од тог комита па наниже:

```
$ git log --pretty=oneline test  
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Сада ваша база података програма Гит концептуално изгледа некако овако:



Слика 150. Објекти из Гит директоријума са референцама на главе грана

Када извршите команду као што је `git branch <име-гране>`, програм Гит у суштини покреће ту `update-ref` команду како би додао SHA-1 вредност последњег комита гране на којој сте тренутно у коју год нову референцу желите да креирате.

Референца HEAD

Сада се поставља питање: када извршите `git branch <име-гране>`, како програм Гит зна SHA-1 последњег комита? Одговор је у фајлу *HEAD*.

Обично је фајл *HEAD* симболичка референца на грану на којој се тренутно налазите. Под симболичком референцом мислимо на референцу која, за разлику од обичне, садржи показивач на другу референцу.

Међутим, у неким ретким случајевима *HEAD* фајл може да садржи SHA-1 вредност гит објекта. Ово се дешава када одјавите ознаку, комит, или удаљену грану што оставља ваш репозиторијум у стању „[одвојене HEAD](#)“ (одвојене главе).

Ако погледате у фајл, обично ћете видети нешто слично овоме:

```
$ cat .git/HEAD  
ref: refs/heads/master
```

Ако извршите `git checkout test`, програм Гит ажурира фајл тако изгледала овако:

```
$ cat .git/HEAD  
ref: refs/heads/test
```

Када извршите `git commit`, она креира се нови комит објекат, постављајући за родитеља тог комит објекта било коју SHA-1 вредност на коју показује референца у *HEAD*.

Можете такође и ручно да измените овај фајл; али опет, постоји сигурнија команда да се то уради: `symbolic-ref`. Вредност ваше *HEAD* можете да прочитате командом:

```
$ git symbolic-ref HEAD  
refs/heads/master
```

Истом командом можете и да поставите вредност *HEAD*:

```
$ git symbolic-ref HEAD refs/heads/test  
$ cat .git/HEAD  
ref: refs/heads/test
```

Симболичку референцу не можете да поставите ван *refs* стила:

```
$ git symbolic-ref HEAD test  
fatal: Refusing to point HEAD outside of refs/
```

Ознаке

Управо смо завршили дискусију о три главна типа Гит објеката (*блобови*, *стабла* и *комитови*), али постоји и четврти. Објекат *ознака* (таг) доста подсећа на комит објекат — садржи податке о особи која је додала ознаку, датум, поруку и показивач. Главна разлика је у томе што објекат ознаке у општем случају показује на комит, а не на стабло. Подсећа на референцу гране, али се никад не помера — увек показује на исти комит, с тим што му даје име које је лакше за памћење.

Као што смо видели у поглављу [Основе програма Гит](#), постоје две врсте ознаке: прибележене и лаке. Лаке ознаке можете да креирате на следећи начин:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769ccbde608743bc96d
```

И то је све што се тиче лаке ознаке — обична референца која се никад не помера. Прибележена ознака је, међутим, доста сложенија. Ако креирате прибележену ознаку, програм Гит креира објекат ознаке и затим пише референцу која показује на рај објекат, а не директно на комит. Ово можете да видите тако што ћете креирати прибележену ознаку (употребом заставице **-a**):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

Ево SHA-1 вредности објекта који је креiran:

```
$ cat .git/refs/tags/v1.1  
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Сада извршите команду `git cat-file -p` над тој SHA-1 вредности:

```
$ 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
```

Приметите да *object* ставка показује на SHA-1 вредност коју сте означили. Такође обратите пажњу и на то да није обавезно да показује на комит; можете означити било који Гит објекат. У извornom коду програма Гит на пример, одржавалац је додао свој GPG јавни кључ као блоб објекат и онда га означио. Јавни кључ можете погледати ако у свом клону Гит

репозиторијума извршите ово:

```
$ git cat-file blob junio-gpg-pub
```

И репозиторијум Линуксовог језгра има објекте ознака који не показују на комитове — прва направљена ознака показује на иницијално стабло увезеног извornог кода.

Удаљене референце

Трећа врста референци коју ћете виђати су удаљене референце. Ако додате удаљену репозиторијум и гурнете на њега, програм Гит ће у директоријуму `refs/remotes` чувати вредност коју сте последњу гурнули на тај удаљени репозиторијум за сваку грану. На пример, можете додати удаљени репозиторијум под именом `origin` и гурнути своју `master` грану на њега:

```
$ 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
```

Након тога можете видети шта је била грана `master` на удаљеном репозиторијуму `origin` последњи пут када сте комуницирали са сервером тако што ћете проверити фајл `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Удаљене референце се разликују од грана (`refs/heads` референци) углавном по томе што се сматра да могу само да се читају. Можете да урадите `get checkout` над неком, али програм Гит неће симболички поставити `HEAD` на неку, тако да је никад нећете ажурирати командом `git commit`. Програм Гит њима управља као са маркерима последње познатог стања тих грана на тим серверима.

Pack фајлови

Ако сте пратили сва упутства у примеру из претходног одељка, требало би да имате тест Гит репозиторијум са 11 објеката — 4 блоба, 3 стабла, 3 комита и 1 ознаку:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cf9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c7606a30 # 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
```

Програм Гит компресује садржаје ових фајлова користећи *zlib*, а ви не чувате много тога, тако да сви заједно заузимају само 925 бајтова. Додаћете мало већи садржај како бисмо демонстрирали једну занимљиву ствар у вези програма Гит. Примера ради, додаћемо `repo.rb` фајл из *Grit* библиотеке — то је око 22К извornог кода.

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ 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%)
```

Ако погледате стабло настало као резултат, видећете SHA-1 вредност израчунату за ваш нови `repo.rb` блоб објекат:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Затим можете употребити `git cat-file` да видите колика је величина тог објекта:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Сада мало измените тај фајл и погледајте шта ће се десити:

```
$ echo '# testing' >>repo.rb
$ git commit -am 'Modify repo.rb a bit'
[master 2431da6] Modify repo.rb a bit
 1 file changed, 1 insertion(+)
```

Проверите стабло које је креирано последњим комитом и видећете нешто занимљиво:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Тај блоб је сада потпуно другачији блоб, што значи да, иако сте додали само једну линију на фајла од 400 линија, програм Гит је тај нови садржај сачувао као потпуно нови објекат:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

На диску имате два скоро идентична објекта од 22К (и сваки је компресован на отприлике 7К). Зар не би било лепо кад би програм Гит могао да сачува један од њих у потпуности, а да уместо другог сачува само разлику у односу на први?

Испоставља се да програм Гит заправо то и ради. Иницијални формат у ком програм Гит чува податке на диску назива се „слободни” формат објекта. Ипак, да би сачувао простор и како би био што ефикаснији, програм Гит понекад спакује неколико таквих објеката у један бинарни фајл који се назива „pack фајл”. Програм Гит ово ради ако имате превише слободних објеката наоколо, ако ручно извршите `git gc`, или ако гурнете на удаљени сервер. Да бисте видели ово на делу, можете ручно да наредите програму Гит да спакује објекте тако што извршите команду `git gc`:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

Ако погледате у директоријум `objects`, видећете да је већина објеката нестала и да се појавио нови пар фајлова:

```
$ 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
```

Објекти који су остали су блобови на које не показује ниједан комит—у овом случају, пример блобови „what is up, doc?” и „test content” које сте креирали раније. Пошто их никад нисте додали у комитове, третирају се као слободне промене и не пакују се у *pack* фајл.

Остали фајлови су нови *pack* фајл и индекс. *pack* фајл је један фајл у којем се налази садржај свих објеката који су уклоњени из вашег фајл система. Индекс је фајл у којем се налазе помераји унутар тог *pack* фајла помоћу којих брзо можете да се поставите на одређени објекат. Одлична ствар је то што иако су објекти на диску пре извршавања команде `gc` укупно заузимали око 15К, нови *pack* фајл заузима само 7К. Смањили сте заузето диска за око пола тиме што сте спаковали објекте.

Како програм Гит ради ово? Када програм Гит пакује објекте, он тражи фајлове које имају слично име и сличну величину, па чува само разлике између једне верзије фајла и наредне. Водоводна команда `git verify-pack` вам омогућава да видите шта је спаковано:

```
$ 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
d368d0ac0678cbcbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffd9f94e0d364e0ea74f579 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
```

Овде блоб `033b4`, што је ако се сећате била прва верзија вашег `hero.rb` фајла, указује на блоб `b042a`, који је друга верзије тог фајла. Трећа колона у излазу је величина објекта у пакету, тако да можете видети да `b042a` заузима 22К фајла, али да `033b4` заузима само 9 бајтова. Оно што је такође занимљиво је то што је друга верзија фајла она која је сачувана у целости, а првобитна верзија је сачувана као разлика — подаци се чувају овако јер је вероватније да ће вам бити потребан брз приступ скорашњој верзији фајла.

Заиста одлична ствар у вези овога је што је препакивање могуће у било које време. Програм Гит ће повремено потпуно аутоматски да препакује вашу базу података, чиме увек покушава да уштеди још простора, али и ви ручно можете да је препакујете кадгод пожелите тако што ћете извршити команду `git gc`.

Рефспек

Кроз ову књигу смо користили једноставна мапирања из удаљених грана на локалне референце, али она могу бити и сложенија. Претпоставимо да сте пратили неколико последњег одељака и креирали мали локални Гит репозиторијум, па сада желите да му додате `remote` (удаљени репозиторијум):

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Извршавање ове команде додаје одељак у `.git/config` фајлу вашег репозиторијума, са наведеним именом удаљеног репозиторијума (`origin`), његовом URL адресом и `рефспек` (спецификацију референце) који ће се користити за преузимање:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Формат рефспека је прво необавезни `+` за којим следи `<изв>:<одр>`, где је `<изв>` шаблон за референце на удаљеној страни, а `<одр>` је место на којем ће се те референце пратити локално. Знак `+` говори програму Гит да ажурира референцу чак и ако није у питању премотавање унапред.

У подразумеваном случају који аутоматски уписује команду `git remote add`, програм Гит преузима све референце под `refs/heads/` на серверу и пише их локално у `refs/remotes/origin/`. Дакле, ако на серверу постоји `master` грана, локално можете да приступите њеном логу на било који од следећих начина:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Сви су еквивалентни, јер их програм Гит развија на `refs/remotes/origin/master`.

Ако уместо тога желите да програм Гит сваки пут довуче само `master` грану, а не и сваку

другу грану са удаљеног сервера, линију за преузимање можете да промените тако да указује само ту грану:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Ово је само подразумевани рефспек за `git fetch` за тај удаљени репозиторијум. Ако само једном желите да преузмете, одређени рефспек такође можете да наведете и на командној линији. Ако желите да `master` грану на удаљеном репозиторијуму повучете у локалну `origin/mymaster`, извршите следеће:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Такође можете да наведете и више рефспекова. Овако из командне линије повлачите неколико грана одједном:

```
$ 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
```

У овом случају је повлачење `master` гране одбијено јер није у питању референца са брзим премотавањем унапред. То можете да премостите ако испред рефспека наведете `+`.

Више рефспекова такође можете да наведете и у конфигурационом фајлу. Ако сваки пут желите да преузимате `master` и `experiment` гране, додајте две линије:

```
[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
```

Почевши са програмом Гит 2.6.0 у шаблону можете да користите парцијалне глобове који се подударају са више грана, тако да следеће функционише:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Још боље, исто можете да постигнете на уређенији начин ако употребите просторе имена (или директоријуме). Ако имате QA тим (тим за контролу квалитета) који гура низ грана, а ви желите да преузмете `master` грану и било коју од грана QA тима, али ништа друго, можете да употребите овакав конфигурациони одељак:

```
[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 користите компликован процес рада у којем QA тим гура гране, програмери гурају гране и тимови за интеграцију гурају и сарађују на удаљеним гранама, на овај начин можете лако да их раздвојите у просторе имена.

Гурање рефспекова

Лепо је што референцу из простора имена можете да преузмете на овај начин, али како је уопште QA ставио њихове гране у `qa/` простор имена? То се постиже употребом рефспекова за гурање.

Ако QA тим жели да своју `master` грану гурне на `qa/master` на удаљеном серверу, могу да изврше:

```
$ git push origin master:refs/heads/qa/master
```

Ако желе да програм Гит аутоматски то ради сваки пут када изврше `git push origin`, у свој конфигурациони фајл могу да додају `push` вредност:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

Да поновимо, после овога ће `git push origin` подразумевано да гурне локалну `master` грану branch на удаљену грану `qa/master`.



Рефспек не можете да искористите за преузимање из једног репозиторијума и гурање на други. Ако то желите да урадите, погледајте пример [Одржавање вашег јавног GitHub репозиторијума ажуарним](#).

Брисање референци

Рефспек можете употребити и за брисање референци са удаљеног сервера тако што извршите нешто овако:

```
$ git push origin :topic
```

Пошто је рефспек `<изв>:<одр>`, ако изоставите `<изв>` део, у суштини кажете да се `topic` грана на удаљеном репозиторијуму постави ни на шта, односно да се обрише.

Или можете да употребите новију синтаксу (доступно од Гит v1.7.0):

```
$ git push origin --delete topic
```

Протоколи за пренос

Програм Гит може да преноси податке између два репозиторијума на два главна начина: „приглуп” протокол и „паметни” протокол. Овај одељак ће укратко представити начин на који функционишу ови главни протоколи.

Приглуп протокол

Ако постављате репозиторијум који ће да се сервира само-за-читање преко HTTP, највероватније ћете употребити приглуп протокол. Овај протокол се назива „приглуп” јер током процеса преноса на серверској страни није потребан никакав Гит специфични код; процес преузимања је низ HTTP `GET` захтева, а клијент може да претпостави распоред Гит репозиторијума на серверу.



У данашње време се приглуп протокол прилично ретко користи. Не може лако да се обезбеди или учини приватним, тако да ће већина Гит хостова (и оних базираних на облаку и оних унутар фирм) одбити да га користи. У општем случају се саветује да се користи паметни протокол који ћемо описати мало касније.

Хајде да испратимо `http-fetch` процес за `simplegit` библиотеку:

```
$ git clone http://server/simplegit-progit.git
```

Прва ствар коју ова команда ради је да повуче `info/refs` фајл. Овај фајл исписује команду `update-server-info` и то је разлог због којег то морате да укључите као `post-receive` куку да би HTTP транспорт функционисао како треба:

```
=> GET info/refs  
ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

Сада имате листу удаљених референци и SHA-1 контролне суме. Затим, погледате шта је HEAD референца, тако да знате шта да одјавите када завршите:

```
=> GET HEAD  
ref: refs/heads/master
```

Када завршите процес, морате да одјавите `master` грану. Сада сте спремни да почнете са извршавањем корака процеса. Пошто је почетна тачка `ca82a6` комит објекат који сте видели у `info/refs` фајлу, крећете да га преузимате:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949  
(179 bytes of binary data)
```

Враћа вам се објекат – тај објекат је слободан на серверу, а преузели сте га статичким HTTP GET захтевом. Можете да га распакујете са *zlib*, одбаците заглавље и погледате садржај комита:

```
$ 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

Затим, постоји још два објекта које треба да преузмете – **cfda3b**, што је стабло садржаја на који управо преузети комит указује; и **085bb3**, што је родитељ комит:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
(179 bytes of data)
```

Овим добијате свој наредни комит објекат. Дохватите објекат стабла:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf  
(404 - Not Found)
```

Упс – изгледа да објекат стабла није слободан на серверу, па вам се враћа 404 одговор. За ово постоји неколико разлога – објекат би могао да се налази у алтернативном репозиторијуму, или би могао да буде унутар *pack* фајла у овом репозиторијуму. Програм Гит најпре проверава постоји ли наведена било која алтернатива:

```
=> GET objects/info/http-alternates  
(empty file)
```

Ако ово буде листа алтернативних URL адреса, програм Гит тамо проверава слободне фајлове и *pack* фајлове – ово је одличан механизам да пројекти који су рачве један другог деле објекте на диску. Међутим, пошто у овом случају нису наведене никакве алтернативе, ваш објекат мора да се налази у *pack* фајлу. Да бисте видели постојеће *pack* фајлове на овом серверу, морате да преузмете фајл **objects/info/packs** који садржи списак ових фајлова (такође га генерише команда **update-server-info**):

```
=> GET objects/info/packs  
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Постоји само један *pack* фајл на серверу, тако да се ваш објекат очигледно тамо налази, али ипак ћете проверити индекс фајл чисто да будете сигурни. Ово је такође корисно ако на серверу постоји више *pack* фајлова, тако да видите који *pack* фајл садржи објекат који вам је потребан:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)
```

Сада када имате индекс *pack* фајла, можете видети да ли се у њему налази ваш објекат – пошто индекс наводи SHA-1 суме објеката који се налазе у *pack* фајлу и помераје до тих објеката. Ваш објекат се тамо налази, па хајде да преузмемо комплетан *pack* фајл:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

Дошли сте до вашег објекта стабла, па настављате да се крећете кроз комитове. И они се налазе унутар *pack* фајла који сте управо преузели, тако да не морате да шаљете још захтева серверу. Програм Гит одјављује радну копију `master` гране на коју указује HEAD референца коју се преузели на почетку.

Паметни протокол

Приглуп протокол је једноставан али није довољно ефикасан и не подржава упис података са клијента на сервер. Паметни протокол је метод за пренос података који се обично користи, али на удаљеној страни је неопходан процес који је интелигентан у вези програма Гит – он може да чита локалне податке, одреди шта клијент има и шта му је потребно, па да за то генерише прилагођени *pack* фајл. Постоје два скрупа процеса за пренос података: пар за слање података на сервер и пар за преузимање података са сервера.

Слање података на сервер

Када треба да пошаље податке на сервер, програм Гит користи `send-pack` и `receive-pack` процесе. Процес `send-pack` се извршава на клијенту и повезује се са `receive-pack` процесом на удаљеној страни.

SSH

На пример, рецимо да у свом пројекту извршите `git push origin master`, а да је `origin` дефинисано као URL адреса која користи SSH протокол. Програм Гит покреће `send-pack` процес који успоставља везу са вашим сервером преко SSH. Он покушава да на удаљеном серверу изврши команду путем SSH позива који изгледа некако овако:

```
$ 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
```

Команда `git-receive-pack` тренутно враћа одговор који има по једну линију за сваку референцу коју тренутно има – у овом случају само `master` грану у њен SHA-1 хеш. Прва линија такође има и листу могућности сервера (овде су то `report-status`, `delete-refs`, и још неке, укључујући стринг идентификатор клијента).

Подаци се преносе у комадима. Свака линија почиње са хекс вредности дужине 4 карактера која наводи дужину остатка линије. Ваша прва линија почиње са 005b, што је хексадецимална представа броја 91, па значи да на тој линији преостаје још 91 бајт. Наредна линија почиње са 003e, што је 62, тако да читате преосталих 62 бајтова. Наредна линија је 0000, што значи да је сервер завршио са листањем референци.

Сада када зна стање сервера, ваш `send-pack` процес одређује које комитове поседује, а још нису на серверу. За сваку референцу коју ће ово гурање да ажурира, процес `send-pack` преноси процесу `receive-pack` ту информацију. На пример, ако ажурирате `master` грану и додајете `experiment` грану, `send-pack` одговор може изгледати отприлике овако:

Програм Гит шаље по линију за сваку референцу коју ажурирате у којој се налази дужина линије, стара SHA-1 сума, нова SHA-1 сума и референца која се ажурира. У првој линији се налазе и могућности клијента. Ако се SHA-1 вредност састоји само од нула, то значи да раније није било ничега – јер додајете експеримент референцу. Ако бришете референцу, видели бисте супротно: све нуле са десне стране.

Затим, клијент шаље *pack* фајл свих објеката које сервер још увек нема. Коначно, сервер одговара тако што означава успех (или неуспех):

000eunpack ok

HTTP(S)

Овај процес је углавном исти као и преко HTTP, мада је руковање донекле другачије. Успостављање везе се започиње следећим захтевом:

```
=> 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~vmpg-bitmaps-bugaloo-608-g116744e  
0000
```

То је крај прве клијент-сервер размене. Клијент затим шаље наредни захтев, овога пута **POST**, са подацима које достави **git-upload-pack**.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

У товару **POST** захтева се налази излаз из **send-pack** и **pack** фајл. Сервер затим својим HTTP одговором означава успех или неуспех.

Имајте на уму да HTTP протокол ове податке може додатно да обмата у пренос по комадима.

Преузимање података са сервера

Када преузимате податке, укључени су **fetch-pack** и **upload-pack** процеси. Клијент покреће **fetch-pack** процес који се повезује са **upload-pack** процесом на удаљеној страни и преговара о томе који подаци ће се пренети.

SSH

Ако преузимање обављате преко SSH, **fetch-pack** извршава нешто отприлике овако:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Након што се **fetch-pack** повеже, **upload-pack** шаље назад нешто отприлике овако:

```
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
```

Ово је врло слично одговору који шаље **receive-pack**, али су другачије могућности. Уз то, шаље назад и оно на шта показује HEAD (**symref=HEAD:refs/heads/master**) тако да клијент зна шта да одјави ако је у питању клонирање.

У овом тренутку **fetch-pack** процес тражи објекте које има и враћа одговор са објектима који су му потребни, тако што пошаље „want” и онда SHA-1 хеш онога што ми је потребно. Све објекте које већ им шаље са „have” и онда SHA-1. На крају ове листе исписује „done” чиме сигнализира процесу **upload-pack** да започне слање **pack** фајла са подацима који су потребни:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

Руковање за операцију преузимања подразумева два HTTP захтева. Први је **GET** исте крајње тачке која се користи и у приглупом протоколу:

```
=> 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
003fcfa82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Ово веома личи на позив **git-upload-pack** преко SSH везе, али се друга размена обавља као одвојени захтев:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fd9a93eb2908e52742248faf0ee993
0000
```

Још једном, ово је исти формат као и раније. Одговор на овај захтев означава успех или неуспех и садржи *pack* фајл.

Резиме протокола

Овај одељак садржи доста упрошћени преглед протокола за пренос. Протокол има још дosta других могућности, као што су **multi_ack** или **side-band** могућности, али је њихово разматрање ван оквира ове књиге. Покушали смо да вам представимо општу размену између клијента и сервера; ако желите више знања о овоме, вероватно би требало да погледате у изворни код програма Гит.

Одржавање и опоравак податак

Повремено може бити потребе да се обави мало чишћења – да се репозиторијум мало сажме, прочисти увезени репозиторијум, или опорави изгубљени рад. Овај одељак ће приказати неке од тих сценарија.

Одржавање

Програм Гит повремен о аутоматски покреће команду која се зове „auto gc”. Ова команда најчешће не ради ништа. Међутим, када има превише слободних објеката (оних који се не налазе у *pack* фајлу) или превише *pack* фајлова, програм Гит покреће темељну `git gc` команду. Овде „gc” замењује *garbage collect* (скупљање отпада) и та команда обавља више ствари: скупља се слободне објекте и поставља их у *pack* фајлове, консолидује *pack* фајлове у један велики *pack* фајл и уклања објекте до којих не може да се стигне из ниједног комита, а стари су барем неколико месеци.

Команду *auto gc* можете и ручно да покренете на следећи начин:

```
$ git gc --auto
```

Да поновимо, ово у општем случају не ради ништа. Морате имати око 7.000 слободних објеката, или више од 50 *pack* фајлова да би програм Гит покренуо праву *gc* команду. Ове границе можете променити конфигурационим подешавањима `gc.auto` и `gc.autopacklimit` респективно.

Друга ствар коју ће *gc* урадити је да ваше референце спакује у један фајл. Претпоставимо да репозиторијум садржи следеће гране и ознаке:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Ако извршите `git gc`, више нећете имати ове фајлове у *refs* директоријуму. Програм Гит ће их у циљу ефикасности преместити у фајл под именом `.git/packed-refs` који изгледа овако:

```
$ 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
```

Ако ажурирате референцу, програм Гит не уређује овај фајл, већ уписује нови фајл у *refs/heads*. Да би за дату референцу дошао до одговарајуће SHA-1 вредности, програм Гит тражи ту референцу у *refs* директоријуму, па онда проверава *packed-refs* фајл. Тако да ако референцу не нађе у *refs* директоријуму, она се највероватније налази у вашем *packed-refs* фајлу.

Приметите последњу линију фајла која почиње са `^`. То значи да је ознака непосредно изнад обележена ознака и да је та линија комит на коју показује обележена ознака.

Опоравак података

У неком тренутку током вашег Гит пута, можете грешком да изгубите комит. У општем случају, ово се дешава јер сте насиљно обрисали грани на којој се налазио рад, а испоставља се да вам је трансакција ипак била потребна; ули урадите *hard-reset* грани, чиме напуштате комитове из којих сте нешто хтели. Под претпоставком да се то догодило, постоји ли начин да комитове вратите?

Ево примера коју ради *hard-reset master* вашег тест репозиторијума на старији комит, па онда опоравља изгубљене комитове. Најпре, хајде да погледамо стање вашег репозиторијума у овом тренутку:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Померимо сада *master* грани на средњи комит:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Ефективно сте изгубили горња два комита – немате више грани из које се може стићи у те комитове. Морате да пронађете SHA-1 последњег комита, па да додате грани која показује на њега. Трик је пронаћи тај SHA-1 последњег комита – не сећате се баш како се то ради, зар не?

Често је најбржи начин да се употреби алат који се зове *git reflog*. Док радите, програм Гит у тишини чува шта је ваш HEAD сваки пут када га измените. Сваки пут када комитујете или пређете на другу грани, ажурира се *reflog*. *Reflog* се такође ажурира командом *git update-ref* и то је још један разлог да корисите њу, уместо да једноставно уписујете SHA-1 вредност у своје *ref* фајлове, као што смо показали у [Гит референце](#). Ако извршите *git reflog*, видећете где сте се налазили у било које време:

```
$ 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
```

Овде можемо видети два комита које смо одјавили, међутим нема много детаља. Ако желите да исте информације видите на много кориснији начин, треба да извршите *git log -g*, што вам приказује уобичајени лог излаз за ваш *reflog*.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700
```

Third commit

```
commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700
```

Modify `repo.rb` a bit

Изгледа да је комит на дну онај који сте изгубили, тако да га опорављате креирајући нову грану на том комиту. На пример, на том комиту (`ab1afef`) можете да започнете нову грану са именом `recover-branch`:

```
$ 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
```

Фин – сада имате грану која се зове `recover-branch` на оном месту на којем се налазила ваша `master` грана, чиме поново може да се дође до прва два комита. Даље, претпоставимо да се из неког разлога ваш губитак не налази у `reflog` – то може да се симулира уклањањем `recover-branch` гране и брисањем `reflog`. Сада ништа не може да дође до прва два комита:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Пошто се `reflog` подаци чувају у `.git/logs/` директоријуму, ефективно више немате `reflog`. Како сада да вратите те комитове? Један начин је да употребите алат `git fsck` који проверава интегритет базе података. Ако га покренете са опцијом `--full`, он вам приказује све објекте на које не указује ниједан други објекат:

```
$ 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
```

У овом случају, ваш комит који недостаје видите иза стринга „dangling commit”. Можете да га опоравите на исти начин, додавањем гране која показује на тај SHA-1.

Уклањање објеката

Постоји много одличних ствари у вези програма Гит, али једна од могућности која може да буде узрок проблема је чињеница да `git clone` преузима целокупну историју пројекта, укључујући сваку верзију сваког фајла. Ово није тако лоше ако је у репозиторијуму само изворни кôд, јер је програм Гит одлично оптимизован да ефикасно компресује податке. Међутим, ако је неко у неком тренутку историје вашег пројекта додао један огроман фајл, свако клонирање ће увек бити принуђено да преузима тај велики фајл, чак и ако је одмах био уклоњен у наредном комиту. Пошто до њега може да се стигне из историје, он ће тамо заувек да остане.

Ово може да представља велики проблем када Subversion или Перфорс репозиторијуме конвертујете у Гит. Пошто у тим системима не преузимате комплетну историју, ова врста додавања има мало последица. Ако сте обавили увоз из неког другога система, или откријете да вам је репозиторијум много већи него што би требало да буде, ево начина на који можете да пронађете и уклоните велике објекте.

Упозоравамо вас: ова техника може да оштети вашу историју комитова. Она поново исписује сваки комит објекат почевши од најранијег стабла које морате да измените како би уклонили референцу на велики фајл. Ако то урадите непосредно након увоза, пре него што је било ко свој рад почeo да базира на комиту, онда нема проблема – у супротном, све сараднике морате обавестити да свој рад морају ребазирати преко ваших нових комитова.

Да бисмо показали ово, додаћете велики фајл у свој тест репозиторијум, уклонићете га у наредном комиту, пронађи га и трајно уклонити из репозиторијума. Најпре, додајте велики објекат у своју историју:

```
$ curl -L 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
```

Ууups – нисте хтели да у свој пројекат додате огромну *tarball* архиву. Боље да је се решите:

```
$ 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
```

А сада, извршите `gc` над својом базом да видите колико простора заузима:

```
$ 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)
```

Можете да извршите и команду `count-objects` да брзо погледате колико простора се користи:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

Ставка `size-pack` је величина ваших *pack* фајлова у килобајтима, тако да користите скоро 5МБ. Пре последњег комита, трошило се близу 2К – очито, уклањање фајла из претходног комита га није уклонило из историје. Сваки пут када неко клонира овај репозиторијум, мораће да клонирају свих 5МБ само да би преузели овај малецки пројекат, јер сге грешком додали велики фајл. Хајде да га се решимо.

Најпре морате да га пронађете. У овом случају већ знате који је то фајл. Али претпоставимо да не знате; како можете да откријете који фајл или фајлови заузимају толики простор? Ако извршите `git gc`, сви објекти се налазе у *pack* фајлу; велике објекте можете да препознане извршавањем друге водоводне команде која се зове `git verify-pack` и сортирањем по трећем пољу излаза, које представља величину. Такође можете и да преусмерите излаз кроз команду `tail` јер вас интересује само неколико највећих фајлова на крају:

```
$ 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
```

Велики објекат је на дну: 5МБ. Да бисте пронашли у ком фајлу се налази, употребићете команду `rev-list` коју сте накратко употребили у [Справођење одређеног формата за комит поруке](#). Ако команди `rev-list` проследите `--objects`, она ће приказати све SHA-1 вредности комитова заједно са придрженим путањама фајлова. Да бисте пронашли име вашег блоба, употребите следеће:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Сада треба да уклоните овај фајл из свих стабала у вашој прошлости. Лако можете да видите који комитови су мењали овај фајл:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - removed large tarball
7b30847 Add git tarball
```

Да бисте овај фајлу потпуно уклонили из Гит историје, морате поново да испишете све комитове низводно од [7b30847](#). За то ћете употребити команду `filter-branch`, коју сте већ користили у [Поновно исписивање историје](#):

```
$ 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
```

Опција `--index-filter` је слична опцији `--tree-filter` која је употребљена у [Поновно исписивање историје](#), осим што уместо да прослеђивања команде која мења одјављене фајлове на диску, она сваки пут мења стејџ или индекс.

Уместо да одређени фајл уклоните са нечим као што је `rm фајл`, морате да га уклоните са `git rm --cached` – морате да га уклоните из индекса, а не са диска. Разлог због којег се овако ради је брзина – пошто програм Гит не мора да одјављује сваку ревизију на диск пре него што изврши ваш филтер, процес може да буде много, много бржи. Ако желите, исти задатак можете да извршите са `--tree-filter`. `--ignore-unmatch` опција команде `git rm` говори да не прекине извршавање са грешком ако не нађе на шаблон који покушавате да уклоните. Коначно, тражите од команде `filter-branch` да поново испише вашу историју почевши од комита [7b30847](#) на овамо, јер знаете да је проблем настало на том месту. У супротном, она ће да

крене од почетак и без потребе ће продужити извршавање.

Ваш историја више нема референцу на тај фајл. Међутим, ваш `reflog` и нови скуп референци које је програм Гит додао када сте извршили `filter-branch` под `.git/refs/original` још увек садрже, тако их морате уклонити и препаковати базу података. Пре поновног паковања, морате да се решите свега што има показивач на те старе комитове:

```
$ 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)
```

Хајде да видимо колико простора сте уштедели.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Величина спакованог репозиторијума је спала на 8К, што је много боље од 5МБ. Из `size` вредности видите да се велики објекат још увек налази у слободним објектима, тако да није нестао; али се приликом гурања или накнадног клонирања неће преносити, а те јо оно што је битно. Ако то заиста желите, могли бисте потпуно да уклоните објекат ако извршите `git prune` са `--expire` опцијом:

```
$ 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
```

Променљиве окружења

Програм Гит се увек извршава у `bash` љуски и користи већи број променљивих окружења које одређује његово понашање. Понекад је згодно знати које су и како могу да се употребе за прилагођење понашања програма Гит вашим потребама. Ово није потпуна листа свих променљивих окружења које програм Гит ослушкује, мада ћемо представити оне најкорисније.

Глобално понашање

Нека од општих понашања програма Гит као компјутерског програма зависе од променљивих окружења.

`GIT_EXEC_PATH` одређује место на којем програм Гит тражи своје потпрограме (као што су `git-commit`, `git-diff` и остали). Тренутну вредност можете проверити ако извршите `git --exec-path`.

`HOME` се у општем случају не сматра за променљиву која треба да се прилагоди (превише осталих ствари зависи од ње), али она је место на којем програм Гит тражи свој глобални конфигурациони фајл. Ако желите заиста преносну инсталацију програма Гит, заједно са глобалном конфигурацијом, у профилу љуске преносног програма Гит можете да преиначите вредност `HOME`.

`PREFIX` је слична, само за конфигурацију на нивоу система. Програм Гит тражи овај фајл у `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, ако је постављена, искључује употребу системског конфигурационог фајла. Ово је корисно ако системска конфигурација кvari ваше команде, али немате дозволу да је измените или уклоните.

`GIT_PAGER` контролише програм који се користи у командној линији за приказ излаза у више страница. Ако није постављена, користиће се вредност променљиве `PAGER`.

`GIT_EDITOR` је едитор који програм Гит покреће када је потребно да корисник уреди неки текст (на пример, комит поруку). Ако није постављена, користиће се вредност променљиве `EDITOR`.

Локације репозиторијума

Програм Гит употребљава неколико променљивих окружења да одреди начин на који је у спрези са текућим репозиторијумом.

`GIT_DIR` је локација `.git` директоријума. Ако се не наведе, програм Гит обиласи стабло директоријума навише све док не стигне у `~` или `/` и у сваком кораку покушава да нађе `.git` директоријум.

`GIT_CEILING_DIRECTORIES` контролише начин на који се тражи `.git` директоријум. Ако приступите директоријумима који се споро учитавају (као што су они који се налазе на уређају који чита и уписује по тракама, или преко споре мрежне везе), вероватно ћете хтети да Програм Гит прекине претрагу раније него иначе, посебно ако је програм Гит позван

током изградње одзива ваше љуске.

GIT_WORK_TREE је локација корена радног директоријума за репозиторијум који није оголјен. Ако се не наведе, користиће се директоријум родитељ **\$GIT_DIR** директоријума.

GIT_INDEX_FILE је путања до индекс фајла (само за неоголјене репозиторијуме).

GIT_OBJECT_DIRECTORY може да се користи за навођење локације директоријума чије је уобичајено место `.git/objects`.

GIT_ALTERNATE_OBJECT_DIRECTORIES је листа раздвојена двотачкама (форматирана као `/дир/један:/дир/два:…`) која програму Гит говори где да потражи објекте ако се не налазе у **GIT_OBJECT_DIRECTORY** директоријуму. Ако имате много пројеката са великим фајловима који имају потпуно исте садржаје, употребите ово како се не би чувало много копија тих фајлова.

Спецификације путања (*Pathspecs*)

„pathspec” се представља начин на који стварима у програму Гит наводите путање, укључујући и употребу цокера. Они се користе у `.gitignore` фајлу, али и на командној линији (`git add *.c`).

GIT_GLOB_PATHSPECS и **GIT_NOGLOB_PATHSPECS** контролишу подразумевано понашање цокера у спецификацијама путање. Ако се **GIT_GLOB_PATHSPECS** постави на 1, цокер карактери се понашају као цокери (што је и подразумевана вредност); ако се **GIT_NOGLOB_PATHSPECS** постави на 1, цокер карактери се подударају сами са собом, што значи да би се нешто као што је `*.c` подударило само са фајлом чије је име „`*.c`”, а не са било којим фајлом чије се име завршава на `.c`. Ово можете да преиначите за појединачне случајеве тако што спецификацију путање почнете са `:(glob)` или `:(literal)`, као у `:(glob)/*.c`.

GIT_LITERAL_PATHSPECS искључује оба претходна понашања; неће радити ниједан цокер карактер, а искључиће се и префикс за премошћавање.

GIT_ICASE_PATHSPECS поставља да све спецификације путања не праве разлику у величини слова.

Комитовање

Завршно креирање Гит комит објекта обично обавља `git-commit-tree` која користи ове променљиве окружења као свој примарни извор информација, а конфигурационим вредностима прибегава само у случају да нису присутне.

GIT_AUTHOR_NAME је читљиво име „author” поља.

GIT_AUTHOR_EMAIL је имејл за „author” поље.

GIT_AUTHOR_DATE је временска ознака која се користи за „author” поље.

GIT_COMMITTER_NAME поставља људско име за „committer” поље.

GIT_COMMITTER_EMAIL је имејл адреса за „committer” поље.

GIT_COMMITTER_DATE се користи за временску ознаку у „committer” пољу.

EMAIL је резервна имејл адреса у случају када није постављена конфигурациона вредност `user.email`. Ако ова није постављена, програм Гит прибегава именима системског корисника и хоста.

Умрежавање

Програм Гит за мрежне операције преко HTTP протокола користи `curl` библиотеку, тако да **GIT_CURL_VERBOSE** програму Гит говори да емитује све поруке које генерише та библиотека. Ово је слично са `curl -v` на командној линији.

GIT_SSL_NO_VERIFY говори програму Гит да не проверава SSL сертификате. Некада то може бити неопходно као на пример када за сервирање Гит репозиторијума преко HTTPS користите самопотписани сертификат, или ако се усред подешавања Гит сервера и још увек нисте инсталериали потпуни сертификат.

Ако је проток података HTTP операције мањи од **GIT_HTTP_LOW_SPEED_LIMIT** фајлова у секунди у периоду дужем од **GIT_HTTP_LOW_SPEED_TIME** секунди, програм Гит ће прекинути ту операцију. Ове вредности преиначују конфигурационе вредности `http.lowSpeedLimit` и `http.lowSpeedTime`.

GIT_HTTP_USER_AGENT поставља *user-agent* стринг који програм Гит користи када комуницира преко HTTP. Подразумевана вредност личи на `git/2.0.0`.

Прављење разлика и спајање

GIT_DIFF_OPTS је донекле нетачно име. Једине важеће вредности су `-u<n>` или `--unified=<n>` које контролишу број контекст линија које се приказују у излазу команде `git diff`.

GIT_EXTERNAL_DIFF се користи за премошћавање конфигурационе вредности `diff.external`. Ако је постављена, програм Гит ће покренути тај програм када извршите команду `git diff`.

GIT_DIFF_PATH_COUNTER и **GIT_DIFF_PATH_TOTAL** су корисне из програма који наведе **GIT_EXTERNAL_DIFF** или `diff.external`. Прва представља фајл из низа за који се ради разлика (почевши од 1), а друга укупан број фајлова у групи.

GIT_MERGE_VERBOSITY контролише излаз за рекурзивну стратегију спајања. Дозвољене су следеће вредности:

- 0 нема никаквог излаза, осим можда једне поруке о грешки.
- 1 приказују се само конфликти.
- 2 приказују се и измене фајла.
- 3 приказује када се фајлови прескоче јер нису били мењани.
- 4 приказују се све путање онако како се обрађују.
- 5 и веће приказује детаљне дибаг информације.

Подразумевана вредност је 2.

Дибаг

Желите *засиста* да знате шта је то програм Гит наумио? У програм Гит је утрађен прилично комплетан скуп трагова, па све што треба да урадите је да их укључите. Ево могућих вредности ових променљивих:

- „true”, „1”, или „2” – категорија трага се исписује на *stderr*.
- Апсолутна путања која почиње са `/` – излаз трага ће се уписати у тај фајл.

GIT_TRACE контролише опште трагове који не упадају ни у једну одређену категорију. То укључује развијање алијаса и делегирање другим потпрограмима.

```
$ 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 контролише праћење *packfile* приступа. Прво поље је *packfile* којем се приступа, а друго је померај унутар тог фајла:

```
$ 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 омогућава праћење мрежних операција на нивоу пакета.

```
$ 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 контролише како се логују подаци о перформанси. Излаз приказује колико траје извршавање неке одређене *git* команде.

```
$ 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'
'rerec' 'gc'
20:18:25.233159 trace.c:414          performance: 6.112217000 s: git command: 'git'
'gc'
```

GIT_TRACE_SETUP приказује информације о ономе што програм Гит открива у вези репозиторијума и окружења са којим је у контакту.

```
$ 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
```

Разно

GIT_SSH, ако се наведе, то је програм који се уместо стандардне `ssh` команде позива онда када програм Гит покуша да се повеже са SSH хостом. Позива се као `$GIT_SSH [корисничкоиме @]хост [-р <порт>] <команда>`. Имајте на уму да ово није најједноставнији начин да се прилагоди начин на који се позива програм `ssh`; он не подржава додатне параметре командне линије, тако да ћете морати да напишете скрипту омотач и поставите да `GIT_SSH` показује на њу. It's probably easier just to use the `~/.ssh/config` file for that.

GIT_ASKPASS је премошћавање `core.askpass` конфигурационе вредности. Ово је програм који се позива када програм Гит треба кориснику да пита за акредитиве, који као аргумент командне линије може да прихвати текстуални одзив, а требало би да врати одговор на `stdout` (за више информација о овом подсистему, погледајте [Складиште акредитива](#)).

GIT_NAMESPACE контролише приступ референцима у простору имена и еквивалентна је са заставицом `--namespace`. Ово је углавном корисно на серверској страни када можете пожелети да у један репозиторијум сачувате више рачви једног јединог репозиторијума, чувајући одвојено само референце.

GIT_FLUSH може да се употреби да се програм Гит примора да када инкрементално уписује на `stdout` користи У/И који се не баферије. Вредност 1 наводи програм Гит да чешће спира податке, а вредност 0 чини да се сав излаз баферије. Подразумевана вредност (ако се ова променљива не постави) је изабрана тако да одговара шеми баферања која зависи од активности и режима излаза.

GIT_REFLOG_ACTION вам омогућава да задате описни текст који се исписује у `reflog`. Ево примера:

```
$ 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
```

Резиме

Сада би требало да прилично добро разумете шта то програм Гит ради у позадини и, донекле, начин на који је имплементиран. Ово поглавље је објаснило већи број водоводних команди — команди нижег нивоа које су једноставније од портуланских команди о којима

сте учили у остатку књиге. Разумевање начина на који програм Гит функционише на нижем нивоу би требало да вам помогне да схватите зашто он ради оно што ради, као и да напишете сопствене алате и помоћне скрипте којима специфични процес рада прилагођавате да ради за вас.

Као фајл систем који се адресира садржајем, програм Гит је веома моћан алат који врло лако можете да користите и као нешто више од обичног VCS. Надамо се да ћете своје новостечено знање унутрашњости програма Гит употребите да имплементирате сопствене кул примене ове технологије и да ћете се осећати комфорније користећи програм Гит на напредније начине.

Додатак А: Програм Гит у другим окружењима

Ако сте прочитали цelu књигу, научили сте доста о томе како се програм Гит користи из командне линије. Можете да радите са локалним фајловима, повезујете преко мреже свој репозиторијум са другим репозиторијумима и ефикасније радите са осталима. Али прича се овде не завршава; програм Гит је обично део већег екосистема, а терминал није увек најбољи начин да се ради са њим. Сада ћемо погледати неке од осталих врста окружења у којима програм Гит може бити користан и како остале апликације (укључујући и вашу) раде уз програм Гит.

Графички интерфејси

Природно окружење програма Гит је терминал. Ту се најпре појаве нове могућности, а и пуну снагу програма Гит имате на располагању само из командне линије. Али чисти текст није најбољи избор за све задатке; понекад је оно што вам треба визуелна представа, а и неки корисници се много комфорније осећају у покажи-и-клиකни интерфејсу.

Важно је приметити да су различити интерфејси прилагођени различитим радним процесима. Неки клијенти излажу само пажљиво одабрани подскуп функционалности програма Гит, како би подржали специфични начин рада који аутор сматра ефикасним. Када се посматра из овог угла, ниједан од ових алата не може да се назове „бољим“ од било ког другог, они су једноставно погоднији за примену којој су намењени. Исто тако, приметите да не постоји ништа што ови графички клијенти могу да ураде, а да не може клијент из командне линије; командна линија је још увек место на којем имате највише снаге и контроле када радите са својим репозиторијумима.

gitk и git-gui

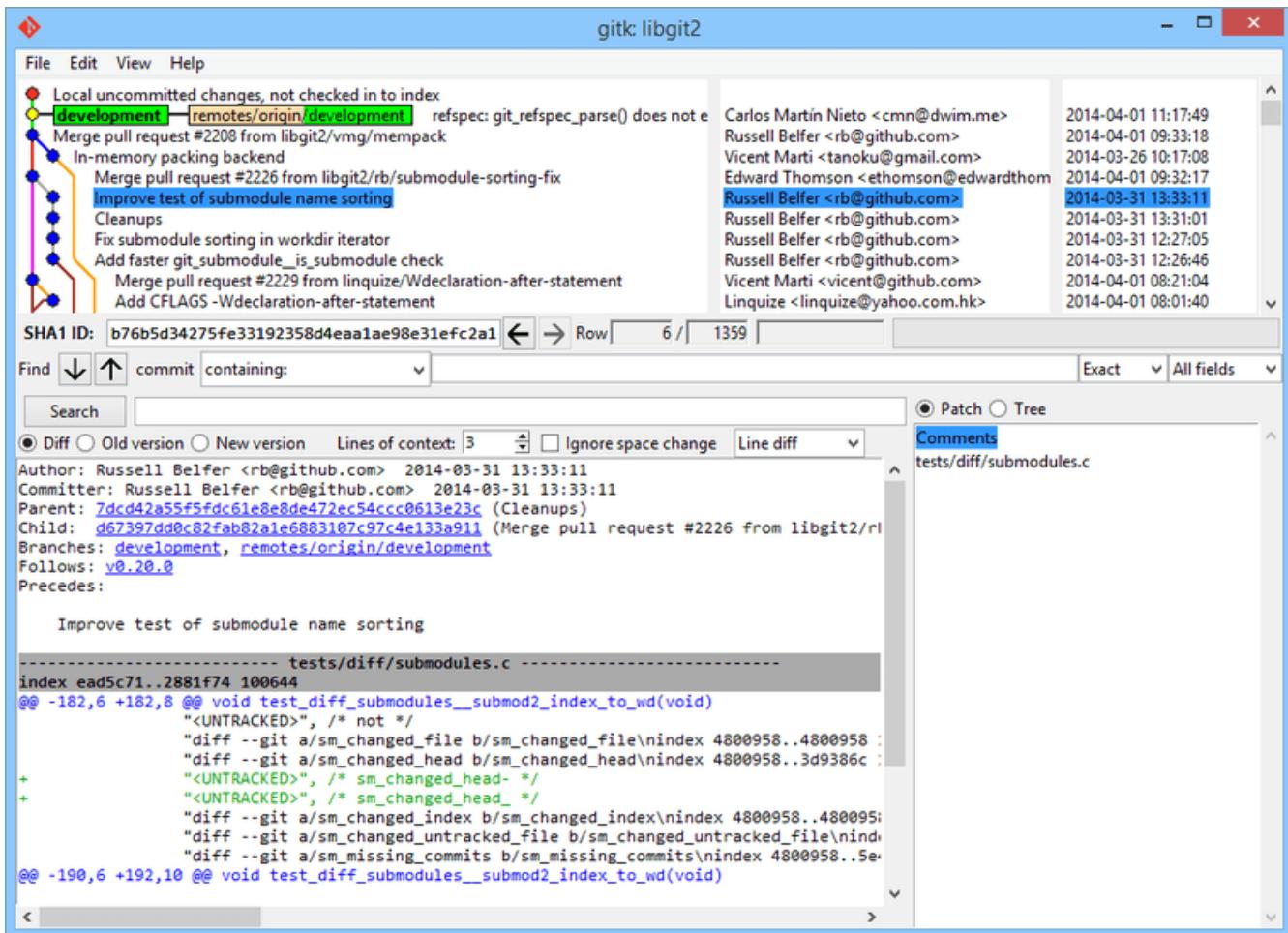
Када инсталирате програм Гит, такође добијате и визуелне алате, `gitk` и `git-gui`.

`gitk` вам даје графички приказ историје. Посматрајте га као моћну ГКИ љуску преко `git log` и `git grep` команди. Ово је алат који треба користити када покушавате да пронађете нешто што се дододило у прошлости, или да визуелизујете историју свог пројекта.

Gitk се најлакше позива из командне линије. Једноставно урадите `cd` у Гит репозиторијум и откуцајте:

```
$ gitk [git log опције]
```

Gitk прихвата многе опције командне линије, од којих већину прослеђује позадинској `git log` акцији. Вероватно једна од најкориснијих заставица је `--all`, која програму `gitk` налаже да прикаже комитове који су доступни из *било које* референце, не само из HEAD. Интерфејс програма `gitk` изгледа овако:



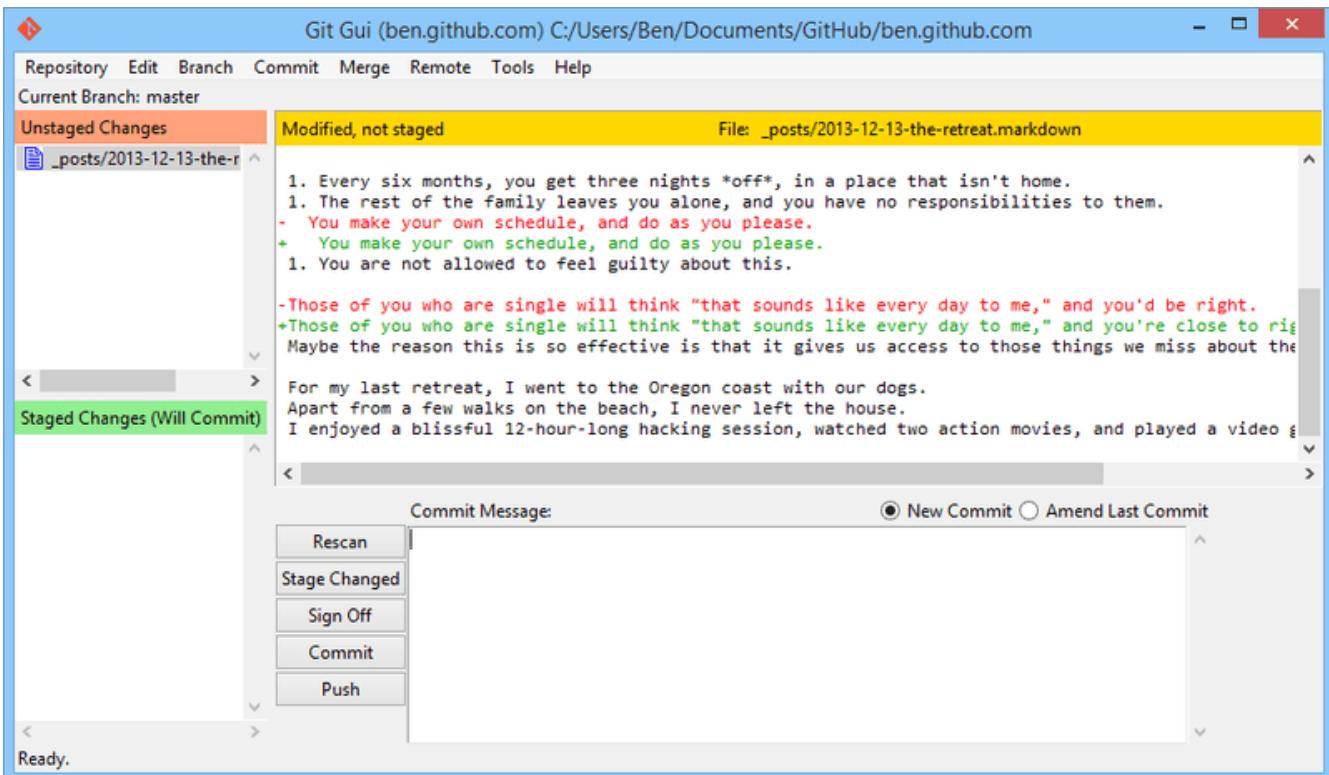
Слика 151. `gitk` приказивач историје

На врху се налази нешто што личи на излаз команде `git log --graph`; свака тачка представља комит, линије представљају родитељске везе, а референце су представљене обојеним правоугаоницима. Жута тачка представља HEAD, а црвена тачка представља измене које још увек нису постале комит. У дну се налази поглед изабраног комита; коментари и закрпа на левој страни, а прегледни поглед са десне стране. Између се налази скуп контрола које се користе за претрагу историје.

