Machine Translated by Google

THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Machine Translated by Google

Para Git

Scott Chacon, Ben Straub

Versão 2.1.359-2-g27002dd, 2022-10-03

Machine Translated by Google

Índice

Licença 1

Prefácio de Scott Chacon 2

Prefácio de Ben Straub 3

Dedicatórias. 4

Contribuintes. 5

Introdução. 8

Introdução 10

Sobre o Controle de Versão. 10

Uma Breve História do Git 14

O que é Git? 14

A Linha de Comando 18

Instalando Git 18

Primeira configuração do Git. 21

Conseguindo ajuda. 24

Resumo 25

Noções básicas do Git. 26

Obtendo um repositório Git 26

Registrando Alterações no Repositório . 28

Visualizando o histórico de commits 40

Desfazendo Coisas. 46

Trabalhando com Controles Remotos 50

Marcando 55

Git Aliases 60

Resumo 62

Ramificação do Git. 63

Ramos em poucas palavras 63

Ramificação e fusão básicas. 70

Gestão de Filiais. 79

Fluxos de trabalho de ramificação 82

Ramos remotos 85

Rebaseando 95

Resumo 104

Git no servidor. 105

Os Protocolos 105

Obtendo o Git em um servidor . 110

Gerando sua chave pública SSH 112

Configurando o Servidor . . 113

Git Daemon 116

Machine Translated by Google

HTTP inteligente 117
GitWeb 119
GitLab 120
Opções hospedadas por terceiros. 124
Resumo 125
Git distribuído. 126
Fluxos de trabalho distribuídos 126
Contribuindo para um Projeto 129
Manutenção de um Projeto 150
Resumo 165
GitHub. 166
Configuração e configuração da conta 166
Contribuindo para um Projeto 171
Manutenção de um Projeto 191
Gerenciando uma organização. . . . . 205 Script do
GitHub 208
Resumo 217
Git Tools 218
Seleção de revisão 218
Encenação Interativa. 226
Escorvamento e Limpeza 230
Assinando seu trabalho. 236
Pesquisando 239
Reescrevendo a História. 243
Redefinir Desmistificado 251
Fusão avançada. 270
Vôo 287
Depurando com Git 293
Submódulos. 296
Agrupamento. 317
Substituir 320
Armazenamento de credenciais 329
Resumo 334
Personalizando Git. 335
Git Configuração 335
Atributos Git. 345
Git Ganchos 353
Um exemplo de política aplicada pelo Git 356
Resumo 365
Git e outros sistemas. 366
Git como cliente 366

Machine Translated by Google

Migrando para o Git 401
Resumo 417
Git Internos. 418
Canalizações e Porcelanas. 418
Objetos Git 419
Referências Git. 429
Packfiles 433
O Refspec 436
Protocolos de transferência 439
Manutenção e Recuperação de Dados 444
Variáveis de ambiente 451
Resumo 456
Apêndice A: Git em outros ambientes. 457
Interfaces Gráficas 457
Git no Visual Studio 462
Git no Visual Studio Code 463
Git em IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine. 463
Git em Sublime Text 464
Gite no Bash 464
git em zsh 465
Git no PowerShell 467
Resumo 469
Apêndice B: Incorporando o Git em seus aplicativos 470
Git de linha de comando 470
Libgit2. 470
JGit. . . . . vai- 475
git 478
Dulwich 480
Apêndice C: Comandos do Git 482
Configuração e configuração 482
Obtendo e Criando Projetos 484
Instantâneos Básicos. 485
Ramificação e fusão 487
Compartilhamento e Atualização de Projetos . . . . . 489 Inspeção e
Comparação 491
Depurando 492
Remendando 493
E-mail. 493
Sistemas Externos 495
Administração 495
Comandos de encanamento 496

Machine Translated by Google

Índice 497

Machine Translated by Google

Licença

Este trabalho está licenciado sob Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Licença não portada. Para ver uma cópia desta licença, visite https://creativecommons.org/licenses/by-nc sa/3.0 ou envie uma carta para Creative Commons, PO Box 1866, Mountain View, CA 94042, EUA.

Machine Translated by Google

Prefácio de Scott Chacon

Bem-vindo à segunda edição do Pro Git. A primeira edição foi publicada há mais de quatro anos.
Desde então, muita coisa mudou e, no entanto, muitas coisas importantes não mudaram. Embora a maioria dos comandos e conceitos principais ainda sejam válidos hoje, já que a equipe principal do Git é fantástica em manter as coisas compatíveis com versões anteriores, houve algumas adições e mudanças significativas na comunidade em torno do Git. A segunda edição deste livro destina-se a abordar essas mudanças e atualizar o livro para que seja mais útil para o novo usuário.

Quando escrevi a primeira edição, o Git ainda era uma ferramenta relativamente difícil de usar e mal adotada pelo hacker mais exigente. Estava começando a ganhar força em certas comunidades, mas não havia chegado nem perto da onipresença que tem hoje. Desde então, quase todas as comunidades de código aberto o adotaram. O Git fez um progresso incrível no Windows, na explosão de interfaces gráficas de usuário para todas as plataformas, no suporte IDE e no uso comercial. O Pro Git de quatro anos atrás não sabe nada disso. Um dos principais objetivos desta nova edição é abordar todas essas novidades

fronteiras na comunidade Git.

A comunidade Open Source usando Git também explodiu. Quando me sentei originalmente para escrever o livro há quase cinco anos (demorei um pouco para lançar a primeira versão), eu tinha acabado de começar a trabalhar em uma empresa muito pouco conhecida desenvolvendo um site de hospedagem Git chamado GitHub. No momento da publicação, havia talvez alguns milhares de pessoas usando o site e apenas quatro de nós trabalhando nele.
Enquanto escrevo esta introdução, o GitHub está anunciando nosso 10 milhões de projetos hospedados, com quase 5 milhões de contas de desenvolvedores registrados e mais de 230 funcionários. Ame ou odeie, o GitHub mudou fortemente grandes áreas da comunidade Open Source de uma forma que era quase inconcebível quando me sentei para escrever a primeira edição.

Escrevi uma pequena seção na versão original do Pro Git sobre o GitHub como um exemplo de Git hospedado com o qual nunca me senti muito confortável. Não gostei muito de estar escrevendo o que considero essencialmente um recurso da comunidade e também falando sobre minha empresa nele. Embora eu ainda não goste desse conflito de interesses, a importância do GitHub na comunidade Git é inevitável. Em vez de um exemplo de hospedagem do Git, decidi transformar essa parte do livro em uma descrição mais profunda do que é o GitHub e como usá-lo com eficiência. Se você vai aprender a usar o Git, saber como usar o GitHub o ajudará a fazer parte de uma grande comunidade, que é valiosa, não importa qual host Git você decida usar para seu próprio código.

A outra grande mudança desde a última publicação foi o desenvolvimento e ascensão do protocolo HTTP para transações de rede Git. A maioria dos exemplos no livro foi alterada para HTTP de SSH porque é muito mais simples.

Tem sido incrível ver o Git crescer nos últimos anos de um sistema de controle de versão relativamente obscuro para basicamente dominar o controle de versão comercial e de código aberto. Estou feliz que o Pro Git tenha se saído tão bem e também tenha conseguido ser um dos poucos livros técnicos do mercado

que é bastante bem-sucedido e totalmente open source.

Espero que você goste desta edição atualizada do Pro Git.

Machine Translated by Google

Prefácio de Ben Straub

A primeira edição deste livro foi o que me deixou viciado no Git. Esta foi minha introdução a um estilo de criação de software que parecia mais natural do que qualquer coisa que eu já tivesse visto antes. Eu já era desenvolvedor há vários anos, mas essa foi a curva certa que me levou a um caminho muito mais interessante do que aquele em que eu estava.

Agora, anos depois, sou colaborador de uma grande implementação do Git, trabalhei para a maior empresa de hospedagem do Git e viajei pelo mundo ensinando às pessoas sobre o Git. Quando Scott perguntou se eu estaria interessado em trabalhar na segunda edição, nem precisei pensar.

Foi um grande prazer e privilégio trabalhar neste livro. Espero que te ajude tanto quanto ajudou meu.

Machine Translated by Google

dedicatórias

À minha esposa, Becky, sem a qual esta aventura nunca teria começado. — Ben

Esta edição é dedicada às minhas meninas. À minha esposa Jessica, que me apoiou durante todos esses anos, e à minha filha Josephine, que me apoiará quando eu estiver velho demais para saber o que está acontecendo. — Scott

Machine Translated by Google

Contribuintes

Como este é um livro de código aberto, recebemos várias errata e alterações de conteúdo doadas ao longo dos anos. Aqui estão todas as pessoas que contribuíram para a versão em inglês do Pro Git como um projeto de código aberto. Obrigado a todos por ajudarem a tornar este livro melhor para todos.

Contribuintes a partir de 27002dd:

4 semanas
Adam Laflamme
Adriano Ollier
Akrom K
Alan D. Salewski Alba
Mendez Aleh
Suprunovich Alex
Povel
Alexander Bezzubov
Alexandre Garnier
Alfred Myers
Amanda Dillon
Andreas Bjørnestad
Andrei Dascalu
Andrew Layman
Andrew Mac Fie
André Metcalf
Andrew Murphy
Andy Gee
Anne, a Ágil
Anthony Loiseau
Anton Trunov
Antonello Piemonte
Antonino Ingargiola
Ardavast Dayleryan
Artem Leschev
Atul Varma
Bagas Sanjaya
Ben Sima
Benjamim Dopplinger
Billy Griffin
Bob Kline
Bohdan Pylypenko
Borek Bernard
Brett Cannon
Lábio
C Nguyen
Cadel Watson
Carlos Martins Nieto
Carlos Tafur
Chaitanya Gurrapu
Jon Freed
jonathan
Jordan Hayashi
Joris Valette
Josh Byster
Joshua Webb
Junjie Yuan
Justin Clift
Jörn Auerbach
Kartic Sivaraam
KatDwo
Katrin Leinweber
Kausar Mehmood
Keith Hill
Kenneth Kin Lum
Kenneth Lum
Klaus Frank
Kristijan "Fremen" Velkovski
Christopher Szumny
Kyrylo Yatsenko
Lars Vogel
Laxman
Lazar95
Leonard Laszlo
Linus Heckemann
Logan Hasson
Louise Corrigan
Luc Morin
Lukas Rollin
Marat Radchenko
Marcin Sÿdÿak-Jakubowski
Marie-Helene Burle
Marius Žilenas
Markus KARG
Martín Bolívar
Mashrur Mia (Saad)
Masood Fallahpoor
Mathieu Dubreuilh
Matt Cooper
Matt Trzcinski
Mateus Mineiro
Sebastian Krause
Serguei Kuznetsov
Severino Lorilla Jr.
Shengbin Meng
Xerez Hietala
Shi Yan
Siarhei Bobryk
Siarhei Krukau
Skyper
Smaug123
Snehal Shekatkar
Solt Budavári
Song Li
stephan vanmaris
Steven Roddis
Stuart P. Bentley
Sudarsan GP Suhaib
Mujahid Susan Stevens
Sven Selberg
Thanix
Thomas Ackermann
Thomas Hartmann
Tiffany
Tom Schady
Tomas orgulhoso
Tomoki Aonuma
tvvirus
Tyler Cipriani
Fora Yzr
UgmaDevelopment
Vadim Markovtsev
Vangelis Katsikaros
Semana de peso
Victor Ma
Vipul Kumar
Vitaly Kuznetsov
Wesley Gonçalves
William Gathoye
William Turrell
Wlodek Bzyl

Machine Translated by Google

Parque Changwoo
Christian Decker
Christoph Bachhuber
Christopher Prokop
Christopher Wilson
CodificaçãoSpiderFox
Cory Donnelly
Cullen Rhodes
Cirilo
Damien Tournoud
E Schmidt
Daniel Hollas
Daniel Knittl-Frank
Daniel Shahaf
Daniel Sturm
Daniel Tricoli
Daniel Larionov
Danny Lin
David Rogers
David Angelocola
Denis Savitsky
dexter
Dexter Morganov
Diamante X
Dieter Ziller
Dino Karic
Dmitry Tikhonov
Dmitry Smirnov
Doug Richardson
Duncan Dean
Dustin Frank
Ed Flanagan
Eden Hochbaum
Eric Henziger
Exploração
Ezra Buehler
Fabien-jrt
Félix Nehrke
Filip Kucharczyk
Fornost461
Frank
Frederico Mazzone
Frej Drejhammar
Guthrie McAfee Armstrong
Hairy Fotr Hamidreza
Mahdavipanah Haruo Nakayama
Helmut KC Tessarek

Ocultar deVries
HonkingGoose
Howard

Matthieu Moy
Mavaddat Javid
Max Coplan
Michael MacAskill
Michael Sheaver
Michael Welch
Michael van der Wulp
Mike Charles
Mike Pennisi
Mike Thibodeau
Mikhail Menshikov
Mitsuru Kariya
Quadros Máximos
Niels Widger
Niko Stotz
Nils Reusse
Noelle Leigh
Oliver Sieweke
Olleg Samoylov
Osman Khwaja
Owen
Pablo Schläpfer
Pascal Berger
Pascal Borreli
Patrice Cracóvia
Patrick Steinhardt
Pavel Janík
Pawel Krupinski
Pessimista
Peter Kokot
Peter Bodnar
Petr Janecek
Petr Kajzar
Phil Mitchell
Philippe Blain
Philippe Miossec
Pratik Nadagouda
Rafi Raphael R Ray
Chen Rex Kerr Reza
Ahmadi Richard
Hoyle Ricky Senft
Rintze M. Zelle Rob
Blanco Robert P.
Goldman Robert PJ
Day Robert Theis
Rohan D'Souza
Roman Kosenko

Xavier Bonaventura
Y.E.
Yann Soubeyrand
Seu nome
Yue Lin Ho
Yunhai Luo
Yusuke SATO
ajax333221
alex-koziell
Allen joslin
Andreas
applecuckoo
atalakam
axmbo
bripmccann
brotherben
delta4d
devwebcl
dualsky
evanderiel
eyherabh
flip111
flyingzumwalt
franjozen
goekboet grgbnc
haripetrov i-give-
up iprok jingsam
jliljekrantz
johnhar leerg
maks mmikeww
mosdalsvsocld

nicktime
noureddin
patrick96
paveljanik
pedrorijo91
peterwwillis
petuter rahrah

rmzelle
Root
Sanders@oreilly.com
Sharpiro Slavos1
Spacewander TD2014

Machine Translated by Google

Inácio
Gato Ilker
Jan Groenewald
Jaswinder Singh
Jean-Noel Ávila
Jeroen Oortwijn
Jim Hill
Joel Davies
Johannes Devender
Johannes Schindelin
John Lin
Jon Forrest
Ronald Wampler
Rory
Rudiger Herrmann
SATO Yusuke
O próprio Ford
o próprio José
Sanders Kleinfeld
Sarah Schneider
Sourav Sachidanand
Scott Bronson
Sean Head
Sean Jacobs
twekberg
uerdogan
pomar e
1versal xJom

extra
yakirwin
zwPapEr
ÿÿÿÿÿÿÿ
Dilú

Machine Translated by Google

Introdução

Você está prestes a passar várias horas de sua vida lendo sobre o Git. Vamos dedicar um minuto para explicar o que temos reservado para você. Aqui está um rápido resumo dos dez capítulos e três apêndices deste livro.

No **Capítulo** 1, abordaremos os sistemas de controle de versão (VCSs) e o básico do Git — sem detalhes técnicos, apenas o que é o Git, por que surgiu em uma terra cheia de VCSs, o que o diferencia e por que tantas pessoas estão usando. Em seguida, explicaremos como baixar o Git e configurá-lo pela primeira vez, caso ainda não o tenha em seu sistema.

No **Capítulo** 2, veremos o uso básico do Git — como usar o Git nos 80% dos casos que você encontrará com mais frequência. Depois de ler este capítulo, você será capaz de clonar um repositório, ver o que aconteceu no histórico do projeto, modificar arquivos e contribuir com alterações. Se o livro entrar em combustão espontânea neste ponto, você já deve ser bastante útil empunhando o Git no tempo que leva para pegar outra cópia.

O Capítulo 3 é sobre o modelo de ramificação no Git, geralmente descrito como o recurso matador do Git. Aqui você aprenderá o que realmente diferencia o Git do pacote. Quando terminar, você pode sentir a necessidade de passar um momento tranquilo refletindo sobre como você vivia antes da ramificação do Git fazer parte de sua vida.

O capítulo 4 abordará o Git no servidor. Este capítulo é para aqueles que desejam configurar o Git dentro de sua organização ou em seu próprio servidor pessoal para colaboração. Também exploraremos várias opções hospedadas, se você preferir deixar que outra pessoa cuide disso para você.

O Capítulo 5 abordará em detalhes vários fluxos de trabalho distribuídos e como realizá-los com o Git. Ao terminar este capítulo, você deverá ser capaz de trabalhar habilmente com vários repositórios remotos, usar o Git por e-mail e manipular habilmente várias ramificações remotas e patches contribuídos.

O Capítulo 6 aborda o serviço de hospedagem GitHub e as ferramentas em profundidade. Cobrimos a inscrição e o gerenciamento de uma conta, a criação e o uso de repositórios Git, fluxos de trabalho comuns para contribuir com projetos e aceitar contribuições para o seu, a interface programática do GitHub e muitas pequenas dicas para facilitar sua vida em geral.

O Capítulo 7 é sobre comandos Git avançados. Aqui você aprenderá sobre tópicos como dominar o assustador comando 'reset', usar a pesquisa binária para identificar bugs, editar o histórico, selecionar a revisão em detalhes e muito mais. Este capítulo irá complementar seu conhecimento do Git para que você seja realmente um mestre.