С друге стране, `git-gui` је првенствено, алат за изградњу комитова. И он се такође најједноставније покреће из командне линије:

```
$ git gui
```

И изгледа овако некако:



Слика 152. `git-gui` алам за комитовање

На левој страни је индекс; измене које нису на стејџу се налазе на врху, измене на стејџу су на дну. Комплетне фајлове можете да премештате из стања у стање тако што кликнете на њихове иконе, или можете да изаберете фајл који желите да погледате кликом на његово име.

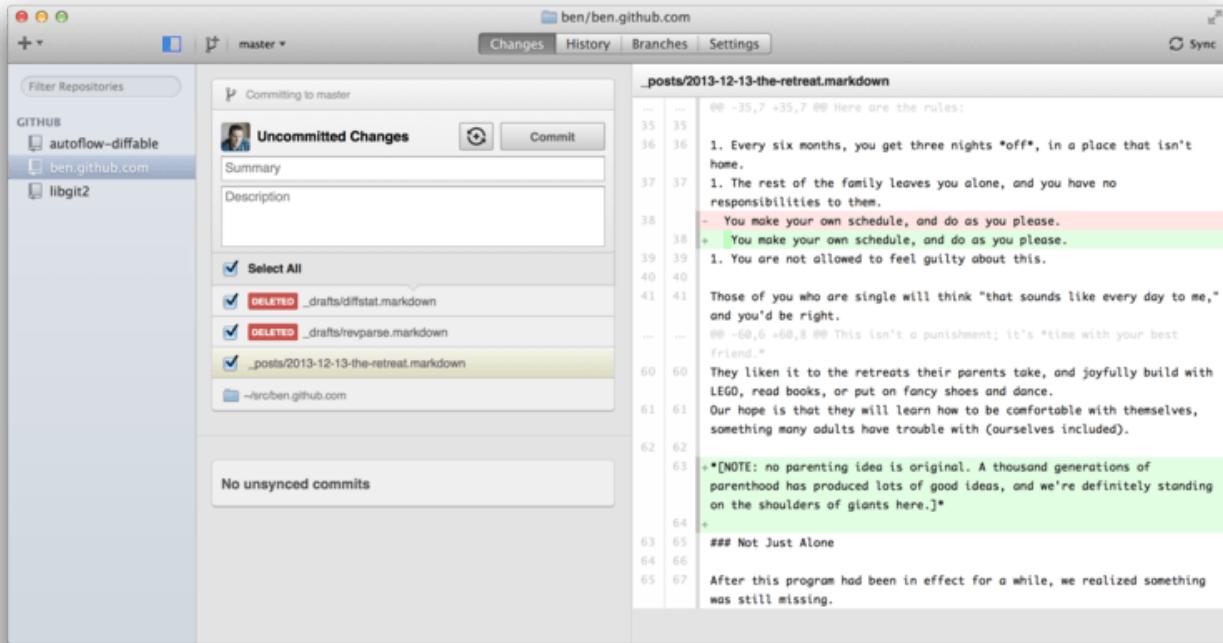
Горе десно се налази поглед разлике који приказује разлике за тренутно изабрани фајл. Поједине комаде (или поједине линије) можете да ставите на стејџ тако што урадите десни клик у овој области.

Доле десно је област за поруку и акције. Када желите да урадите нешто слично са `git commit`, откуцајте своју поруку у текст поље и кликните „Commit”. Можете изабрати да преправите последњи комит тако што кликнете на „Amend” радио дугме, па ће то да ажурира „Staged Changes” област са садржајем последњег комита. Затим једноставно можете да неке измене ставите на стејџ или да их уклоните са њега, измените комит поруку, па поново кликнете на „Commit” да замените стари комит са новим.

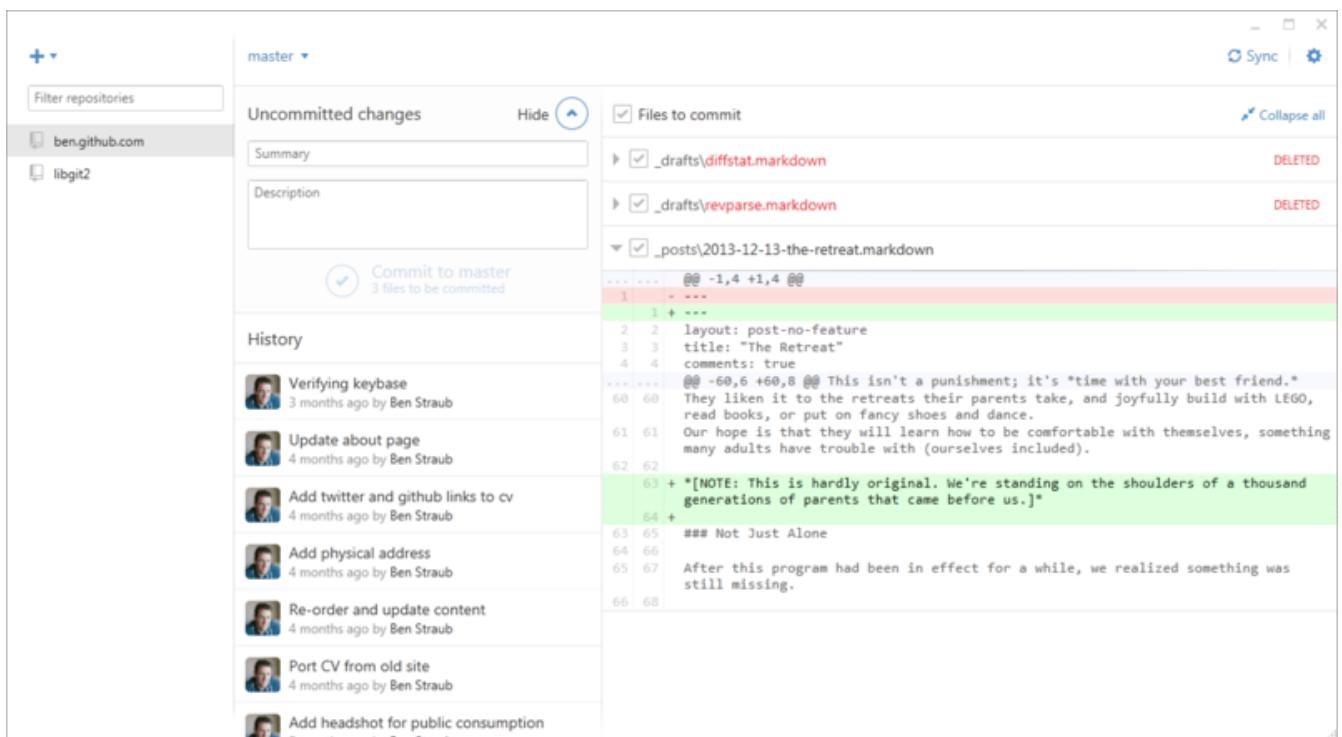
`gitk` и `git-gui` су примери алата који су оријентисани на задатак. Сваки је прилагођен специфичној намени (прегледу историје и креирању комитова, респективно), и изостављају могућности које нису неопходне за тај задатак изостављају могућности које нису неопходне за тај задатак.

GitHub за мекОС и Виндоуз

GitHub је креирао два Гит клијента оријентисана на процес рада: један за Виндоуз и један за мекОС. Ови клијенти су добар пример алата оријентисаних на процес рада – уместо да изложе комплетну функционалност Гит програма, они се фокусирају на одабрани скуп могућности које се уобичајено користе у добро функционишу заједно. They look like this:



Слика 153. GitHub за мекОС



Слика 154. GitHub за Виндоуз

Дизајнирани су тако да изгледају и раде врло слично, тако да ћемо их у овом поглављу третирати као да су један производ. Нећемо проћи кроз детаљно извршавање ових алата (они имају сопствену документацију), али следи брзо представљање „changes” погледа (што је место на којем проводите највећи део свог времена).

- Са леве стране је листа репозиторијума које клијент прати; репозиторијум можете додати (било клонирањем, било локалним прикачивањем) тако што кликнете на икону „+” у врху ове области.

- У средини се налази област за унос комита која вам омогућава да унесете комит поруку и да изаберете фајлове који ће ући у комит. На Виндоуз систему се комит историја проказује непосредно испод овога; на мекОС, она се налази у одвојеној картици.
- Са десне стране је погледа разлике који приказује шта се изменило у радном директоријуму, или које су измене у изабраном комиту.
- Последња ствар коју треба приметите је дугме „Sync” у горњем десном углу које представља основни начин за интеракцију преко мреже.



Да бисте користили ове алате, није потребно да поседујете *GitHub* налог. Мада су дизајнирани да истакну *GitHub* сервис и препоручени процес рада, фино ће радити са било којим репозиторијумом и обављаће мрежне операције са било којим Гит хостом.

Инсталација

GitHub за Виндоуз може да се преузме са адресе <https://windows.github.com>, а *GitHub* за мекОС са <https://mac.github.com>. Када се апликације покрену по први пут, оне вас воде кроз комплетно подешавање програма Гит, као што је конфигурисање вашег имена и имејл адресе и оба постављају разумне подразумеване вредности за већину конфигурационих опција, као што су кешеви акредитива и CRLF понашање.

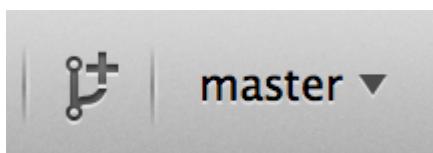
Ова програма су „evergreen” – ажурирања се преузимају и инсталирају у позадини док се апликације извршавају. Ово згодно укључује и верзију програма Гит који се испоручује уз апликације, а то значи на више нећете морати да водите рачуна о ручном ажурирању. На Виндоуз систему, клијент има и пречицу за покретање *Powershell* са *Posh-git*, о којем ћемо говорити касније у овом поглављу.

Следећи корак је да алатима доставите неколико репозиторијума са којима ће да раде. Клијент вам приказује листу репозиторијума за које имате приступ на сервису *GitHub* и може да их клонира у једном кораку. Ако већ имате локални репозиторијум, једноставно превуците његов директоријум из *Finder* или *Windows Explorer* прозора у прозор *GitHub* клијента и он ће постати део листе репозиторијума са леве стране.

Препоручени процес рада

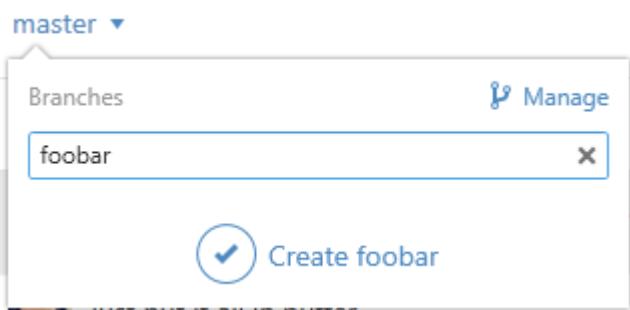
Једном када се инсталира и подеси, *GitHub* клијент можете користити за многе уобичајене Гит послове. Предвиђени процес рада се понекада назива *GitHub* процес. Ово представљамо са више детаља у [GitHub процес рада](#), али општи *gist* је да ћете (а) комитовати у грану, и (б) прилично редовно ћете се синхронизовати са удаљеним репозиторијумом.

Управљање гранама је једна од ствари у којој се ова два алата разилазе. За креирање нове гране на мекОС постоји дугме на врху прозора:



Слика 155. „Create Branch” дугме на мекОС

На Виндоуз систему, ово се ради уношењем имена нове гране у вицет за прелаз на другу грану:



Слика 156. Креирање гране на Виндоуз систему

Када направите грану, креирање нових комитова је просто. Направите мало измена у радном директоријуму, па када пређете у прозор GitHub клијента, он ће вам приказати измене фајлове. Унесите комит поруку, изаберите фајлове које желите да укључите у комит и кликните на „Commit” button (ctrl-ентер или -ентер).

Главни начин путем којег имате интеракцију преко мреже са осталим репозиторијумима је употребом „Sync” могућности. Програм Гит интерно има одвојене операције за гурање преузимање, спајање и ребазирање, али GitHub клијенти их све стапају у једну могућност у више корака. Ево шта се дешава када кликнете на Sync дугме:

1. `git pull --rebase`. Ако ово не успе због конфликта при спајању, прибегава на `git pull --no-rebase`.
2. `git push`.

Ово је најубичајенији низ мрежних команда када се ради у овом стилу, тако да њихово сабирање у једну команду штеди доста времена.

Резиме

Ови алати су веома погодни за процес рада којем су намењени. Програмери и они који то нису, час посла могу кренути да сарађују на пројекту. Међутим, ако се ваш процес рада разликује, или ако желите више контроле над тиме како и када се врше мрежне операције, препоручујемо вам да користите неки други клијент или командну линију.

Остали ГКИ

Постоји већи број других графичких Гит клијената који су део гаме од специјализованих алата за једну намену, па све до апликација које покушавају да изложе све што програм Гит може. Званични Гит веб сајт има одржавану листу најпопуларнијих клијената на адреси <http://git-scm.com/downloads/guis>. Свеобухватнија листа постоји на Фит вики сајту, на адреси https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Гит у Visual Studio

Почевши од Visual Studio 2019 верзије 16.8, Visual Studio има Гит алат уграден директно у ИДЕ.

Алат подржава следеће Гит функционалности:

- Креирање или клонирање репозиторијума.
- Отварање и прглед историје репозиторијума.
- Креирање и одјављивање грана и ознака.
- Скривање, постављање на стејџ и комитовање измена.
- Преузимање, повлачење, гурање или синхронизација комитова.
- Спајање и ребазитање грана.
- Разрешавање конфликтата при спајању.
- Преглед разлика.
- ... и још!

Да сазнате више, прочитајте [званичну документацију](#).

Гит у Visual Studio Code

У *Visual Studio Code* је уgraђена гит подршка. Потребно је да имате инсталан програм Гит верзије 2.0.0 (или новије).

Главне могућности су:

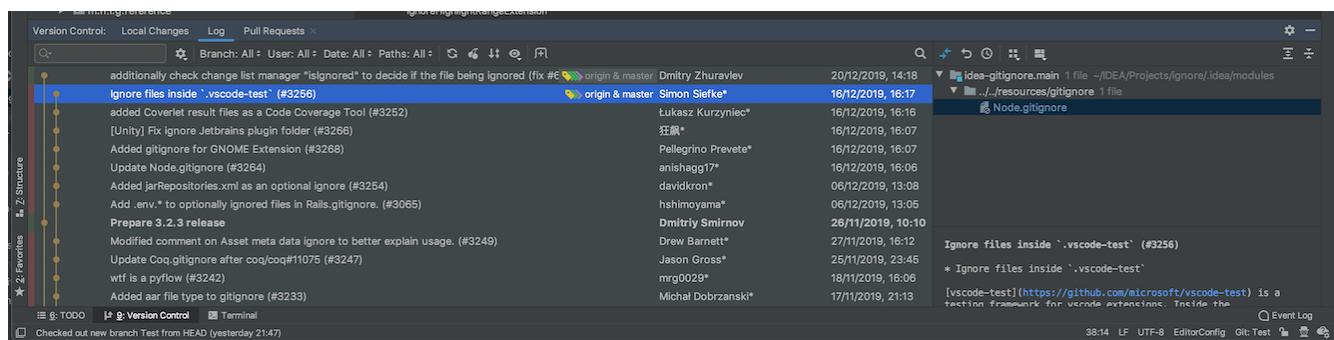
- Преглед разлика између фајла који уређујете и оног са стране.
- Гит Статусна Трака (доле лево) приказује текућу грану, индикатори запрљаности, долазни и одлазни комитови.
- Већину уобичајених гит операција можете да урадите из едитора:
 - Иницијализација репозиторијума.
 - Клонирање репозиторијума.
 - Креирање грана и ознака.
 - Стављање на стејџ и комитовање измена.
 - Гурање/повлачење/синхронизација са удаљеном граном.
 - Разрешавање конфликтата при спајању.
 - Преглед разлика.
- Уз помоћ проширења, такође можете да обрађујете и *GitHub* захтевеза повлачење: <https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github>.

Званична адреса се налази на адреси: <https://code.visualstudio.com/Docs/editor/versioncontrol>.

Гит у IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine

JetBrains базирани ИДЕ (као што су *IntelliJ IDEA*, *PyCharm*, *WebStorm*, *PhpStorm*, *RubyMine* и

остали) се испоручују са додатком за Гит интеграцију. Он обезбеђује посебан поглед у ИДЕ за рад са програмом Гит и GitHub захтевима за повлачење.



Слика 157. Version Control ToolWindow у JetBrains ИДЕ

Интеграција се ослања на гит клијент из командне линије, па је потребно да се неки од њих инсталира. Званична документација се налази на адреси <https://www.jetbrains.com/help/idea/using-git-integration.html>.

Гит у Sublime Text

Почевши од верзије 3.2 надаље, *Sublime Text* поседује гит интеграцију у едитору.

Могућности су следеће:

- Трака са стране ће приказивати гит статус фајлова и директоријума беџом/иконом.
- Фајлови и директоријуми у `.gitignore` фајлу ће бити затамњени у траци са стране.
- У статусној траци можете да видите тренутну гит грани и број начињених измена.
- Сада се помоћу маркера виде све измене над фајлом.
- Део *Sublime Merge* функционалности гит функционалности из самог *Sublime Text*. За ово је потребно да буде инсталiran *Sublime Merge*. Погледајте: <https://www.sublimemerge.com/>.

Званична документација за *Sublime Text* се налази на адреси: https://www.sublimetext.com/docs/3/git_integration.html.

Гит унутар Bash

Ако сте Bash корисник, можете да се прикачите на неке од могућности ваше љуске и своје искуство са програмом Гит учините много пријатнијим. Програм Git се уствари испоручује са додацима за неколико љуски, али то није подразумевано укључено.