O Capítulo 8 é sobre como configurar seu ambiente Git personalizado. Isso inclui a configuração de scripts de gancho para impor ou incentivar políticas personalizadas e o uso de definições de configuração do ambiente para que você possa trabalhar da maneira que desejar. Também abordaremos a criação de seu próprio conjunto de scripts para impor uma política de confirmação personalizada.

O Capítulo 9 trata do Git e outros VCSs. Isso inclui usar Git em um mundo Subversion (SVN) e converter projetos de outros VCSs para Git. Muitas organizações ainda usam SVN e não estão prestes a mudar, mas a essa altura você já terá aprendido o incrível poder do Git — e este capítulo mostra como lidar se você ainda tiver que usar um servidor SVN. Também abordamos como importar projetos de

Machine Translated by Google

vários sistemas diferentes, caso você convença todos a mergulhar.

O Capítulo 10 investiga as profundezas obscuras e belas dos internos do Git. Agora que você sabe tudo sobre o Git e pode manipulá-lo com poder e graça, pode discutir como o Git armazena seus objetos, qual é o modelo de objeto, detalhes de packfiles, protocolos de servidor e muito mais. Ao longo do livro, faremos referência a seções deste capítulo, caso você queira mergulhar fundo nesse ponto; mas se você for como nós e quiser se aprofundar nos detalhes técnicos, talvez queira ler primeiro o Capítulo 10.
Nós deixamos isso para você.

No **Apêndice** A, examinamos vários exemplos de uso do Git em vários ambientes específicos. Cobrimos várias GUIs e ambientes de programação IDE diferentes nos quais você pode querer usar o Git e o que está disponível para você. Se você estiver interessado em uma visão geral do uso do Git em seu shell, IDE ou editor de texto, dê uma olhada aqui.

No **Apêndice** B, exploramos a criação de scripts e a extensão do Git por meio de ferramentas como libgit2 e JGit. Se você estiver interessado em escrever ferramentas personalizadas complexas e rápidas e precisar de acesso Git de baixo nível, é aqui que você pode ver como é essa paisagem.

Por fim, no **Apêndice** C, examinamos todos os principais comandos do Git, um de cada vez, e revisamos onde no livro os abordamos e o que fizemos com eles. Se você quiser saber onde no livro usamos qualquer comando específico do Git, você pode procurar aqui.

Vamos começar.

Machine Translated by Google

Começando

Este capítulo será sobre como começar com o Git. Começaremos explicando algumas informações básicas sobre ferramentas de controle de versão, depois passaremos para como fazer o Git rodar em seu sistema e, finalmente, como configurá-lo para começar a trabalhar. No final deste capítulo, você deve entender por que o Git existe, por que você deve usá-lo e deve estar preparado para isso.

Sobre o Controle de Versão

O que é “controle de versão” e por que você deveria se importar? O controle de versão é um sistema que registra alterações em um arquivo ou conjunto de arquivos ao longo do tempo para que você possa recuperar versões específicas posteriormente. Para os exemplos deste livro, você usará o código-fonte do software como os arquivos cuja versão está sendo controlada, embora, na realidade, você possa fazer isso com praticamente qualquer tipo de arquivo em um computador.

Se você é um designer gráfico ou web e deseja manter todas as versões de uma imagem ou layout (o que você certamente deseja), um Sistema de Controle de Versão (VCS) é uma coisa muito sábia para usar. Ele permite que você reverta os arquivos selecionados para um estado anterior, reverta todo o projeto para um estado anterior, compare as alterações ao longo do tempo, veja quem modificou pela última vez algo que pode estar causando um problema, quem introduziu um problema e quando e muito mais. Usar um VCS geralmente também significa que, se você estragar tudo ou perder arquivos, poderá recuperá-los facilmente. Além disso, você obtém tudo isso com muito pouca sobrecarga.

Sistemas de Controle de Versão Local

O método de controle de versão escolhido por muitas pessoas é copiar os arquivos para outro diretório (talvez um diretório com carimbo de data/hora, se forem espertos). Essa abordagem é muito comum porque é muito simples, mas também é incrivelmente propensa a erros. É fácil esquecer em qual diretório você está e acidentalmente gravar no arquivo errado ou copiar arquivos que não pretendia.

Para lidar com esse problema, há muito tempo os programadores desenvolveram VCSs locais que tinham um banco de dados simples que mantinha todas as alterações nos arquivos sob controle de revisão.

Machine Translated by Google

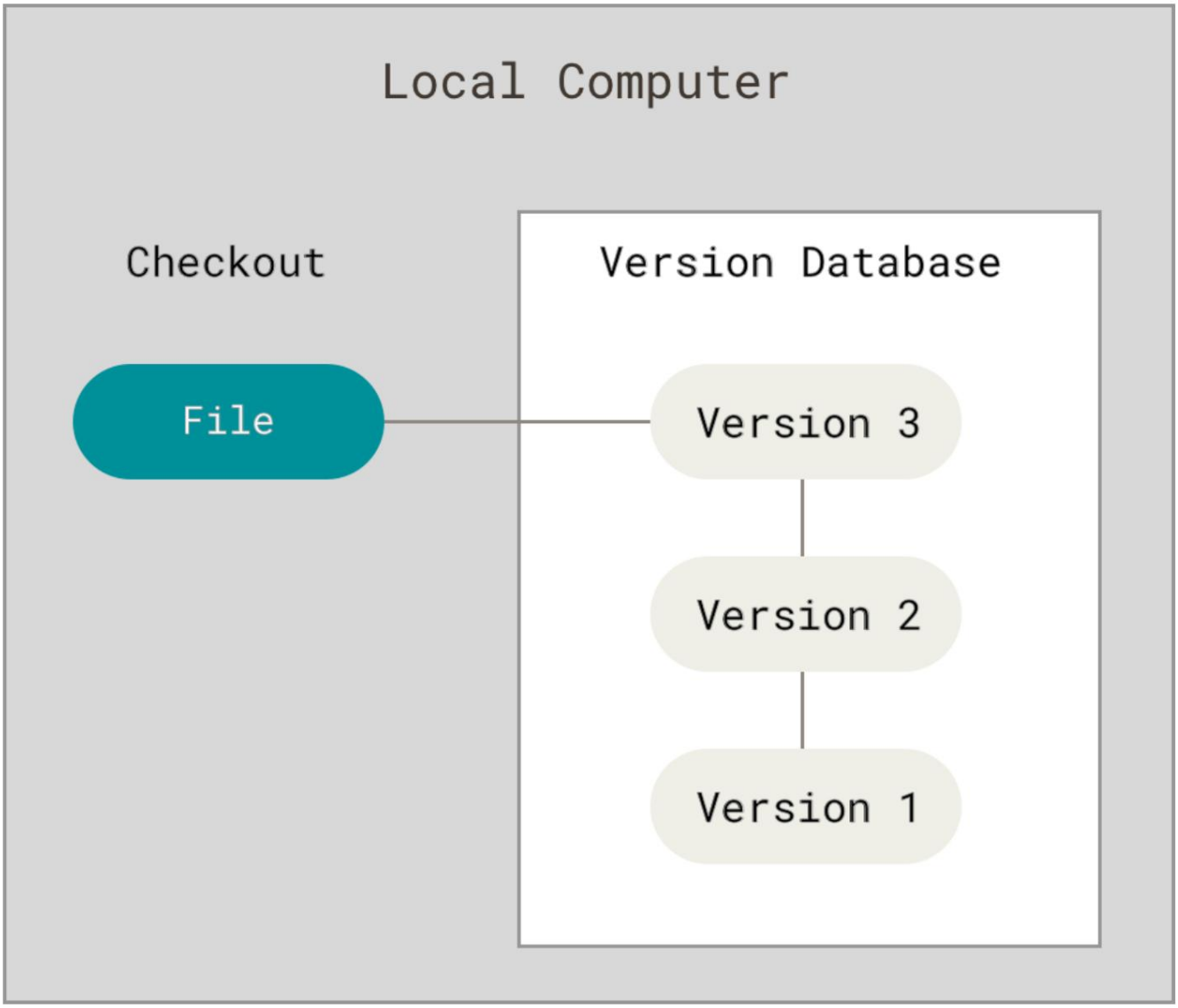


Figura 1. Controle de versão local

Uma das ferramentas VCS mais populares era um sistema chamado RCS, que ainda hoje é distribuído em muitos computadores. RCS funciona mantendo os conjuntos de patches (ou seja, as diferenças entre os arquivos) em um formato especial no disco; ele pode recriar a aparência de qualquer arquivo a qualquer momento, adicionando todos os patches.

Sistemas de Controle de Versão Centralizado

O próximo grande problema que as pessoas encontram é que precisam colaborar com desenvolvedores em outros sistemas. Para lidar com esse problema, sistemas de controle de versão centralizado (CVCSs) foram desenvolvidos. Esses sistemas (como CVS, Subversion e Perforce) têm um único servidor que contém todos os arquivos com versão e vários clientes que verificam os arquivos desse local central. Por muitos anos, este tem sido o padrão para controle de versão.

Machine Translated by Google

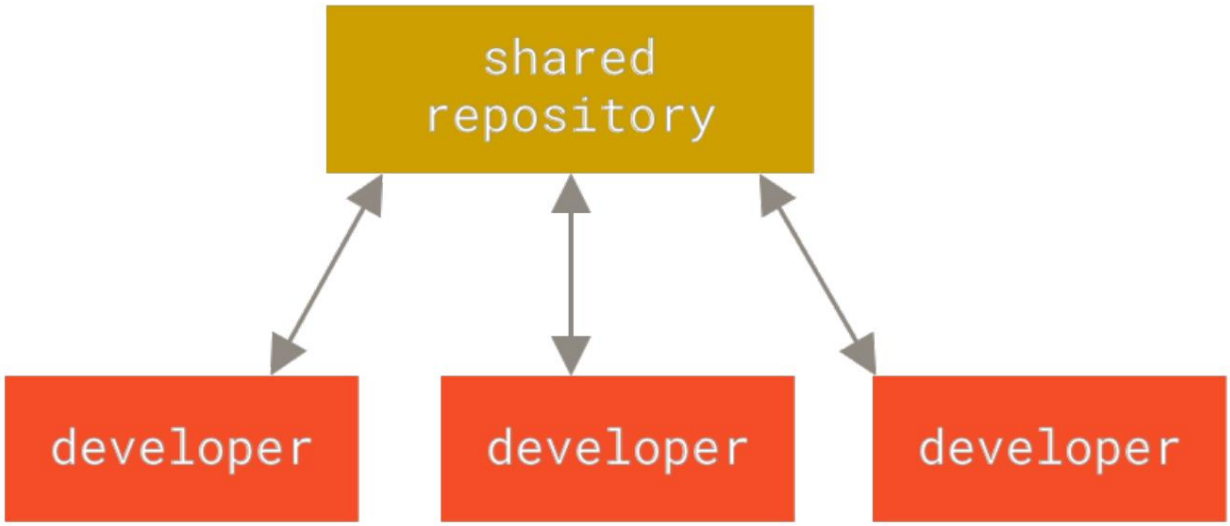


Figura 2. Controle de versão centralizado

Esta configuração oferece muitas vantagens, especialmente sobre VCSs locais. Por exemplo, todo mundo sabe até certo ponto o que todos os outros no projeto estão fazendo. Os administradores têm controle refinado sobre quem pode fazer o quê, e é muito mais fácil administrar um CVCS do que lidar com bancos de dados locais em cada cliente.

No entanto, esta configuração também tem algumas desvantagens sérias. O mais óbvio é o ponto único de falha que o servidor centralizado representa. Se esse servidor ficar inativo por uma hora, durante essa hora ninguém poderá colaborar ou salvar alterações de versão em qualquer coisa em que esteja trabalhando. Se o disco rígido no qual o banco de dados central está corrompido e os backups adequados não foram mantidos, você perde absolutamente tudo - todo o histórico do projeto, exceto quaisquer instantâneos únicos que as pessoas tenham em suas máquinas locais. Os VCSs locais sofrem desse mesmo problema — sempre que você tiver todo o histórico do projeto em um único lugar, corre o risco de perder tudo.

Sistemas de Controle de Versão Distribuída

É aqui que entram os Sistemas de Controle de Versão Distribuída (DVCSs). Em um DVCS (como Git, Mercurial, Bazaar ou Darcs), os clientes não verificam apenas o instantâneo mais recente dos arquivos; em vez disso, eles espelham totalmente o repositório, incluindo seu histórico completo. Portanto, se algum servidor morrer e esses sistemas estiverem colaborando por meio desse servidor, qualquer um dos repositórios do cliente poderá ser copiado de volta para o servidor para restaurá-lo. Cada clone é realmente um backup completo de todos os dados.

Machine Translated by Google

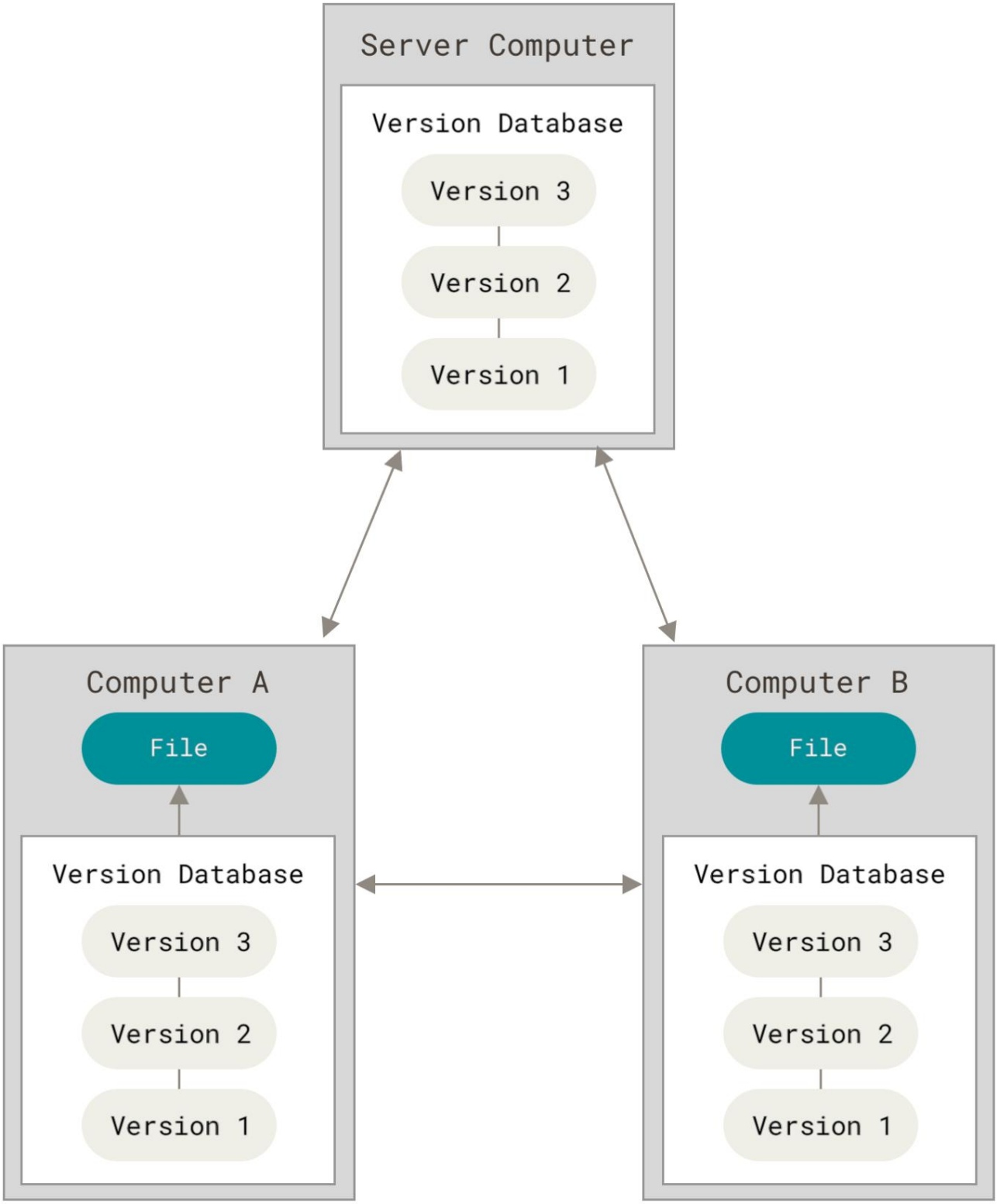


Figura 3. Controle de versão distribuído

Além disso, muitos desses sistemas lidam muito bem com vários repositórios remotos com os quais podem trabalhar, para que você possa colaborar com diferentes grupos de pessoas de maneiras diferentes simultaneamente no mesmo projeto. Isso permite configurar vários tipos de fluxos de trabalho que não são possíveis em sistemas centralizados, como modelos hierárquicos.

Machine Translated by Google

Uma Breve História do Git

Assim como acontece com muitas coisas boas na vida, o Git começou com um pouco de destruição criativa e controvérsia inflamada.

O kernel do Linux é um projeto de software de código aberto de escopo bastante amplo. Durante os primeiros anos da manutenção do kernel do Linux (1991–2002), as alterações no software foram repassadas como patches e arquivos arquivados. Em 2002, o projeto do kernel Linux começou a usar um DVCS proprietário chamado BitKeeper.