Најпре је потребно да преузмете копију фајла за довршавање из изворног кода издања програма Гит који користите. Верзију проверите са `git version`, па онда употребите `git checkout tags/vX.Y.Z`, где `vX.Y.Z` одговара верзији програма Гит коју користите. Копирајте `contrib/completion/git-completion.bash` фајл на неко згодно место, као што је ваш почетни директоријум, па додајте следеће у свој `.bashrc` фајл:

```
. ~/git-completion.bash
```

Када се то изврши, промените директоријум на гит репозиторијум и откуцајте:

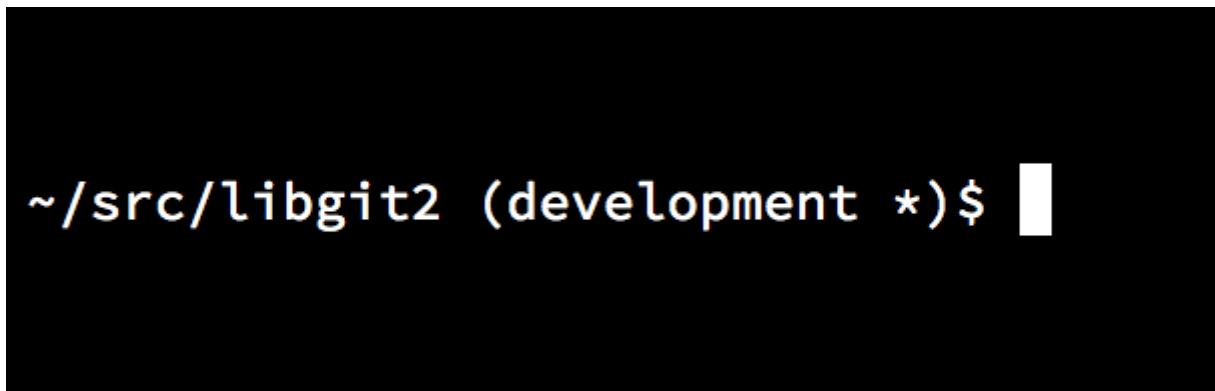
```
$ git chec<tab>
```

...и Bash ће то аутоматски да доврши на `git checkout`. Ово функционише за све Гит подкоманде, параметре командне линије, и имена удаљених репозиторијума и референци тамо где је пригодно.

Такође је корисно и да прилагодите свој одзив тако да прикаже информације о Гит репозиторијуму текућег директоријума. Ово може бити једноставно или компликовано колико год желите, али у општем случају постоји неколико информација које жели да види већина људи, као што је текућа грана и статус радног директоријума. Да бисте их додали у свој одзив, једноставно копирајте фајл `contrib/completion/git-prompt.sh` из репозиторијума Гит изворног кода у свој почетни директоријум и додајте нешто слично следећем у свој `.bashrc` фајл:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(_git_ps1 " (%s)")\$ '
```

`\w` значи прикажи текући радни директоријум, `\$` исписује \$ део одзыва, а `_git_ps1 " (%s)"` позива функцију коју обезбеђује `git-prompt.sh` са аргументом форматирања. Сада ће ваш bash одзив да изгледа овако када се налазите било где унутар пројекта који контролише Гит:



Слика 158. Прилагођени bash одзив

Обе ове скрипте долазе са корисном документацијом; за више информација, погледајте садржај `git-completion.bash` `git-prompt.sh`.

Гит у Zsh

Zsh љуска се испоручује са библиотеком таб-довољавања за програм Гит. Сада бисте је користили, једноставно извршите `autoload -Uz compinit && compinit` у вашем `.zshrc` фајлу. Интерфејс Zsh љуске је моћнији од оног у Bash:

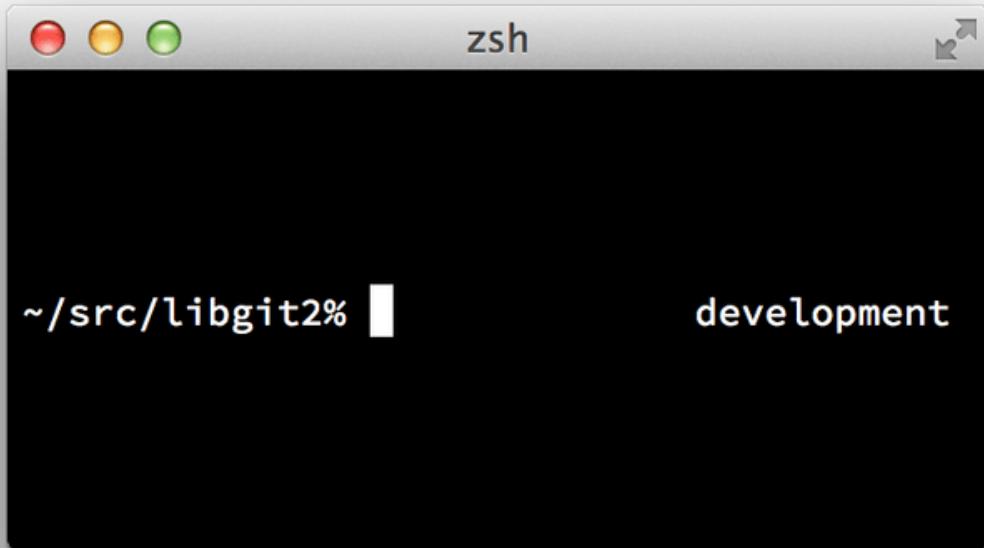
```
$ git che<tab>
check-attr      -- display gitattributes information
check-ref-format -- ensure that a reference name is well formed
checkout        -- checkout branch or paths to working tree
checkout-index   -- copy files from index to working directory
cherry          -- find commits not merged upstream
cherry-pick     -- apply changes introduced by some existing commits
```

Двосмислена таб-доворшавања нису само наведена; уз њих се наводе и корисни описи, а можете и графички да се крећете кроз листу тако што настављате да притискате тастер таб. Ово функционише са Гит командама, њиховим аргументима и именима ствари у репозиторијуму (као што су референце и удаљени репозиторијуми), као и са именима фајлова и свим осталим стварима које Zsh зна како да таб-доворши.

Zsh љуска се испоручује са оквиром за добијање информација из система за контролу верзије, под називом `vcs_info`. Ако на десној страни желите да видите име гране, додајте следеће линије у свој `~/.zshrc` фајл:

```
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'
```

Резултат овога је да се на десној страни прозора терминала приказује текућа грана, кадгод се љуска налази у Гит репозиторијуму. Такође се подржава и лева страна, наравно; једноставно уклоните коментар са доделе за PROMPT. Изгледа отприлике овако:

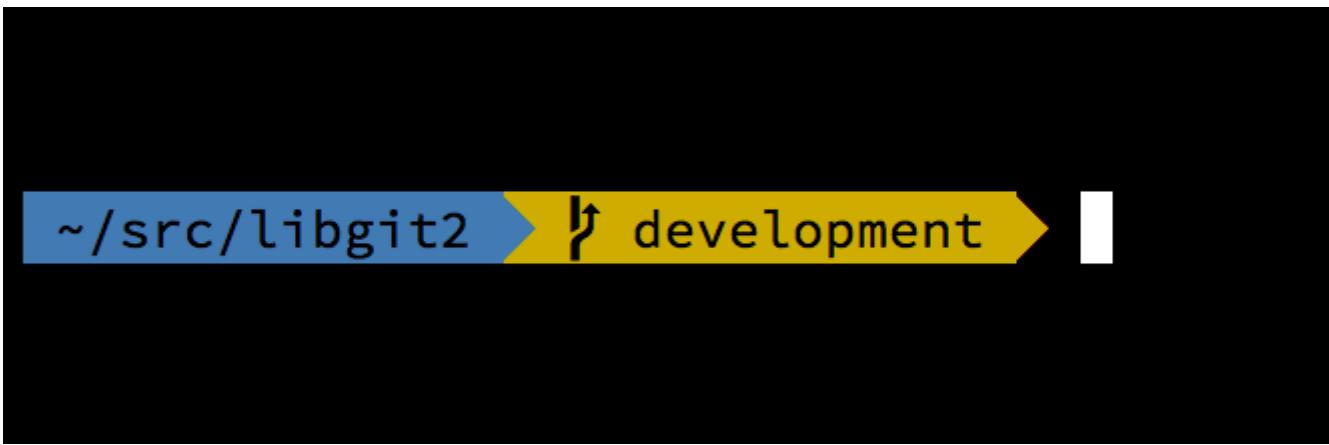


Слика 159. Прилагођени zsh одзив

За виђе информација у вези `vcs_info`, погледајте његову документацију у `zshcontrib(1)` страници упутства, или на мрежи на адреси <http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information>.

Можда ће вам уместо `vcs_info` више одговарати скрипта за прилагођавање одзива која се испоручује уз програм Гит, под називом `git-prompt.sh`; за детаље погледајте <https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh>. `git-prompt.sh` је компатибилна и са Bash и са Zsh.

Zsh љуска је довољно моћна да постоје комплетни радни оквири који служе за њено побољшање. Један од њих се назива „oh-my-zsh” и може да се нађе на адреси <https://github.com/robbyrussell/oh-my-zsh>. Систем додатака за `oh-my-zsh` долази за моћним таб-доворшавању за Гит и има разне „теме” одзива, од којих многе приказују податке из контроле верзије. Пример `oh-my-zsh` теме је само један од примера шта може да се постигне овим системом.



Слика 160. Пример *oh-my-zsh* теме.

Гит у Powershell

Стари терминал командне линије на Виндоуз систему ([cmd.exe](#)) није баш способан за прилагођено Гит искуство, али ако користите *Powershell*, имате среће. Ово такође функционише ако извршавате *PowerShell Core* на Линукс или мекОС систему. Пакет под именом *Posh-Git* (<https://github.com/dahlbyk/posh-git>) обезбеђује моћне системе за табдовршавање, као и побољшани одзив који вам помаже да имате једноставан поглед на статус репозиторијума. Изгледа овако:

```
posh-git [master] ~ PowerShell 6.1.1 64-bit (18588)
~\GitHub\dahlbyk\posh-git [master ≡ +1 ~0 -0 | +0 ~1 -0 !]>
```

Слика 161. Powershell са Posh-git

Инсталација

Предуслови (само за Виндоуз)

Пре него што будете у стању да на вашој машини извршавате *PowerShell* скрипте, морате да поставите локалну *ExecutionPolicy* на *RemoteSigned* (у суштини на било шта осим *Undefined* и *Restricted*). Ако сте уместо *RemoteSigned* избрали *AllSigned*, онда и локалне скрипте (које сте ви написали) морају бити дигитално потписане да би могле да се извршавају. Са *RemoteSigned*, само скрипте којима је *ZoneIdentifier* постављен на *Internet* (односно које су преузете са интернета) могу бити потписане, остале не морају. Ако сте администратор и ово желите да поставите за све кориснике на машини, употребите *-Scope LocalMachine*. Ако сте обични корисник, без административних права, употребите *-Scope CurrentUser* да полису

поставите само за свој налог.

Више о *PowerShell* опсезима: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes.

Више о *PowerShell ExecutionPolicy*: <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy>.

Ако желите да поставите вредност *ExecutionPolicy* на *RemoteSigned* за све кориснике, употребите следећу команду:

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

PowerShell Gallery

Ако имате инсталiran барем *PowerShell 5* или *PowerShell 4* са инсталirаним *PackageManagement*, за инсталацију *posh-git* можете да употребите менаџер пакета.

Више информација о *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 *posh-git* желите да инсталирате за све кориснике, употребите *-Scope AllUsers* и извршите команду из *PowerShell* конзоле са администраторским правима. У случају да друга команда не успе да се изврши и врати нешто као *Module 'PowerShellGet' was not installed by using Install-Module*, прво ћете морати да извршите једну другу команду:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

Затим можете да се вратите и покушате поново. Ово се дешава јер су модули који се испоручују са *Windows PowerShell* потписани са другачијим сертификатом издавача.

Ажурирање PowerShell одзива

Да бисте у одзив укључили гит информације, потребно је да увезете *posh-git* модул. Ако желите да се *posh-git* увози сваки пут када се покрене *PowerShell*, извршите *Add-PoshGitToProfile* команду која ће у вашу *\$profile* скрипту додати наредбу за увоз. Ова скрипта се увози сваки пут када отворите *PowerShell* конзолу. Имајте на уму да постоји више *\$profile* скрипти. Нпр. једна за конзолу и друга за *ISE* (интегрисано скрипт окружење).

```
> Import-Module posh-git  
> Add-PoshGitToProfile -AllHosts
```

Из извornog кода

Једноставно преузмите *posh-git* издање са адресе <https://github.com/dahlbyk/posh-git/releases> и распакујте га. Затим уvezите модул користећи пуну путању до **posh-git.psd1** фајла:

```
> Import-Module <path-to-uncompress-folder>\src\posh-git.psd1  
> Add-PoshGitToProfile -AllHosts
```

То ће додати одговарајућу линију у ваш **profile.ps1** фајл и *posh-git* ће бити активан следећи пут када покренете *PowerShell*.

За опис информација о Гит статусу које се приказују у одзиву погледајте: <https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-summary-information> За више детаља о начину да прилагодите *posh-git* одзив, погледајте: <https://github.com/dahlbyk/posh-git/blob/master/README.md#customization-variables>.

Резиме

Научили сте како да из алата које свакодневно користите укротите снагу програма Гит, а такође и како да приступите Гит репозиторијумима из ваших програма.

Додатак В: Уграђивање програма Гит у ваше апликације

Ако је ваша апликација намењена програмерима, велике су шансе да би јој користила интеграција са контролом извornог кода. Чак би и апликације који нису намењене програмерима, као што су едитори докумената, потенцијално могле да имају користи од могућности контроле верзије, а модел програм Гит врло добро функционише у многим различитим сценаријима.

Ако је потребно да програм Гит интегришете са својом апликацијом, у суштини имате две опције: покрените љуску и позовите `git` програм из командне линије, или уградите Гит библиотеку у своју апликацију. Овде ћемо приказати интеграцију са командном линијом, као и неколико најпопуларнијих Гит библиотека намењених за уградњу.

Гит из командне линије

Једна опција је да се покрене процес љуске и да се употреби Гит алат из командне линије да обави посао. Предност овога је што је канонски и што су подржане све могућности програма Гит. Такође је прилично једноставно, јер већина окружења за време извршавања имају релативно просте начине за позивање процеса са аргументима командне линије. Ипак, овај процес има и неке мане.

Једна је што је комплетан излаз чисти текст. Ово значи да ћете морати да парсирате излазни формат програма Гит који се временама на време мења да би очитали информације о напретку и резултатима, а то може бити неефикасно и склоно грешкама.

Друга је недостатак опоравка од грешке. Ако је репозиторијум некако оштећен, или корисника има погрешно формирану конфигурациону вредност, програм Гит ће једноставно одбити да изврши многе операције.

Још једна је управљање процесом. Програм Гит од вас захтева да одржавате окружење љуске у одвојеним процесима, што може додати нежељену сложеност. Покушај да се координишу многи овакви процеси (посебно када се потенцијално истом репозиторијуму приступа из неколико репозиторијума) може бити прилично захтеван.

Libgit2

Још једна доступна опција је да употребите *Libgit2*. *Libgit2* је Гит имплементација која нема зависности, са тежиштем на лепом API која је предвиђена за употребу у другим програмима. Можете да је пронађете на адреси <http://libgit2.github.com>.

Најпре, хајде да погледамо како изгледа С API. Ево вртоглаве турнеје:

```

// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);

```

Првих неколико линија отвара Гит репозиторијум. Тип `git_repository` представља ручку репозиторијума са кешом у меморији. Ово је најједноставнија метода, када тачно знате путању до радног директоријума репозиторијума, или до `.git` директоријума. Постоји и `git_repository_open_ext` који има опције претраге, `git_clone` и пријатељи за креирање локалног клона удаљеног репозиторијума и `git_repository_init` за креирање потпуно новог репозиторијума.

Друго парче кода користи *rev-parse* синтаксу (за више информација о овоме, погледајте [Референце грана](#)) да дође до комита на који показује HEAD. Повратни тип је `git_object` показивач, који представља нешто што постоји у Гит бази података објекта за репозиторијум. `git_object` је уствари „родитељски” тип за неколико различитих врста објекта; распоред меморије за сваку врсту „дете” типова је исти као за `git_object`, тако да безбедно можете да кастујете у одговарајући. У овом случају, `git_object_type(commit)` би вратило `GIT_OBJ_COMMIT`, тако да је безбедно да се кастује у `git_commit` показивач.

Следеће парче показује како да се приступи особинама комита. Последња линија овде користи тип `git_oid`; ово је *Libgit2* репрезентација SHA-1 хеша.

Из овог примера, почело је да се појављује неколико шаблона:

- Ако декларишете показивач и проследите референцу на њега у *Libgit2* позив, тај позив ће вероватно да врати целобројни код грешке. Вредност `0` означава успех; све мање од тога је грешка.
- Ако *Libgit2* за вас попуни показивач, ваш је задатак да га ослободите.
- Ако *Libgit2* из позива врати `const` показивач, не морате да га ослободите, али ће престати да важи када се ослободи објекат којем припада.
- Писање С кода је помало болно.

Последње значи да је мало вероватно да ћете писати на језику С када користите *Libgit2*. На

срећу, постоји већи број везивања за одређене језике која знатно олакшавају рад са Гит репозиторијумима из вашег одређеног језика и окружења. Хајде да погледамо претходни пример написан Руби везивањима за *Libgit2*, која се зову *Rugged*, и можете да их пронађете на адреси <https://github.com/libgit2/rugged>.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Као што видите, код је много мање запетљан. Прво, *Rugged* користи изузетке; може да баши ствари као што су *ConfigError* или *ObjectError* којима сигнализира стања грешке. Друго, нема експлицитног ослобађања ресурса, јер језик Руби скупља ћубре. Хајде да погледамо мало компликованији пример: креирање комита од самог почетка:

```
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 Креира нови блоб у којем се налази садржај новог фајла.