Em 2005, o relacionamento entre a comunidade que desenvolveu o kernel do Linux e a empresa comercial que desenvolveu o BitKeeper foi rompido, e o status gratuito da ferramenta foi revogado. Isso levou a comunidade de desenvolvimento do Linux (e em particular Linus Torvalds, o criador do Linux) a desenvolver sua própria ferramenta com base em algumas das lições que aprenderam ao usar o BitKeeper. Alguns dos objetivos do novo sistema foram os seguintes:

- Velocidade
- Design simples
- Forte suporte para desenvolvimento não linear (milhares de ramificações paralelas)
- Totalmente distribuído
- Capaz de lidar com grandes projetos como o kernel do Linux de forma eficiente (velocidade e tamanho dos dados)

Desde o seu nascimento em 2005, o Git evoluiu e amadureceu para ser fácil de usar e ainda manter essas qualidades iniciais. É incrivelmente rápido, muito eficiente com grandes projetos e possui um incrível sistema de ramificação para desenvolvimento não linear (consulte Git Branching).

O que é Git?

Então, o que é Git em poucas palavras? Esta é uma seção importante a ser absorvida, porque se você entender o que é o Git e os fundamentos de como ele funciona, usar o Git de forma eficaz provavelmente será muito mais fácil para você. À medida que você aprende Git, tente limpar sua mente das coisas que você pode saber sobre outros VCSs, como CVS, Subversion ou Perforce — isso o ajudará a evitar confusões sutis ao usar a ferramenta.
Embora a interface do usuário do Git seja bastante semelhante a esses outros VCSs, o Git armazena e pensa sobre as informações de uma maneira muito diferente, e entender essas diferenças ajudará você a evitar confusão ao usá-lo.

Instantâneos, não diferenças

A principal diferença entre o Git e qualquer outro VCS (Subversion e amigos incluídos) é a maneira como o Git pensa sobre seus dados. Conceitualmente, a maioria dos outros sistemas armazena informações como uma lista de alterações baseadas em arquivo. Esses outros sistemas (CVS, Subversion, Perforce, Bazaar e assim por diante) pensam nas informações que armazenam como um conjunto de arquivos e as alterações feitas em cada arquivo ao longo do tempo (isso é comumente descrito como controle de versão baseado em delta) .

Machine Translated by Google

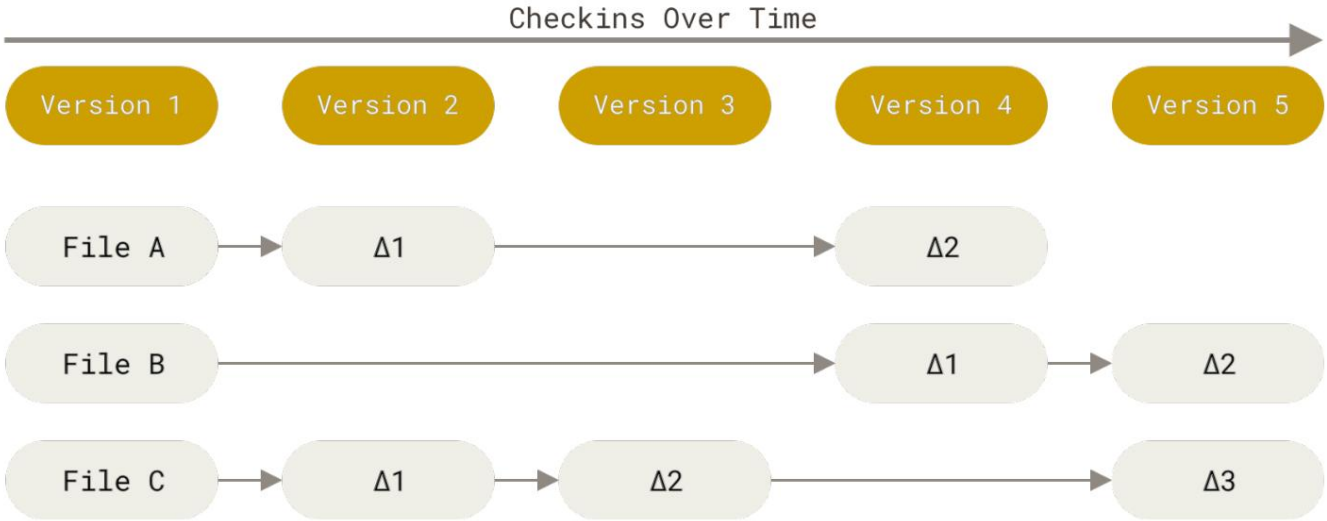


Figura 4. Armazenando dados como alterações em uma versão base de cada arquivo

O Git não pensa ou armazena seus dados dessa maneira. Em vez disso, o Git pensa em seus dados mais como uma série de instantâneos de um sistema de arquivos em miniatura. Com o Git, toda vez que você confirma ou salva o estado do seu projeto, o Git basicamente tira uma foto da aparência de todos os seus arquivos naquele momento e armazena uma referência a esse instantâneo. Para ser eficiente, se os arquivos não foram alterados, o Git não armazena o arquivo novamente, apenas um link para o arquivo anterior idêntico que ele já armazenou. O Git pensa em seus dados mais como um **fluxo de instantâneos.**

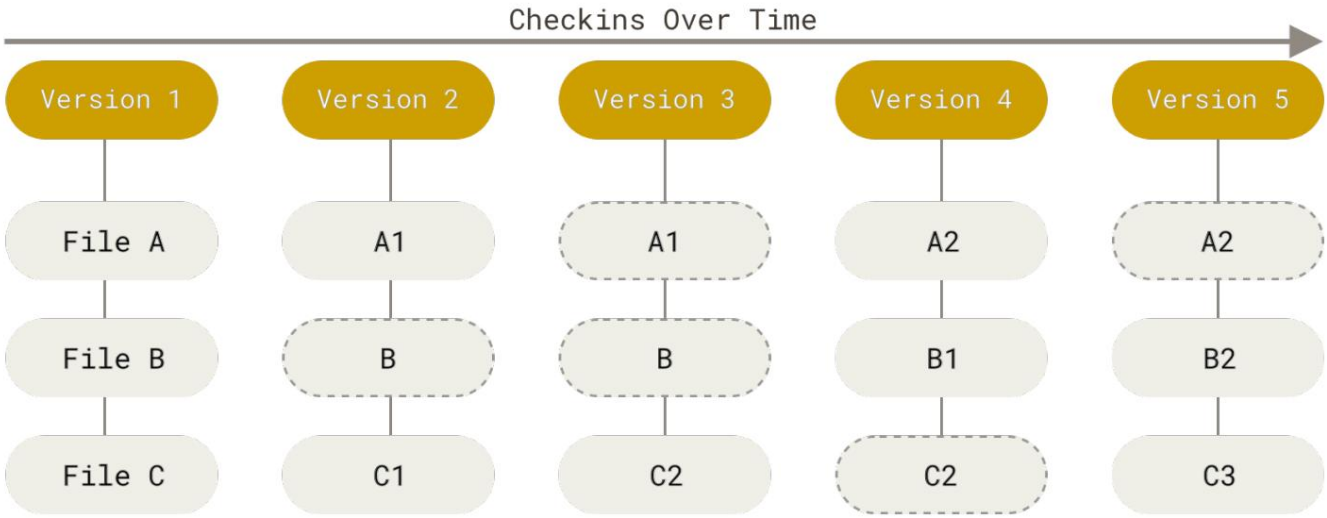


Figura 5. Armazenando dados como instantâneos do projeto ao longo do tempo

Esta é uma distinção importante entre o Git e quase todos os outros VCSs. Isso faz com que o Git reconsidere quase todos os aspectos do controle de versão que a maioria dos outros sistemas copiou da geração anterior. Isso torna o Git mais parecido com um mini sistema de arquivos com algumas ferramentas incrivelmente poderosas construídas sobre ele, em vez de simplesmente um VCS. Exploraremos alguns dos benefícios que você obtém ao pensar em seus dados dessa maneira quando abordamos a ramificação do Git em Git Branching.

Quase todas as operações são locais

A maioria das operações no Git precisa apenas de arquivos e recursos locais para operar — geralmente nenhuma informação é necessária de outro computador em sua rede. Se você está acostumado a um CVCS onde a maioria das operações tem sobrecarga de latência de rede, esse aspecto do Git fará você pensar que os deuses da velocidade abençoaram o Git com poderes sobrenaturais. Como você tem todo o histórico do projeto no disco local, a maioria das operações parece quase instantânea.

Machine Translated by Google

Por exemplo, para navegar pelo histórico do projeto, o Git não precisa ir até o servidor para obter o histórico e exibi-lo para você — ele simplesmente o lê diretamente de seu banco de dados local. Isso significa que você vê o histórico do projeto quase instantaneamente. Se você quiser ver as alterações introduzidas entre a versão atual de um arquivo e o arquivo de um mês atrás, o Git pode procurar o arquivo de um mês atrás e fazer um cálculo de diferença local, em vez de pedir a um servidor remoto para fazer isso ou extraia uma versão mais antiga do arquivo do servidor remoto para fazer isso localmente.

Isso também significa que há muito pouco que você não pode fazer se estiver offline ou sem VPN. Se você entrar em um avião ou trem e quiser fazer um pouco de trabalho, você pode se comprometer alegremente (para sua cópia *local* , lembra?) até chegar a uma conexão de rede para fazer o upload. Se você for para casa e não conseguir que seu cliente VPN funcione corretamente, ainda poderá trabalhar. Em muitos outros sistemas, fazer isso é impossível ou doloroso. No Perforce, por exemplo, você não pode fazer muito quando não está conectado ao servidor; no Subversion e no CVS, você pode editar arquivos, mas não pode enviar alterações ao seu banco de dados (porque seu banco de dados está offline). Isso pode não parecer grande coisa, mas você pode se surpreender com a grande diferença que pode fazer.

Git tem integridade

Tudo no Git recebe uma soma de verificação antes de ser armazenado e, em seguida, é referido por essa soma de verificação. Isso significa que é impossível alterar o conteúdo de qualquer arquivo ou diretório sem que o Git saiba disso. Essa funcionalidade é incorporada ao Git nos níveis mais baixos e é parte integrante de sua filosofia. Você não pode perder informações em trânsito ou corromper arquivos sem que o Git seja capaz de detectá-los.

O mecanismo que o Git usa para essa soma de verificação é chamado hash SHA-1. Esta é uma string de 40 caracteres composta por caracteres hexadecimais (0–9 e a–f) e calculada com base no conteúdo de um arquivo ou estrutura de diretório no Git. Um hash SHA-1 se parece com isto:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Você verá esses valores de hash em todo lugar no Git porque ele os usa muito. Na verdade, o Git armazena tudo em seu banco de dados não pelo nome do arquivo, mas pelo valor de hash de seu conteúdo.

Git geralmente apenas adiciona dados

Quando você executa ações no Git, quase todas elas apenas *adicionam* dados ao banco de dados do Git. É difícil fazer com que o sistema faça algo que não possa ser desfeito ou apague os dados de qualquer forma. Como em qualquer VCS, você pode perder ou atrapalhar as alterações que ainda não fez, mas depois de enviar um instantâneo para o Git, é muito difícil perdê-lo, especialmente se você enviar regularmente seu banco de dados para outro repositório.

Isso torna o uso do Git uma alegria porque sabemos que podemos experimentar sem o perigo de estragar tudo. Para uma visão mais aprofundada de como o Git armazena seus dados e como você pode recuperar dados que parecem perdidos, consulte Desfazendo coisas.

os tres estados

Preste atenção agora — aqui está a principal coisa a ser lembrada sobre o Git se você quiser que o resto do seu processo de aprendizado ocorra sem problemas. O Git tem três estados principais nos quais seus arquivos podem residir: *modificado, testado* e *confirmado:*

Machine Translated by Google

- Modificado significa que você alterou o arquivo, mas ainda não o transferiu para o banco de dados.
- Preparado significa que você marcou um arquivo modificado em sua versão atual para ir para a próxima instantâneo de confirmação.
- Confirmado significa que os dados estão armazenados com segurança em seu banco de dados local.

Isso nos leva às três seções principais de um projeto Git: a árvore de trabalho, a área de preparação e o diretório Git.

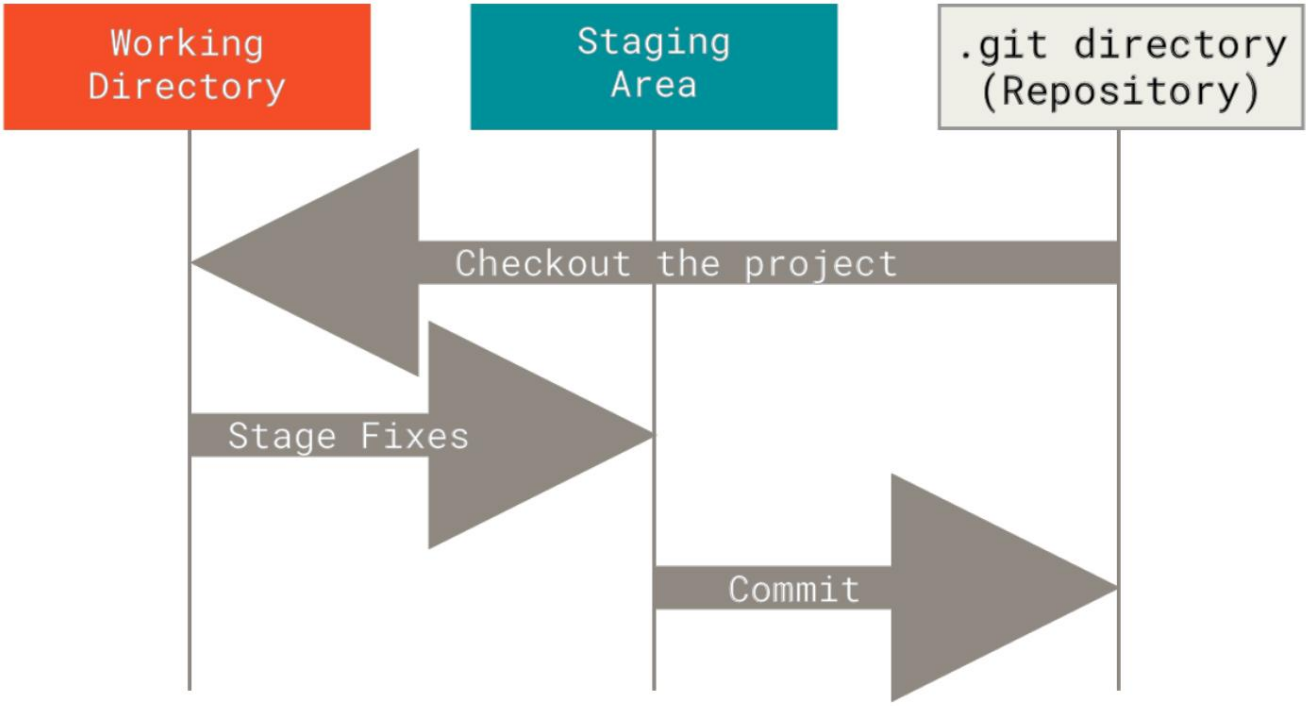


Figura 6. Árvore de trabalho, área de preparação e diretório Git

A árvore de trabalho é um único checkout de uma versão do projeto. Esses arquivos são retirados do banco de dados compactado no diretório Git e colocados no disco para você usar ou modificar.

A área de staging é um arquivo, geralmente contido em seu diretório Git, que armazena informações sobre o que irá para seu próximo commit. Seu nome técnico na linguagem do Git é "índice", mas a frase "área de preparação" também funciona.

O diretório Git é onde o Git armazena os metadados e o banco de dados de objetos do seu projeto. Esta é a parte mais importante do Git, e é o que é copiado quando você *clona* um repositório de outro computador.

O fluxo de trabalho básico do Git é mais ou menos assim:

1. Você modifica arquivos em sua árvore de trabalho.
2. Você prepara seletivamente apenas as alterações que deseja que façam parte de seu próximo commit, o que adiciona *apenas* essas alterações na área de preparação.
3. Você faz uma confirmação, que pega os arquivos como estão na área de preparação e armazena esse instantâneo permanentemente em seu diretório Git.

Se uma versão específica de um arquivo estiver no diretório Git, ela será considerada *confirmada.* Se foi

Machine Translated by Google

modificado e adicionado à área de staging, ele é *staged.* E se foi alterado desde o check-out, mas não foi testado, ele é *modificado.* No Git Basics, você aprenderá mais sobre esses estados e como pode aproveitá-los ou ignorar totalmente a parte do estágio.

A Linha de Comando

Existem várias maneiras diferentes de usar o Git. Existem as ferramentas de linha de comando originais e muitas interfaces gráficas de usuário com recursos variados. Para este livro, usaremos o Git na linha de comando. Por um lado, a linha de comando é o único lugar onde você pode executar *todos* os comandos do Git — a maioria das GUIs implementam apenas um subconjunto parcial da funcionalidade do Git para simplificar. Se você sabe como executar a versão de linha de comando, provavelmente também pode descobrir como executar a versão da GUI, embora o oposto não seja necessariamente verdadeiro. Além disso, embora a escolha do cliente gráfico seja uma questão de gosto pessoal, *todos* os usuários terão as ferramentas de linha de comando instaladas e disponíveis.

Portanto, esperamos que você saiba como abrir Terminal no macOS ou Prompt de Comando ou PowerShell no Windows. Se você não sabe do que estamos falando aqui, pode ser necessário parar e pesquisar rapidamente para poder seguir o restante dos exemplos e descrições deste livro.

Instalando o Git

Antes de começar a usar o Git, você deve disponibilizá-lo em seu computador. Mesmo que já esteja instalado, provavelmente é uma boa ideia atualizar para a versão mais recente. Você pode instalá-lo como um pacote ou por meio de outro instalador, ou baixar o código-fonte e compilá-lo você mesmo.