② Попуњава индекс са стаблом head комита, па додаје нови фајл на путању *newfile.txt*.

③ Ово креира ново стабло у ODB (бази података објекта), па га користи за нови комит.

④ Користимо исти потпис и за поље аутора и за поље комитера.

⑤ Комит порука.

⑥ Када се креира комит, морате да наведете родитеље новог комита. Ово користи врх HEAD референце као једног родитеља.

⑦ *Rugged* (и *Libgit2*) могу необавезно да ажурирају референцу када праве комит.

- ⑧ Враћена вредност је SHA-1 хеш новог комит објекта коју онда можете употребити да добијете `Commit` објекат.

Руби кôд је фин и чист, али пошто *Libgit2* диже највећи део тежине, овај кôд ће се такође и извршавати прилично брзо. Ако нисте рубиста, дотичемо се и осталих везивања у [Остале везивања](#).

Напредна функционалност

Libgit2 поседује неколико могућности вам опсега Гит језгра. Један од примера је проширивост: *Libgit2* вам дозвољава да доставите прилагођене „позадинске механизме” за неколико врста операција, тако да ствари можете складиштити на начин који се разликује од онога који користи програм Гит. *Libgit2* између осталог дозвољава прилагођене позадинске механизме за конфигурацију, складиштење референци и базу објеката.

Хајде да видимо како ово функционише. Кôд који следи је позајмљен из скупа примера позадинских механизама које је представио *Libgit2* тим (и могу да се пронађу на адреси <https://github.com/libgit2/libgit2-backends>). Ево како се поставља прилагођени позадински механизам за базу података објекта:

```
git_odb *odb;
int errgor = git_odb_new(&odb); ①

git_odb_backend *my_backend;
errgor = git_odb_backend_mine(&my_backend, /*...*/); ②

errgor = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
errgor = git_repository_open(&repo, "some-path");
errgor = git_repository_set_odb(repo); ④
```

Приметите да се грешке хватају, али се не обрађују. Надамо се да је ваш кôд бољи од нашег.

- ① Иницијализује „чеони део” празне базе података објекта (ODB), која ће служити као контејнер за позадинске механизме који уствари одрађују прави посао.
- ② Иницијализује прилагођени ODB позадински механизам.
- ③ Додаје позадински механизам у чеони део.
- ④ Отвара репозиторијум и подешава га да користи нашу ODB за претрагу објеката.

Али шта је та `git_odb_backend_mine` ствар? Па, то је конструктор ваше сопствене ODB имплементације и ту можете да урадите штагод желите, док год исправно попуните `git_odb_backend` структуру. Ево како то могло да изгледа:

```

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;
}

```

Најсуптилније ограничење овде је да први члан `my_backend_struct` мора бити `git_odb_backend` структура; то обезбеђује да распоред меморије буде онакав какав очекује *Libgit2* код. Остатак је произвољан; ова структура можете бити велика или мала колико год је то потребно.

Функција иницијализације алоцира нешто меморије за структуру, поставља прилагођени контекст, па затим попуњава чланове `parent` структуре коју подржава. Погледајте фајл `include/git2/sys/odb_backend.h` у *Libgit2* изворном коду да видите комплетан скуп потписа позива; ваш дати случај коришћења ће вам помоћи да одредите које од њих ћете пожелети да подржите.

Остале везивања

Libgit2 има везивања за много језика. Ево малог примера који користи неколико комплетних пакета везивања у време писања овог текста; постоје библиотеке за многе друге језике, укључујући C++, Go, Node.js, Erlang и JVM, сви у различитим фазама зрелости. Званична колекција везивања може да се пронађе прегледом репозиторијума на адреси <https://github.com/libgit2>. Код који ћемо ми написати ће да враћа комит поруку из комита на који коначно указује HEAD (нешто као `git log -1`).

LibGit2Sharp

Ако пишете .NET или Mono апликацију, LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>) је оно што тражите. Везивања су написана ма C# и посвећено је доста пажње да се сирови *Libgit2* позиви умотају у CLR API који изгледају природно. Ево како изгледа наш програм:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

За десктоп Виндоуз апликације, постоји чак и *NuGet* пакет који ће вам помоћи да брзо почнете.

objective-git

Ако се ваша апликација извршава на Епл платформи, највероватније користите Objective-C као свој језик имплементације. *Objective-Git* (<https://github.com/libgit2/objective-git>) је име *Libgit2* везивања за то окружење. Пример програма изгледа овако:

```
GTRepository *repo =  
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]  
error:NULL];  
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git у потпуности саражује са *Swift*, тако да се не морате плашити ако сте оставили *Objective-C*.

pygit2

Libgit2 везивања за Пајтон се називају *Pygit2* и можете да их пронађете на адреси <http://www.pygit2.org/>. Наш пример програма:

```
pygit2.Repository("/path/to/repo") # open repository  
    .head                         # get the current branch  
    .peel(pygit2.Commit)           # walk down to the commit  
    .message                       # read the message
```

Наставак читања

Наравно, потпуни третман *Libgit2* могућности је ван опсега ове књиге. Ако желите више информација о самом *Libgit2*, на адреси <https://libgit2.github.com/libgit2> постоји API документација, а скуп водича на адреси <https://libgit2.github.com/docs>. За остале везивања, проверите испоручени *README* и тестове; тамо често постоје мали туторијали и смернице за наставак читања.

JGit

Ако Гит желите да користите из Јава програма, постоји Гит библиотека са комплетним могућностима, пос називом *JGit*. *JGit* је релативно комплетна Гит имплементација написана природно на Јави и доста је користи у Јава заједници. *JGit* пројекат је под *Eclipse* кишобраном и његова почетна страница се налази на адреси <https://www.eclipse.org/jgit>.

Почетно подешавање

Постоји већи број начина да повежете свој пројекат са *JGit* и почнете да пишете код уз њега. Вероватно најлакши је да користите *Maven* – интеграција се постиже додавањем следећег исечка у `<dependencies>` ознаку у `pom.xml` фајл:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

У време када ово читате, `version` је највероватније мало напредовала; проверите <https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> да погледате ажуриране информације о репозиторијуму. Када се изврши овај корак, *Maven* ће аутоматски да преузме и употреби *JGit* библиотеке које су вам потребне.

Ако радије сами желите да управљате зависностима, на адреси <https://www.eclipse.org/jgit/download> можете пронаћи већ изграђене *JGit* бинарне фајлове. На следећи начин можете да их уградите у свој пројекат:

```
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
```

Водовод

JGit поседује два основна API нивоа: водовод и порцелан. Ова терминологија долази из самог програма Гит, а *JGit* је грубо говорећи подељен на исте врсте области: порцелан API су пријатни приступ уобичајеним акцијама на корисничком нивоу (она врста ствари коју би обични корисник обављао користећи Гит алат из команде линије), док су водоводни API за директну интеракцију са објектима репозиторијума ниског нивоа.

Почетна тачка већине *JGit* сесија је `Repository` класа и прва ствар коју желите да урадите је да креирате њену инстанцу. За репозиторијум базиран на фајл систему (тако је, *JGit* омогућава и друге моделе складиштења), ово се постиже употребом `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

Изграђивач поседује течни API којим достављање свих ствари које су му потребне да пронађе Гит репозиторијум, без обзира на то да ли ваш програм зна или не где се он тачно налази. Може да користи променљиве окружења (`.readEnvironment()`), крене са места у радном директоријуму и претражује (`.setWorkTree(...).findGitDir()`), или једноставно отвори познати `.git` директоријум као што је приказано изнад.

Када дођете до `Repository` инстанце, можете да урадите свашта са њом. Ево у кратким цртама неких могућности:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

Овде се свашта дешава, па хајде да опишемо одељке, један по један.

Прва линија добавља показивач на `master` референцу. *JGit* аутоматски прибавља *стварну* мастер референцу која се налази у `refs/heads/master`, и враћа објекат који вам омогућава да дођете до информација у вези референце. Можете да добијете име (`.getName()`), било циљни објекат директне референце (`.getObjectId()`) или референцу на коју показује симболичка референца (`.getTarget()`). *Ref* објекти се користе и за представљање референци ознака и објеката, тако да можете питати да ли је ознака „ољуштена”, што значи да показује на крајњи циљ (потенцијално дугачког) стринга објеката ознака.

Друга линија враћа циљ `master` референце који се добија као `ObjectId` инстанца. `ObjectId` представља SHA-1 хеш објекта који може а не мора да постоји у Гит бази објеката. Трећа линија је слична, али приказује како *JGit* обрађује *rev-parse* синтаксу (за више о овоме,

погледајте [Референце грана](#)); можете да проследите било који спецификатор објекта који програм Гит разуме, и *JGit* ће вратити или важећи *ObjectId* за тај објекат, или `null`.

Наредне две линије показују како да се учита сирови садржај неког објекта. У овом примеру позивамо `ObjectLoader.copyTo()` да стримује садржај објекта директно на *stdout*, али *ObjectLoader* такође има методе за читање типа и величине објекта, као да га врати као низ бајтова. За велике објекте (где `.isLarge()` враћа `true`), можете да позовете `.openStream()` и добијете објекат који личи на *InputStream* и који може да прочита податке сировог објекта без потребе да га целог довлачи у меморију.

Наредних неколико линија приказују шта је потребно да се креира нова грана. Креирало *RefUpdate* инстанцу, конфигуришемо неке параметре, па позовемо `.update()` да окинемо измену. Непосредно затим следи код за брисање те исте гране. Приметите да је неопходно `.setForceUpdate(true)` да би ово радило; у супротном ће `.delete()` позив да врату `REJECTED` и ништа се неће дрогодити.

Последњи пример приказује како да се из Гит конфигурационих фајлова добије вредност опције `user.name`. Ова *Config* инстанца користи репозиторијум који смо раније отворили за локалну конфигурацију, али ће аутоматски да детектује фајлове глобалне и системске конфигурације и такође ће из њих а чита вредности.

Ово је само мали део комплетног водоводног API; доступно је још много метода и класа. Овде није приказан начин на који *JGit* обрађује грешке, а то је путем изузетака. *JGit* API понекад бацају стандардне Јава изузетке (као што је `IOException`), али доступна је и гомила *JGit* специфичних типова изузетака (као што су `NoRemoteRepositoryException`, `CorruptObjectException` и `NoMergeBaseException`).

Портулан

Водоводни APIs је прилично комплетан, али може бити незгодно да акције надовежу заједно како би се постигли уобичајени циљеви, као што је додавање фајл у индекс, или прављење новог комита. *JGit* обезбеђује скуп API вишег нивоа који помаже у томе и почетна тачка ових API је `Git` класа:

```
Repositoryrepo;  
// constructrepo...  
Gitgit = new Git(repo);
```

Гит класа поседује фин скуп метода вишег нивоа у *builder* стилу које могу да се употребе за конструкцију прилично сложеног понашања. Хајде да погледамо приме – постизање нечега као што је `git ls-remote`:

```

CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",
    "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}

```

Ово је уобичајен шаблон са Гит класом; методе враћају објекат команде који вам омогућава да уланчавате позиве метода како би поставили параметре, а који се затим извршавају када позовете `.call()`. У овом случају, питамо удаљени репозиторијум `origin` да нам пошаље ознаке, али не и главе. Приметите такође употребу `CredentialsProvider` објекта за аутентификацију.

Кроз Гит класу су доступне многе друге команде, укључујући, али не и само `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert` и `reset`.

Наставак читања

Ово је само мали део свих могућности библиотеке *JGit*. Ако вас интересује да научите још, ево где да потражите информације и инспирацију:

- Званична документација за *JGit* API је дотупна на мрежи, на адреси <https://www.eclipse.org/jgit/documentation>. То је стандардни *Javadoc*, тако да ће ваш омиљени *JVM IDE* такође моћи да је инсталира у локално.
- *JGit Cookbook* на адреси <https://github.com/centic9/jgit-cookbook> има много примера начина да се одређени задаци изврше са *JGit*.

go-git

У случају да Гит желите да интегришете у сервис написан на *Golang* језику, постоји такође и чиста *Go* имплементација библиотеке. Ова имплементација нема никакве природне зависности, па није склона грешкама услед ручног управљања меморијом. Такође је транспарентна за стандардне *Golang* алате за анализу перформанси као што су CPU, Профилисање меморије, детектор *race* услова, итд.

go-git је фокусиран на проширивост, компатибилност и подржава већину водоводног API, што је документовано у <https://github.com/go-git/go-git/blob/master/COMPATIBILITY.md>.

Ево основног примера употребе *Go* API:

```
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,
})
```

Чим дођете до `Repository` инстанце, можете приступити информацијама и извршити мутације над њом:

```
// retrieves the branch pointed by HEAD
ref, err := r.Head()

// get the commit object, pointed by ref
commit, err := r.CommitObject(ref.Hash())

// retrieves the commit history
history, err := commit.History()

// iterates over the commits and print each
for _, c := range history {
    fmt.Println(c)
}
```

Напредна функционалност

go-git има неколико запажених могућности, а једна од њих је систем складиштења који може да се проширује, што личи на *Libgit2* позадинске механизме. Подразумевана имплементација је складиштење унутар меморије, што је веома брзо.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{
    URL: "https://github.com/go-git/go-git",
})
```

Прошириво складиште нуди много интересантних опција. На пример, [https://github.com/go-git/go-git/tree/master/examples/storage\[\]](https://github.com/go-git/go-git/tree/master/examples/storage[]) вам омогућава да референце, објекте и конфигурацију складиштите у *Aerospike* бази података.

Још једна могућност је флексибилна апстракција фајл система. Употребом <https://pkg.go.dev/github.com/go-git/go-billy/v5?tab=doc#Filesystem> је једноставно да се сви фајлови чувају на различите начине нпр. да се сви упакују у једну архиву на диску или да се два чувају унутар меморије.

Још један напредни случај коришћења је фино подесиви HTTP клијент као што је онај који може да се нађе на адреси https://github.com/go-git/go-git/blob/master/_examples/custom_http/main.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
    },
}

// Override http(s) default protocol to use our custom client
client.InstallProtocol("https", githttp.NewClient(customClient))

// Clone repository using the new client if the protocol is https://
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})

```

Наставак читања

Потпуни третман *go-git* могућности излази ван оквира ове књиге. Ако желите више информација у вези *go-git*, на адреси <https://pkg.go.dev/github.com/go-git/go-git/v5> се налази API документација, а скуп примера употребе на адреси https://github.com/go-git/go-git/tree/master/_examples.

Dulwich

Постоји и чиста Пајтон Гит имплементација — *Dulwich*. Пројекат се хостује на <https://www.dulwich.io/>. Циљ је да обезбеди интерфејс ка гит репозиторијумима (и локалним и удаљеним) који не позивају директно програм Гит, већ уместо тога користе чисти Пајтон. Мада поседује и необавезна С проширења која знатно унапређују перформансе.

Dulwich следи дизајн програма Гит и раздаваја два основна API нивоа: водовод и порцелан.

Ево примера употребе API ниског нивоа да се приступи комит поруци последњег комита:

```

from dulwich.repo import Repo
r = Repo('.')
r.head()
# '57fbe010446356833a6ad1600059d80b1e731e15'

c = r[r.head()]
c
# <Commit 015fc1267258458901a94d228e39f0a378370466>

c.message
# 'Add note about encoding.\n'

```

Ако желите да испишете комит лог користећи порцелански API високог нивоа, извршите следеће:

```
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
```

Наставак читања

API документација, туторијал и многи примери начина на који се са *Dulwich* извршавају одређени задаци су доступни на званичном веб сајту <https://www.dulwich.io>.

Додатак С: Гит команде

Кроз ову књигу смо увели десетине Гит команди и заиста смо се трудали да их уводимо уз причу, полако додајући још команди како се прича развија. Међутим, због тога су примери употребе команди донекле разбацани по целој књизи.

У овом додатку ћемо проћи кроз све Гит команде које смо обрадили у књизи, грубо груписане по намени. О свакој команди ћемо говорити врло уопштено, па ћемо онда указати на место у књизи на којем смо је употребили.



Дугачке опције можете да скратите. На пример, можете да откуцате `git commit --a`, што је исто као да сте откуцали `git commit --amend`. Ово функционише само када су слова након `--` јединствена за једну опцију. Када пишете скрипте, немојте да скраћујете опције.

Подешавање и конфигурација

Постоје две команде које се прилично често употребљавају, од првог покретања програма Гит до уобичајеног свакодневног штеловања и референцирања, `config` и `help` команде.

git config

Програм Гит поседује подразумевани начин за извршавање стотина ствари. За већину њих, програму Гит можете наложити да их подразумевано обавља на другачији начин, или да поставите своје жељене поставке. У ово спада све од тога да програму Гит кажете које је ваше име до одређених боја које желите на терминалу или текст едитора који желите да користите. Постоји неколико фајлова које из којих ће ова команда да чита и у њих уписује, тако да вредности можете поставити глобално или само за одређене репозиторијуме.

Команда `git config` је коришћена у скоро сваком поглављу књиге.

У [Подешавања за први пут](#) смо је користили да наведемо своје име, имејл адресу и жељени едитор пре него што смо уопште и почели да користимо програм Гит.

У [Гит алијаси](#) смо показали како бисте могли да је употребите за креирање пречица команди које се развијају у дугачке низове опција, тако да не морате сваки пут да их куцате.

У [Ребазирање](#) смо је користили да `--rebase` буде подразумевана опција када извршите `git pull`.

У [Складиште акредитива](#) смо је употребили да поставимо подразумевано складиште за ваше НТТР лозинке.

У [Проширење кључних речи](#) смо показали како да поставите запрљане и чисте филтере над садржајем који излази и који улази у Гит.

Конечно, у суштини је комплетан садржај [Конфигурисање програма Гит](#) посвећен овој команди.

git config core.editor команде

Пратећа упутства за конфигурацију су у [Ваш едитор](#), многи едитори могу да се поставе на следећи начин:

Табела 4. Испрвна листа core.editor конфигурационих команда

Едитор	Конфигурациона команда
Atom	<code>git config --global core.editor "atom --wait"</code>
BBEdit (Мек, са алатима из команде линије)	<code>git config --global core.editor "bbedit -w"</code>
Emacs	<code>git config --global core.editor emacs</code>
Gedit (Линукс)	<code>git config --global core.editor "gedit --wait --new-window"</code>
Gvim (Виндоуз 64-битни)	<code>git config --global core.editor "'C:\Program Files\Vim\vim72\gvim.exe' --nofork '%'"</code> (Погледајте и напомену испод)
Kate (Линукс)	<code>git config --global core.editor "kate"</code>
nano	<code>git config --global core.editor "nano -w"</code>
Notepad (Виндоуз 64-битни)	<code>git config core.editor notepad</code>
Notepad++ (Виндоуз 64-битни)	<code>git config --global core.editor "'C:\Program Files\Notepad\notepad.exe' -multiInst -notabbar -nosession -noPlugin"</code> (Погледајте и напомену испод)
Scratch (Линукс)	<code>git config --global core.editor "scratch-text-editor"</code>
Sublime Text (мекОС)	<code>git config --global core.editor "/Applications/Sublime Text.app/Contents/SharedSupport/bin/subl --new-window --wait"</code>
Sublime Text (Виндоуз 64-битни)	<code>git config --global core.editor "'C:\Program Files\Sublime Text 3\sublime_text.exe' -w"</code> (Погледајте и напомену испод)
TextEdit (мекОС)	<code>git config --global core.editor "open --wait-apps --new -e"</code>
Textmate	<code>git config --global core.editor "mate -w"</code>
Textpad (Виндоуз 64-битни)	<code>git config --global core.editor "'C:\Program Files\TextPad 5\TextPad.exe' -m</code> (Погледајте и напомену испод)
UltraEdit (Виндоуз 64-битни)	<code>git config --global core.editor Uedit32</code>
Vim	<code>git config --global core.editor "vim --nofork"</code>
Visual Studio Code	<code>git config --global core.editor "code --wait"</code>
VSCodium (Слободни/Libre Open Source Software бинарни фајлови VSCode)	<code>git config --global core.editor "codium --wait"</code>
WordPad	<code>git config --global core.editor '"C:\Program Files\Windows NT\Accessories\wordpad.exe"'</code>
Xi	<code>git config --global core.editor "xi --wait"</code>



Ако имате 32-битни едитор на Виндоуз 64-битном систему, програм ће бити инсталiran у `C:\Program Files (x86)\` а не у `C:\Program Files\` као што пише горњој табели.

git help

Команда `git help` се користи за приказ комплетне документације која се испоручује уз програм Гит о било којој команди. Пошто у овом додатку излажемо површински преглед већине популарнијих команда, потпуни списак свих могућих опција и заставица за сваку команду увек можете видети ако извршите `git help <команда>`.

Команду `git help` смо увршили у `git help` и показали вам како да је користите да добијете више информација о `git shell` у [Подешавање сервера](#).

Набављање и креирање пројектата

Постоје два начина да дођете до Гит репозиторијума. Један је да га копирате из постојећег репозиторијума на мрежи или на неком другом месту, а други је да креирате нови у неком постојећем директоријуму.

git init

Када желите да неки директоријум претворите у нови Гит репозиторијум, тако да можете почети да контролишете његове верзије, једноставно можете да извршите `git init`.

Ово прво уводимо у [Прављење Гит репозиторијума](#), где показујемо креирање потпуно новог репозиторијума са којим починјете да радите.

Укратко причамо о томе како можете да промените подразумевано име гране из „master” у [Удаљене гране](#).

Ову команду користимо у [Постављање оголјеног репозиторијума на сервер](#) за креирање празног оголјеног репозиторијума за сервер.

Конечно, у [Водовод и порцелан](#) пролазимо кроз неке од детаља о ономе шта се заиста догађа у позадини.

git clone

Команда `git clone` је у суштини нешто као омотач око неколико других команда. Она креира нови директоријум, улази у њега и извршава `git init` којом прави празан Гит репозиторијум, додаје удаљени репозиторијум (`git remote add`) са URL адресе коју јој проследите (подразумевано под именом `origin`), извршава `git fetch` са тог удаљеног репозиторијума и онда одјављује најновији комит у ваш радни директоријум командом `git checkout`.

Команда `git clone` се у књизи користи на доста места, али ћемо навести неколико интересантнијих места.

У суштини је уведена и објашњена у [Клонирање постојећег репозиторијума](#), где пролазимо кроз неколико примера.

У [Постављање програма Гит на сервер](#) приказујемо употребу опције `--bare` којом се креира копија Гит репозиторијума без радног директоријума.

У [Паковање](#) је користимо да распакујемо спаковани Гит репозиторијум.

Конечно, у [Клонирање пројекта са подмодулима](#) учимо о `--recursive-submodules` опцији којом је клонирање репозиторијума са подмодулима мало једноставније.

Мада се користи на многим другим местима у књизи, ово су она донекле јединствена или она на којима се њена потреба донекле разликује у односу на остала места.

Основно снимање

За основни процес рада у којем се садржај поставља на стејџ и комитује у историју, постоји само неколико основних команда.

git add

Команда `git add` поставља садржај из радног директоријума на стејџ (или „индекс“) за наредни комит. Када се изврши команда `git commit`, она подразумевано гледа у овај стејџ, тако да се `git add` користи да прецизно обликује оно што желите да да постане део снимка наредног комита.

Ова команда је изузетно важна у програму Гит и помиње се на дosta места у књизи. Брзо ћемо проћи кроз неке јединствене употребе које можете да пронађете.

`git add` најпре уводимо и детаљно објашњавамо у [Праћење нових фајлова](#).

Помињемо како се користи за разрешавање конфликата при стајању у [Основни конфликти при спајању](#).

Показујемо како се користи за интерактивно постављање делова изменjenог фајла на стејџ у [Интерактивно стејџовање](#).

Конечно, емулирамо је на ниском нивоу у [Објекти стабла](#), тако да можете да стекнете идеју шта она обавља у позадини.

git status

Команда `git status` ће вам приказати различита стања фајлова у радном директоријуму и на стејџу. Који фајлови су изменjeni и нису на стејџу, који се налазе на стејџу али још увек нису комитовани. У свом уобичајеном облику, она ће такође да покаже и неке основне савете како да померате фајлове изменју ових етапа.

`status` најпре описујемо у [Провера статуса фајлова](#), и у њеном основном и у поједностављеном облику. Мада је користимо по целој књизи, овде је показано скоро све што можете да урадите командом `git status`.

git diff

Команда `git diff` се користи када желите да видите разлике изменју било која два стабла. То би могла бити разлика изменју радног окружења и стејџа (`git diff` само по себи), изменју стејџа и вашег последњег комита (`git diff --staged`), или изменју два комита (`git diff master`

`branchB`).

Најпре представљамо основне употребе команде `git diff` у [Преглед стејџованих и нестејџованих промена](#), где показујемо како да погледате које измене се налазе на стејџу, а које још увек нису тамо.

Користимо је у [Смернице за комитове](#) да опцијом `--check` пронађемо евентуалне проблеме са празним простором пре него што комитујемо.

У [Како утврдити шта је уведено](#) сазнајемо како да ефективније погледамо разлике између две гране са `git diff A…B` синтаксом.

Користимо је да филтрирамо разлике у празном простору са `-b` и сазнајемо начин како да упоредимо различите етапе фајлова у конфликту са `--theirs`, `--ours` и `--base` у [Напредно спајање](#).

Конечно, користимо је да ефективно упоредимо измене у подмодулу са `--submodule` у [Први кораци са подмодулима](#).

git difftool

Команда `git difftool` једноставно покреће спољни алат за прилаз разлика између два стабла у случају када вам је потребно нешто више од онога што може уградјена `git diff` команда.

Укратко је помињемо у [Преглед стејџованих и нестејџованих промена](#).

git commit

Команда `git commit` узима садржаје свих фајлова који су са `git add` постављени на стејџ и прави трајни снимак у бази података, па затим помера показивач текуће гране навише да показује на њега.

Основе комитовања најпре показујемо у [Комитовање промена](#). Ту такође показујемо како се користи заставица `-a` којом се у свакодневним процесима рада прескаче `git add` корак и како се користи заставица `-m` којом се уместо покретања едитора директно прослеђује комит порука.

У [Опозив](#) објашњавамо опцију `--amend` којом се врши измена најновијег комита.

У [Укратко о гранању](#) улазимо у детаље онога што ради `git commit`, као и разлога зашто то ради баш на тај начин.

Сазнали смо како да криптоографски потпишемо комитове помоћу заставице `-S` у [Потписивање комитова](#).

Конечно, представљамо шта команда `git commit` ради у позадини и како је заиста имплементирана у [Комит објекти](#).

git reset

Команда `git reset` се првенствено користи за поништавање ствари, што можете и да погодите када видите глагол. Она се помера око `HEAD` показивача и необавезно мења `index` или стејџ, а такође необавезно може и да изменити радни директоријум ако употребите `--hard`. Ако се неправилно употреби, ова последња опција може да проузрокује губитак рада, тако да морате добро да је разумете пре него што почнете да је користите.

Најпре ефективно покривамо најједноставнији случај употребе команде `git reset` у [Уклањање фајла са стејџа](#), где је користимо да са стејџа уклонимо фајл над којим смо извршили `git add`.

Затим је у [Демистификовани ресет](#) детаљно представљамо, комплетно поглавље је посвећено објашњењу ове команде.

У [Прекид спајања](#) користимо `git reset --hard` да прекинемо спајање, где такође користимо и `git merge --abort`, што донекле представља омотач око `git reset` команде.

git rm

`git rm` команда се користи за уклањање фајлова са стејџа и из радног директоријума програма Гит. Слична је команди `git add` у томе што на стејџ за наредни комит поставља акцију уклањања фајла.

Команду `git rm` детаљно представљамо у [Уклањање фајлова](#), укључујући рекурзивно уклањање фајлова и уклањање само са стејџа, уз остављање у радном директоријуму са `--cached`.

Једина различита употреба команде `git rm` у књизи је у [Уклањање објекта](#) где укратко употребљавамо и објашњавамо `--ignore-unmatch` када се извршава `git filter-branch`, што једноставно не изазива грешку када фајл који покушавамо да уклонимо не постоји. Ово може бити корисно када се пишу скрипте.

git mv

Команда `git mv` је мала згодна команда за померање фајла па покретање `git add` над новим фајлом и `git rm` над старим фајлом.

Само је укратко помињемо у [Премештање фајлова](#).

git clean

Команда `git clean` се користи за уклањање нежељених фајлова из радног директоријума. То може да укључи уклањање привремених артефакта изградње или фајлова конфликта при спајању.

Доста опција и сценарија у којима би могли да употребите команду за чишћење покривамо у [Чишћење вашег радног директоријума](#).

Гранање и спајање

Постоји само неколико команди које у програму Гит имплементирају већину функционалности гранања и спајања.

git branch

Команда `git branch` је уствари нека врста алата за управљање гранама. Може да вам испише постојеће гране, креира нову грану, обрише гране и промени им име.

Већи део [Гранање у програму Гит](#) је посвећен команди `branch` која се користи кроз цело поглавље. Најпре је уводимо у [Прављење нове гране](#) и кроз већину њених осталих могућности (испис и брисање) пролазимо у [Управљање гранама](#).

У [Гране за праћење](#) користимо опцију `git branch -u` да поставимо грану за праћење.

Конечно, у [Гит референце](#) прелазимо нешто од онога што ова команда обавља у позадини.

git checkout

Команда `git checkout` се користи за промену текуће гране и одјављивање садржаја у радни директоријум.

Први пут се сусрећемо са овом командом у [Мењање грана](#) заједно уз `git branch` команду.

Сазнајемо како да је употребимо да започнемо праћење гране заставицом `--track` у [Гране за праћење](#).

Користимо је да поново уведемо конфликте фајлова са `--conflict=diff3` у [Одјављивање конфликтата](#).

Улазимо у детаље њене везе са `git reset` у [Демистификовани ресет](#).

Конечно, приказујемо неке детаље имплементације у [Референца HEAD](#).

git merge

Алат `git merge` се користи за спајање једне или више грана у грану коју сте одјавили. Затим ће померити унапред текућу грану на резултат спајања.

Команда `git merge` је први пут уведена у [Основе гранања](#). Мада се користи на многим местима у књизи, постоји мало варијација `merge` команде — углавном само `git merge <грана>` уз име једне гране у коју желите да спојите.

Показали смо како да се уради спљескано спајање (када програм Гит спаја рад али се претвара као да је то нови комит и не бележи историју гране коју спајате) на самом крају [Рачвани јавни пројекат](#).

У [Напредно спајање](#) смо дosta говорили о процесу спајања и самој команди, укључујући команду `-Xignore-space-change` и `--abort` заставицу која прекида проблематично спајање.

Научили смо како да проверимо потписе пре спајања у случају да ваш пројекат користи GPG потписивање у [Потписивање комитова](#).

Конечно, научили смо о спајању подстабала у [Спајање подстабла](#).

git mergetool

Команда `git mergetool` просто покреће спољни помоћник за спајање у случају да имате проблема са спајањем у програму Гит.

Укратко је помињемо у [Основни конфликти при спајању](#), а детаљно објашњавамо како да имплементирате сопствени спољни алат за спајање у [Спољни алати за спајање и приказ разлика](#).

git log

Команда `git log` се користи да прикаже забележену историју пројекта до које може да се дође када се из последњег комит снимка крене уназад. Подразумевано ће да прикаже само историју гране на којој се тренутно налазите, али јој можете проследити различиту, или чак и више глава или грана из којих треба да крене у обилазак. Такође се често користи за приказ разлика између две или више грана на нивоу комита.

Ова команда се користи у скоро сваком поглављу књиге да покаже историју пројекта.

Команду уводимо и донекле детаљно представљамо у [Преглед историје комитова](#). Ту објашњавамо опције `-p` и `--stat` како би стекли идеју шта је увео сваки комит, затим опције `--pretty` и `--oneline` за концизнији приказ историје, уз нешто једноставнијих опција за филтрирање по датуму и аутору.

У [Прављење нове гране](#) је користимо уз опцију `--decorate` да се једноставно графички прикаже где се налазе показивачи наше гране, а користимо и опцију `--graph` да видимо како изгледају историје које се разилазе.

У [Мали приватни тим](#) и [Опсези комитова](#) објашњавамо `гранаА..гранаб` синтаксу која се користи уз команду `git log` за приказ комитова који су у грани јединствени релативно у односу на неку другу грану. У [Опсези комитова](#) ово приказујемо прилично детаљно.

У [Лог спајања и Трострука тачка](#) приказујемо `гранаА…гранаб` формат и `--left-right` синтаксу да видимо шта се налази на једној грани или на другој, али не на обе. У [Лог спајања](#) такође приказујемо како да се опција `--merge` користи као помоћ код дебаговања конфликта при спајању, као и употребу опције `--cc` за преглед конфликата комитова спајања у историји.

У [RefLog кратка имена](#) користимо опцију `-g` за преглед Гит `reflog` кроз овај алат, уместо да вршимо обилазак гране.

У [Претрага](#) приказујемо употребу опција `-S` и `-L` за обављање прилично софицицираних претрага нечега што се историјски десило у коду, као што је праћење историје неке функције.

У [Потписивање комитова](#) видимо како се користи `--show-signature` за додавање

валидационог стринга у сваки комит излаза `git log` команде у зависности од тога да ли је комит био исправно потписан или не.

git stash

`git stash` команда се користи за привремено чување рада који није комитован како би се очистио радни директоријум без потребе да се комитује незавршени посао на грани.