ÿ

Este livro foi escrito usando o Git versão 2. Como o Git é excelente em preservar a compatibilidade com versões anteriores, qualquer versão recente deve funcionar bem. Embora a maioria dos comandos que usamos deva funcionar mesmo em versões antigas do Git, alguns deles podem não funcionar ou podem agir de maneira ligeiramente diferente.

Instalando no Linux

Se você deseja instalar as ferramentas básicas do Git no Linux por meio de um instalador binário, geralmente pode fazê-lo por meio da ferramenta de gerenciamento de pacotes que vem com sua distribuição. Se você estiver no Fedora (ou qualquer distribuição baseada em RPM intimamente relacionada, como RHEL ou CentOS), você pode usar dnf:

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

Se você estiver em uma distribuição baseada em Debian, como o Ubuntu, tente o apt:

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

Para mais opções, há instruções para instalação em várias distribuições Unix diferentes no site do Git, em https://git-scm.com/download/linux.

Machine Translated by Google

Instalando no macOS

Existem várias maneiras de instalar o Git em um Mac. O mais fácil é provavelmente instalar as ferramentas de linha de comando do Xcode. No Mavericks (10.9) ou superior, você pode fazer isso simplesmente tentando executar o git do Terminal pela primeira vez.

```
$ git --version
```

Se você ainda não o tiver instalado, ele solicitará que você o instale.

Se você deseja uma versão mais atualizada, também pode instalá-la por meio de um instalador binário. Um instalador do macOS Git é mantido e está disponível para download no site do Git, em https://git-scm.com/download/mac .

Figura 7. Git macOS Installer

Instalando no Windows

Existem também algumas maneiras de instalar o Git no Windows. A compilação mais oficial está disponível para download no site do Git. Basta acessar https://git-scm.com/download/win e o download começará automaticamente. Observe que este é um projeto chamado Git for Windows, que é separado do próprio Git; para mais informações, acesse https://gitforwindows.org.

Para obter uma instalação automatizada, você pode usar o pacote Git Chocolatey. Observe que o pacote Chocolatey é mantido pela comunidade.

Machine Translated by Google

Instalando da fonte

Algumas pessoas podem achar útil instalar o Git a partir do código-fonte, porque você obterá a versão mais recente. Os instaladores binários tendem a ficar um pouco atrasados, embora, como o Git amadureceu nos últimos anos, isso fez menos diferença.

Se você deseja instalar o Git a partir do código-fonte, precisa ter as seguintes bibliotecas das quais o Git depende: autotools, curl, zlib, openssl, expat e libiconv. Por exemplo, se você estiver em um sistema que possui dnf (como Fedora) ou apt-get (como um sistema baseado em Debian), você pode usar um desses comandos para instalar as dependências mínimas para compilar e instalar o Git binários:

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

Para poder adicionar a documentação em vários formatos (doc, html, info), essas dependências adicionais são necessárias:

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

ÿ Usuários de RHEL e derivados de RHEL como CentOS e Scientific Linux terão que ativar o repositório EPEL para baixar o pacote docbook2X .

Se você estiver usando uma distribuição baseada em Debian (Debian/Ubuntu/Ubuntu-derivatives), também precisará do pacote install-info :

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

Se você estiver usando uma distribuição baseada em RPM (Fedora/RHEL/RHEL-derivatives), também precisará do pacote getopt (que já está instalado em uma distribuição baseada em Debian):

```
$ sudo dnf install getopt
```

Além disso, se você estiver usando derivados do Fedora/RHEL/RHEL, precisará fazer o seguinte:

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

devido a diferenças de nomes binários.

Quando você tiver todas as dependências necessárias, poderá ir em frente e pegar o tarball de lançamento marcado mais recente de vários lugares. Você pode obtê-lo através do site kernel.org, em https://www.kernel.org/pub/software/scm/git , ou o espelho no site do GitHub, em https://github.com/git/git/releases. Isso é

Machine Translated by Google

geralmente um pouco mais claro qual é a versão mais recente na página do GitHub, mas a página kernel.org também possui assinaturas de lançamento se você quiser verificar seu download.

Em seguida, compile e instale:

```
$ 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- informação
```

Feito isso, você também pode obter o Git via Git para atualizações:

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

Primeira configuração do Git

Agora que você tem o Git em seu sistema, convém fazer algumas coisas para personalizar seu ambiente Git. Você deve fazer essas coisas apenas uma vez em qualquer computador; eles permanecerão entre as atualizações. Você também pode alterá-los a qualquer momento executando os comandos novamente.

O Git vem com uma ferramenta chamada git config que permite obter e definir variáveis de configuração que controlam todos os aspectos de como o Git se parece e opera. Essas variáveis podem ser armazenadas em três lugares diferentes:

1. [path]/etc/gitconfig file: Contém valores aplicados a cada usuário no sistema e todos os seus repositórios. Se você passar a opção --system para git config, ele lê e escreve deste arquivo especificamente. Como este é um arquivo de configuração do sistema, você precisaria de privilégio administrativo ou de superusuário para fazer alterações nele.

2. Arquivo ~/.gitconfig ou ~/.config/git/config : Valores específicos para você, o usuário. Você pode fazer o Git ler e gravar neste arquivo especificamente passando a opção --global , e isso afeta *todos* os repositórios com os quais você trabalha em seu sistema.

3. arquivo de configuração no diretório Git (ou seja, .git/config) de qualquer repositório que você esteja usando atualmente: Específico para aquele único repositório. Você pode forçar o Git a ler e gravar neste arquivo com a opção --local , mas isso é de fato o padrão. Sem surpresa, você precisa estar localizado em algum lugar em um repositório Git para que esta opção funcione corretamente.

Cada nível substitui os valores do nível anterior, então os valores em .git/config superam aqueles em [path]/etc/gitconfig.

Em sistemas Windows, o Git procura o arquivo .gitconfig no diretório $HOME (C:\Users\$USER para a maioria das pessoas). Ele também ainda procura por [path]/etc/gitconfig, embora seja relativo à raiz do MSys, que é onde quer que você decida instalar o Git em seu sistema Windows ao executar o instalador. Se você estiver usando a versão 2.x ou posterior do Git para Windows, também há um arquivo de configuração no nível do sistema em C:\Documents and Settings\All Users\Application Data\Git\config no Windows XP e em

Machine Translated by Google

C:\ProgramData\Git\config no Windows Vista e mais recente. Este arquivo de configuração só pode ser alterado por git config -f <arquivo> como administrador.

Você pode visualizar todas as suas configurações e de onde elas vêm usando:

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

Sua identidade

A primeira coisa que você deve fazer ao instalar o Git é definir seu nome de usuário e endereço de e-mail. Isso é importante porque todo commit do Git usa essas informações, e elas são integradas de forma imutável nos commits que você começa a criar:

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

Novamente, você precisa fazer isso apenas uma vez se passar a opção --global , porque o Git sempre usará essa informação para qualquer coisa que você fizer nesse sistema. Se você deseja substituir isso por um nome ou endereço de e-mail diferente para projetos específicos, pode executar o comando sem a opção --global quando estiver nesse projeto.

Muitas das ferramentas da GUI o ajudarão a fazer isso quando você as executar pela primeira vez.

seu editor

Agora que sua identidade está configurada, você pode configurar o editor de texto padrão que será usado quando o Git precisar que você digite uma mensagem. Se não estiver configurado, o Git usa o editor padrão do seu sistema.

Se você quiser usar um editor de texto diferente, como o Emacs, faça o seguinte:

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

Em um sistema Windows, se você deseja usar um editor de texto diferente, deve especificar o caminho completo para seu arquivo executável. Isso pode ser diferente dependendo de como seu editor é empacotado.

No caso do Notepad++, um editor de programação popular, é provável que você queira usar a versão de 32 bits, já que no momento da escrita a versão de 64 bits não suporta todos os plug-ins. Se você estiver em um sistema Windows de 32 bits ou tiver um editor de 64 bits em um sistema de 64 bits, digite algo como isto:

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

ÿ

Vim, Emacs e Notepad++ são editores de texto populares frequentemente usados por desenvolvedores em sistemas baseados em Unix como Linux e macOS ou um sistema Windows. Se você estiver usando

Machine Translated by Google

outro editor ou uma versão de 32 bits, encontre instruções específicas sobre como configurar seu editor favorito com o Git nos comandos git config core.editor.

ÿ

Você pode descobrir que, se não configurar seu editor dessa maneira, ficará confuso quando o Git tentar iniciá-lo. Um exemplo em um sistema Windows pode incluir uma operação Git encerrada prematuramente durante uma edição iniciada pelo Git.

Seu nome de filial padrão

Por padrão, o Git criará uma ramificação chamada *master* quando você criar um novo repositório com git init.
A partir da versão 2.28 do Git, você pode definir um nome diferente para o branch inicial.