Она је у суштини потпуно представљена у [Скривање и чишћење](#).

git tag

Команда `git tag` се користи за постављање сталног маркера на одређено место у историји кода. У општем случају се ово користи за ствари као што су издања.

Ова команда је представљена и детаљно објашњена у [Означавање](#), а у пракси је користимо у [Означавање издања](#).

Такође смо у [Потписивање вашег рада](#) показали како да се креира GPG потписана ознака заставицом `-s` и провера ознаке заставицом `-v`.

Дељење и ажурирање пројекта

Нема много команди у програму Гит које приступају мрежи, скоро све команде оперишу над локалном базом података. Када сте спремни да поделите свој рад или да повучете измене са неког другог места, постоји неколико команди које раде са удаљеним репозиторијумима.

git fetch

Команда `git fetch` комуницира са удаљеним репозиторијумом, преузима све информације из тог репозиторијума које се не налазе у вашем тренутном, па их смешта у локалну базу података.

Са овом командом се први пут сусрећемо у [Добављање и повлачење из удаљених репозиторијума](#), па настављамо са приказом примера њене употребе у [Удаљене гране](#).

Такође је користимо у неколико примера у [Како се даје допринос пројекту](#).

Користимо је да преузмемо једну одређену референцу која се налази ван подразумеваног простора у [Референце на захтев за повлачење](#), а у [Паковање](#) видимо како да преузмемо из запакованог репозиторијума.

Подешавамо детаљно прилагођене рефспекове да `git fetch` уради нешто мало другачије од онога што подразумевано ради у [Рефспек](#).

git pull

Команда `git pull` је у суштини комбинација команди `git fetch` и `git merge`, којом програм Гит преузме са удаљеног репозиторијума који наведете и онда непосредно након тога покуша да изврши спајање у грани на којој се тренутно налазите.

Укратко смо је представили у [Добављање и повлачење из удаљених репозиторијума](#) и показали како да видите шта ће спојити ако је покренете у [Истраживање удаљеног репозиторијума](#).

Такође показујемо како да је употребите као помоћ при тешкоћама ребазирања у[Ребазирање када ребазирате](#).

У [Одјављивање удаљених грана](#) показујемо како да је користите са URL адресом да једнократно повучете измене.

Конечно, у [Потписивање комитова](#) укратко помињемо да можете употребити опцију `--verify-signatures` ове команде да проверите да ли су комитови које повлачите GPG потписани.

git push

Команда `git push` се користи за комуникацију са другим репозиторијумом, израчунавање шта ваш репозиторијум има што удаљени нема, па затим тура разлику у други репозиторијум. Она захтева да имате право уписа у удаљени репозиторијум, па је логично да се аутентификује на неки начин.

Команду `git push` прво представљамо у [Гурање ка удаљеним репозиторијумима](#). Ту обрађујемо основе гурања гране на удаљени репозиторијум. У [Гурање](#) идемо мало детаљније у гурање одређених грана, а у [Гране за праћење](#) показујемо како да поставите гране за праћење на које се аутоматски гура. У [Брисање удаљених грана](#) користимо заставицу `--delete` да обришемо грану на серверу са `git push`.

Кроз [Како се даје допринос пројекту](#) дајемо неколико примера употребе `git push` за дељење рада на гранама кроз више удаљених репозиторијума.

Начин употребе ове команде за дељење ознака које сте направили опцијом `--tags` показујемо у [Дељење ознака](#).

У [Објављивање измена подмодула](#) користимо опцију `--recurse-submodules` да проверимо да ли је сав рад из наших подмодула објављен пре него што гурамо на суперпројекат, што је заиста корисно када се користе подмодули.

У [Остале куке на клијентској страни](#) укратко причамо о `pre-push` куки, која представља скрипту коју постављамо да се изврши пре него што се заврши гурање и која проверава да ли је дозвољено да се обави гурање.

Конечно, у [Гурање рефспекова](#) причамо о гурању са потпуним рефспеком уместо уопштених пречица које се обично користе. Ово вам помаже да будете потпуно одређени у вези рада који желите да поделите.

git remote

Команда `git remote` је алат за управљање вашом колекцијом удаљених репозиторијума. Омогућава вам да дугачке URL адресе сачувате као кратке ручке, као што је „origin” тако да не морате стално да их уносите. Можете да их имате неколико и команда `git remote` се

користи да их за додате, измените и обришете.

Ова команда је детаљно представљена у [Рад са удаљеним репозиторијумима](#), укључујући приказ, додавање, уклањање и измену имена.

Такође се користи се и у скоро сваком наредном поглављу књиге, али увек у стандардном `git remote add <име> <url>` формату.

git archive

Команда `git archive` се користи за креирање фајла архиве одређеног снимка пројекта.

У [Припрема за издање](#) команду `git archive` користимо да направимо *tarball* архиву пројекта коју можемо да делимо.

git submodule

Команда `git submodule` се користи за управљање спољним репозиторијумима унутар обичних репозиторијума. Ово би могло да послужи за библиотеке или остале врсте дељених ресурса. Команда `submodule` има неколико подкоманди (`add`, `update`, `sync`, итд.) којима се управља овим ресурсима.

Ова команда се помиње само у [Подмодули](#) и ту је детаљно обрађена.

Инспекција и поређење

git show

Команда `git show` може да прикаже Гит објекат на једноставан начин лако читљив људима. Ову команду ћете углавном користити да погледате информације о ознаки или комиту.

По први пут је користимо у [Прибележене ознаке](#) да нам прикаже информације о прибележеним ознакама.

Касније је поприлично користимо у [Избор ревизија](#) за приказ комитова које разрешавају разни начини избора ревизија.

Једна од интересантнијих ствари које радимо са `git show` је у [Ручно поновно спајање фајла](#) где издвајамо садржај одређеног фајла из различитих етапа за време конфликта при спајању.

git shortlog

Команда `git shortlog` се користи за резимирање излаза команде `git log`. Она ће прихватити многе опције исте као и команда `git log` или ће уместо да прикаже све комитове, приказати сажетак комитова груписан по ауторима.

У [Кратки лог](#) смо показали како да је употребите да креирате фин лог измена.

git describe

Команда `git describe` се користи да узме све што разрешава у комит и креира стринг који је донекле читљив људима и који се неће изменити. То је начин да се добије опис комита који је једнозначен као и SHA-1 хеш комита, али лакше разумљив.

`git describe` користимо у [Генерисање броја изградње](#) и [Припрема за издање](#) да добијемо стринг којим након тога именујемо фајл издања.

Отклањање грешака

Програм Гит поседује неколико команди које се користе као помоћ при отклањању грешака у коду. То иде од одређивања места на којем је нешто уведено, до одређивања онога ко је то увео.

git bisect

Алат `git bisect` не невероватно користан алат за исправљање грешака који се користи да се пронађе одређени комит који је први увео бак или проблем извршавањем аутоматизоване бинарне претраге.

У потпуности је објашњен у [Бинарна претрага](#) и помиње се само у том одељку.

git blame

Команда `git blame` означава линије било ког фајла информацијом који комит је последњи унео измену у сваку од линија фајла, као и особу која је направила тај комит. Ово помаже да се открије особа која треба да пита за више информација у вези одређеног дела кода.

Ово је објашњено у [Означавање фајла](#) и помиње се само у том одељку.

git grep

Команда `git grep` може да вам помогне да пронађете било који стринг или регуларни израз у било ком од фајлова вашег извornог кода, чак и у старијим верзијама пројекта.

Објашњена је у [Git Grep](#) и помиње се само у том одељку.

Крпљење

Неколико команди програма Гит је фокусирано на концепт да се комитови посматрају у светлу измена које уводе, као да је низ комитова низ закрпа. Те команде вам помажу да гранама управљате на овај начин.

git cherry-pick

Команда `git cherry-pick` се користи да се одабере измена коју је увео један Гит комит и да се покуша њено поновно увођење на грани на којој се тренутно налазите. Ово може бити корисно да се из гране узме само један или два комита појединачно, јер се спајањем гране

узимају све измене.

Одабир измена (*cherry picking*) је описан и показан у [Процеси рада са ребазирањем и одабиром \(*cherry-picking*\)](#).

git rebase

Команда `git rebase` је у основи аутоматизована команда `cherry-pick`. Она одређује низ комитова па их затим одабира један по један и у истом редоследу их применjuје не неком другом месту.

Ребазирање је детаљно обрађено у [Ребазирање](#), укључујући и расправу о проблемима у сарадњи када се примени ребазирање грана које су већ јавне.

Ми је у [Замена](#) користимо у вежби током примера поделе историје у два одвојена репозиторијума, користећи и заставицу `--onto`.

У [Rerere](#) показујемо наилазак на конфликт при спајању током ребазирања.

Такође је користимо у [Измена више комит порука одједном](#) за режим интерактивног скриптинга са опцијом `-i`.

git revert

Команда `git revert` је у суштини обрнута команда `git cherry-pick`. Она креира нови комит који примењује тачно супротно од измене уведене комитом који циљате, што га ефективно поништава.

Користимо је [Враћање комита](#) за поништавање комита спајања.

Имејл

Многи Гит пројекти, укључујући и сам Гит, се у потпуности одржавају путем мејлинг листи. У програм Git је уgraђен већи број алата који олакшавају тај процес, од генерисања закрпа које лако можете да пошаљете имејлом до примене тих закрпа из имејл сандучета.

git apply

Команда `git apply` примењује закрпу коју је креирала команда `git diff` или чак *GNU diff* команда. То слично ономе што би могла да уради команда `patch` уз неколико мањих разлика.

У [Примењивање закрпа из имејлова](#) приказујемо како се користи као и околности у којима би могли да је употребите.

git am

Команда `git am` се користи да примени закрпе из долазног имејл сандучета, тачније оног форматираног као *mbox*. Ово је корисно да се путем имејла приме закрпе и да се једноставно примене на пројекат.

Употребу и процес рада око `git am` команде смо показали у [Примењивање закрпе са am](#) укључујући и употребу опција `--resolved`, `-i` и `-3`.

Такође постоји и већи број кука које можете користити као помоћ у процесу рада везаног за команду `git am` и све оне су приказане у [Куке у вези процеса рада са имејловима](#).

Такође је у [Имејл обавештења](#) користимо да се примени закрпа са изменама форматираним као *GitHub* захтев за повлачење.

git format-patch

Команда `git format-patch` се користи да генерише низ закрпа у *mbox* формату које у исправном облику можете да пошаљете на мејлинг листу.

У [Јавни пројекат преко имејла](#) прелазимо пример давања доприноса пројекту коришћењем `git format-patch` алата.

git imap-send

Команда `git imap-send` шаље серверу поштанско сандуче које је генерисала команда `git format-patch` у IMAP директоријум незавршених порука.

У [Јавни пројекат преко имејла](#) пролазимо кроз пример давања доприноса пројекту слањем закрпа алатом `git imap-send`.

git send-email

Команда `git send-email` се користи за слање закрпа које је генерисала команда `git format-patch` имејлом.

У [Јавни пројекат преко имејла](#) пролазимо кроз пример давања доприноса пројекту слањем закрпа алатом `git send-email`.

git request-pull

Команда `git request-pull` се једноставно користи да генерише пример тела имејл поруке коју треба некоме да пошаљете. Ако имате грану на јавном серверу и желите да неко зна како да интегрише те измене без потребе да закрпе шаљете имејлом, можете да извршите ову команду и пошаљете њен излаз особи која треба да повуче измене.

Начин употребе команде `git request-pull` за генерисање поруке повлачења описујемо у [Рачвани јавни пројекат](#).

Спољни системи

Програм Гит долази са неколико команди којима се интегрише са осталим системима за контролу верзије.

git svn

Команда `git svn` се користи за комуникацију са *Subversion* системом за контролу верзије као клијент. То значи да програм Гит можете употребити да одјавите са и да комитујете на *Subversion* сервер.

Ова команда је детаљно приказана у [Гит и Subversion](#).

git fast-import

За остале системе контроле верзије или за увоз из скоро било ког формата, можете да употребите команду `git fast-import` која брзо мапира други формат у нешто што програм Гит лако може да забележи.

Ова команда је детаљно приказана у [Прилагодљиви увозник](#).

Администрација

Ако администрирате Гит репозиторијум или ако морате да поправите нешто крупно, програм Git вам обезбеђује већи број административних команди које вам помажу у томе.

git gc

Команда `git gc` обавља „скупљање ћубрета” по вашем репозиторијуму, уклањајући непотребне фајлове из базе података и пакујући преостале фајлове у ефикаснији формат.

Ова команда се обично извршава у позадини, мада можете и ручно да је покренете ако то желите. У [Одржавање](#) приказујемо неколико примера.

git fsck

Команда `git fsck` се користи за проверу конзистентности интерне базе података и проблема су вез ње.

Користили смо је само једном у [Опоравак података](#) да пронађемо висеће објекте.

git reflog

Команда `git reflog` пролази кроз лог свих места на које су указивале главе ваших грана док сте радили да би пронашла комитове које сте можда изгубили кроз поновно исписивање историја.

Ову команду углавном приказујемо у [RefLog кратка имена](#), где представљамо уобичајену употребу, као и како да употребите `git log -g` да исте информације погледате у излазу команде `git log`.

У [Опоравак података](#) такође пролазимо кроз практични пример опоравка једне такве изгубљене гране.

git filter-branch

Команда `git filter-branch` се користи за поновно исписивање гомиле комитова према одређеним шаблонима, као што је уклањање фајла са свих места или филтрирање комплетног репозиторијума на један једини поддиректоријум у циљу издвајања пројекта.

У [Уклањање фајла из сваког комита](#) објашњавамо команду и истражујемо неколико различитих опција као што су `--commit-filter`, `--subdirectory-filter` и `--tree-filter`.

У [Git-p4](#) је користимо да поправимо уvezене спољне репозиторијуме.

Водоводне команде

Такође постоји већи број водоводних команда низког нивоа које срећемо кроз књигу.

Прва на коју наилазимо је [ls-remote](#) у [Референце на захтев за повлачење](#) коју користимо за преглед сирових референци на серверу.

[ls-files](#) користимо у [Ручно поновно спајање фајла](#), [Rerere](#) и [Индекс](#) да прикажемо сирови поглед на стање стејџа.

Такође помињемо [rev-parse](#) у [Референце грана](#) да узме скоро било који стринг и претвори га у SHA-1 објекта.

Међутим, већину водоводних команда ниског нивоа представљамо у [Гит изнутра](#), што је мање-више оно на шта се фокусира то поглавље. Покушали смо да их не употребимо у већем делу остатка књиге.

Индекс

- @
- \$EDITOR, 358
- \$VISUAL
 - погледајте \$EDITOR, 358
- .NET, 512
- .gitignore, 360
- @{upstream}, 98
- @{u}, 98
- Базаар, 437
- Виндоуз
 - инсталирање, 19
- ГКИ, 494
- Гит као клијент, 392
- Графички алати, 494
- Интероперабилност са осталим VCS системима
 - Subversion, 392
 - Меркуријал, 403
 - Перфорс, 414
- Линукс, 13
 - инсталирање, 18
- Линус Торвалдс, 13
- Меркуријал, 403, 434
- Миграирање на Гит, 431
- Перфорс, 414, 440
 - Git Fusion, 415
- Увоз
 - из Subversion, 431
 - из Меркуријала, 434
 - из Перфорс, 440
 - из осталих, 442
 - из програма Базаар, 437
- акредитиви, 351
- алијаси, 63
- архивирање, 375
- атрибути, 368
- бинарни фајлови, 369
- бројеви изградње, 174
- гит команде
 - add, 29, 30, 30
 - am, 162
 - apply, 161
 - archive, 174
 - branch, 68, 82
 - checkout, 69
 - cherry-pick, 171
 - clone, 27
 - bare, 116
 - commit, 37, 66
 - config, 21, 23, 37, 64, 159, 357
 - credential, 351
 - daemon, 123
 - describe, 174
 - diff, 34
 - check, 137
 - fast-import, 442
 - fetch, 54
 - fetch-pack, 478
 - filter-branch, 441
 - format-patch, 158
 - gitk, 494
 - gui, 494
 - help, 24, 122
 - init, 27, 30
 - bare, 117, 120
 - instaweb, 126
 - log, 41
 - merge, 76
 - squash, 157
 - mergetool, 81
 - p4, 423, 440
 - pull, 55
 - push, 55, 61, 95
 - rebase, 100
 - receive-pack, 476
 - remote, 52, 54, 55, 57
 - request-pull, 154
 - rerere, 172
 - send-pack, 476
 - shortlog, 175
 - show, 60
 - show-ref, 395
 - status, 29, 36
 - svn, 392
 - tag, 57, 59, 60
 - upload-pack, 478
 - „http-backend”, 124
- ране, 66
 - брисање удаљене, 99
 - дифовање, 165

дуготрајне, 86
криирање, 68
основни ток рада, 73
праћење, 97
прелазак, 69
спајање, 78
тематске, 87, 161
удаљене, 89, 164
узводне, 97
управљање, 82
турање, 95
дистрибуирани гит, 133
доприношење, 136
велики јавни пројекат, 157
мали приватни тим, 139
мали јавни пројекат, 153
приватни тим са руководством, 147
едитор
измена подразумеваног, 37
игнорисање фајлова, 32
издавање, 174
имејл, 159
примењивање закрпа из, 161
интегрисање рада, 167
комит шаблони, 358
контрола верзије, 9
дистрибуирана, 11
локална, 9
централизована, 10
крајеви линија, 366
куке, 378
post-update, 113
мекОС
инсталирање, 18
одзиви љуске
PowerShell, 505
bash, 501
zsh, 502
одржавање пројекта, 160
ознаке, 57, 173
лаке, 59
потписивање, 173
прибележене, 59
повлачење, 98
празан простор, 365
пример полисе, 381
протокол
локални, 110
протоколи
SSH, 114
гит, 115
паметни HTTP, 112
приглуп HTTP, 112
процеси рада, 133
диктатор и поручници, 135
менаџер интеграције, 134
ребазирање и одабир, 171
спајање, 167
централизован, 133
„спајање (велико)”, 169
проширење кључних речи, 372
рачвање, 135, 181
ребазирање, 99
опасности, 104
против спајања, 108
референце
удаљени, 89
сервирање репозиторијума, 110
GitLab, 127
GitWeb, 125
HTTP, 124
SSH, 117
гит протокол, 122
спајање, 78
конфликти, 79
против ребазирања, 108
стратегије, 377
стејџ
прескакање, 38
таб довршавање
PowerShell, 505
bash, 501
zsh, 502
фајлови
премештање, 40
уклањање, 38
фильтрирање лога, 47
форматирање лога, 44

A
Apache, 124
Apple, 513
autocorrect, 360

B
BitKeeper, 13
bash, 501

C
C, 508
C#, 512
CVS, 10
Cocoa, 513
color, 361
crlf, 366

D
Dulwich, 519
difftool, 362

E
excludes, 360, 452

G
GPG, 359
GitHub, 176
 API, 224
 захтеви за повлачење, 185
 кориснички налози, 176
 организације, 216
 процес, 182
GitHub за Виндоуз, 496
GitHub за мекОС, 496
GitLab, 127
GitWeb, 125
Go, 517
git-svn, 392
gitk, 494
go-git, 517

I
IRC, 24

J
Java, 513
jgit, 513

L
libgit2, 508

M
Mono, 512
master, 67
mergetool, 362

O
Objective-C, 513
origin, 90

P
Perforce, 10, 13
PowerShell, 505
Python, 513, 519
pager, 359
posh-git, 505

R
Ruby, 509
rerere, 172

S
SHA-1, 15
SSH кључеви, 118
 са GitHub, 177
Subversion, 10, 13, 134, 392, 431

V
Visual Studio, 499

X
Xcode, 18

Z
zsh, 502