Para definir *main* como o nome da ramificação padrão, faça:

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

Verificando suas configurações

Se quiser verificar suas definições de configuração, você pode usar o comando git config --list para listar todas as configurações que o Git pode encontrar nesse ponto:

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

color.diff=auto
...
```

Você pode ver as chaves mais de uma vez, porque o Git lê a mesma chave de arquivos diferentes ([caminho]/etc/gitconfig e ~/.gitconfig, por exemplo). Nesse caso, o Git usa o último valor para cada chave exclusiva que vê.

Você também pode verificar o que o Git acha que é o valor de uma chave específica digitando git config <chave>:

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

ÿ

Como o Git pode ler o mesmo valor de variável de configuração de mais de um arquivo, é possível que você tenha um valor inesperado para um desses valores e não saiba por quê. Em casos como esse, você pode consultar o Git quanto à *origem* desse valor, e ele informará qual arquivo de configuração teve a palavra final na definição desse valor:

Machine Translated by Google

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

Conseguindo ajuda

Se você precisar de ajuda ao usar o Git, existem três maneiras equivalentes de obter a ajuda abrangente da página de manual (manpage) para qualquer um dos comandos do Git:

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

Por exemplo, você pode obter a ajuda da página de manual para o comando git config executando isto:

```
$ configuração de ajuda do git
```

Esses comandos são bons porque você pode acessá-los em qualquer lugar, mesmo offline. Se as páginas de manual e este livro não forem suficientes e você precisar de ajuda pessoalmente, tente os canais #git, #github ou #gitlab no servidor Libera Chat IRC, que pode ser encontrado em https://libera. bater papo/. Esses canais são regularmente preenchidos com centenas de pessoas que conhecem muito bem o Git e geralmente estão dispostas a ajudar.

Além disso, se você não precisa da ajuda completa da página de manual, mas apenas precisa de uma atualização rápida sobre as opções disponíveis para um comando Git, você pode solicitar a saída de “ajuda” mais concisa com a opção -h, como em :

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

-n, --dry-run -v, --                teste
verbose                             seja detalhado

-i, --interactive -p, --            seleção interativa
patch -e, --edit -f, --force        selecionar trechos editar
-u, --update --renormalize          interativamente o diff atual e aplicar
                                    permitir adicionar arquivos de outra forma
                                    ignorados atualizar arquivos rastreados
                                    renormalizar EOL de arquivos rastreados (implica -u)
-N, --intent-to-add                 registrar apenas o fato de que o caminho será adicionado posteriormente
-A, --all --ignore-                 adicionar alterações de todos os arquivos rastreados e não rastreados
removal -all) --refresh             ignorar caminhos removidos na árvore de trabalho (o mesmo que --no

                                    não adicione, apenas atualize o índice
--ignore-errors                     apenas pule arquivos que não podem ser adicionados por causa de
erros
```

Machine Translated by Google

--ignore-missing --chmod (+|-)x --pathspec-from-file <arquivo> lê pathspec do arquivo --pathspec-file-nul com --pathspec-from-file, elementos pathspec são separados com NUL

verifique se - mesmo ausentes - os arquivos são ignorados no teste de substituição do bit executável dos arquivos listados

Resumo

Você deve ter uma compreensão básica do que é o Git e como ele é diferente de qualquer sistema de controle de versão centralizado que você possa ter usado anteriormente. Agora você também deve ter uma versão funcional do Git em seu sistema configurada com sua identidade pessoal. Agora é hora de aprender alguns fundamentos do Git.

Machine Translated by Google

Se você pode ler apenas um capítulo para começar com Git, é isso. Este capítulo cobre cada comando básico que você precisa para fazer a grande maioria das coisas que eventualmente você gastará seu tempo fazendo com o Git. No final do capítulo, você será capaz de configurar e inicializar um repositório, iniciar e interromper o rastreamento de arquivos e preparar e confirmar alterações. Também mostraremos como configurar o Git para ignorar certos arquivos e padrões de arquivo, como desfazer erros de forma rápida e fácil, como navegar no histórico do seu projeto e visualizar as alterações entre os commits e como fazer push e pull de repositórios remotos .

Obtendo um repositório Git

Normalmente, você obtém um repositório Git de duas maneiras:

1. Você pode pegar um diretório local que atualmente não está sob controle de versão e transformá-lo em um Git repositório, ou
2. Você pode *clonar* um repositório Git existente de outro lugar.

Em ambos os casos, você acaba com um repositório Git em sua máquina local, pronto para trabalhar.

Inicializando um repositório em um diretório existente

Se você tem um diretório de projeto que atualmente não está sob controle de versão e deseja começar a controlá-lo com Git, primeiro você precisa ir para o diretório desse projeto. Se você nunca fez isso, parece um pouco diferente dependendo de qual sistema você está executando:

para Linux:

```
$ cd /home/user/meu_projeto
```

para macOS:

```
$ cd /Users/user/meu_projeto
```

para Windows:

```
$ cd C:/Usuários/usuário/meu_projeto
```

e digite:

```
$ git calor
```

Isso cria um novo subdiretório chamado .git que contém todos os seus arquivos de repositório necessários — um esqueleto de repositório Git. Neste ponto, nada em seu projeto é rastreado ainda. Consulte Git Internals para

Machine Translated by Google

mais informações sobre exatamente quais arquivos estão contidos no diretório .git que você acabou de criar.

Se você deseja iniciar o controle de versão dos arquivos existentes (em oposição a um diretório vazio), provavelmente deve começar a rastrear esses arquivos e fazer um commit inicial. Você pode fazer isso com alguns comandos git add que especificam os arquivos que deseja rastrear, seguidos por um git commit:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Versão inicial do projeto'
```

Veremos o que esses comandos fazem em apenas um minuto. Neste ponto, você tem um repositório Git com arquivos rastreados e um commit inicial.

Clonando um repositório existente

Se você deseja obter uma cópia de um repositório Git existente — por exemplo, um projeto para o qual gostaria de contribuir — o comando necessário é git clone. Se você estiver familiarizado com outros VCSs, como o Subversion, notará que o comando é "clone" e não "checkout". Esta é uma distinção importante — em vez de obter apenas uma cópia de trabalho, o Git recebe uma cópia completa de quase todos os dados que o servidor possui. Cada versão de cada arquivo para o histórico do projeto é baixada por padrão quando você executa o git clone. Na verdade, se o disco do seu servidor for corrompido, você pode usar praticamente qualquer um dos clones em qualquer cliente para definir o servidor de volta ao estado em que estava quando foi clonado (você pode perder alguns ganchos do lado do servidor e outros, mas todos os dados versionados estariam lá — veja Obtendo o Git em um servidor para mais detalhes).

Você clona um repositório com git clone <url>. Por exemplo, se você deseja clonar a biblioteca vinculável do Git chamada libgit2, pode fazer assim:

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

Isso cria um diretório chamado libgit2, inicializa um diretório .git dentro dele, extrai todos os dados desse repositório e verifica uma cópia de trabalho da versão mais recente. Se você entrar no novo diretório libgit2 que acabou de ser criado, verá os arquivos do projeto lá, prontos para serem trabalhados ou usados.

Se você deseja clonar o repositório em um diretório com nome diferente de libgit2, pode especificar o novo nome do diretório como um argumento adicional:

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

Esse comando faz a mesma coisa que o anterior, mas o diretório de destino é chamado mylibgit.

O Git tem vários protocolos de transferência diferentes que você pode usar. O exemplo anterior usa o protocolo https:// , mas você também pode ver git:// ou user@server:path/to/repo.git, que usa o protocolo de transferência SSH. Obter o Git em um servidor apresentará todas as opções disponíveis que o servidor pode configurar para acessar seu repositório Git e os prós e contras de cada um.

Machine Translated by Google

Gravando Alterações no Repositório

Neste ponto, você deve ter um repositório Git *de boa-fé* em sua máquina local e um checkout ou *cópia de trabalho* de todos os seus arquivos à sua frente. Normalmente, você deseja começar a fazer alterações e enviar instantâneos dessas alterações para o seu repositório sempre que o projeto atingir um estado que você deseja registrar.

Lembre-se de que cada arquivo em seu diretório de trabalho pode estar em um dos dois estados: *rastreado* ou *não rastreado.* Arquivos rastreados são arquivos que estavam no último instantâneo, bem como quaisquer arquivos recém-preparados; eles podem ser inalterados, modificados ou preparados. Resumindo, arquivos rastreados são arquivos que o Git conhece.

Arquivos não rastreados são todo o resto - quaisquer arquivos em seu diretório de trabalho que não estavam em seu último instantâneo e não estão em sua área de preparação. Quando você clona um repositório pela primeira vez, todos os seus arquivos serão rastreados e não modificados porque o Git apenas os verificou e você não editou nada.

À medida que você edita arquivos, o Git os vê como modificados, porque você os alterou desde seu último commit.
À medida que trabalha, você prepara seletivamente esses arquivos modificados e, em seguida, confirma todas as alterações preparadas e o ciclo se repete.

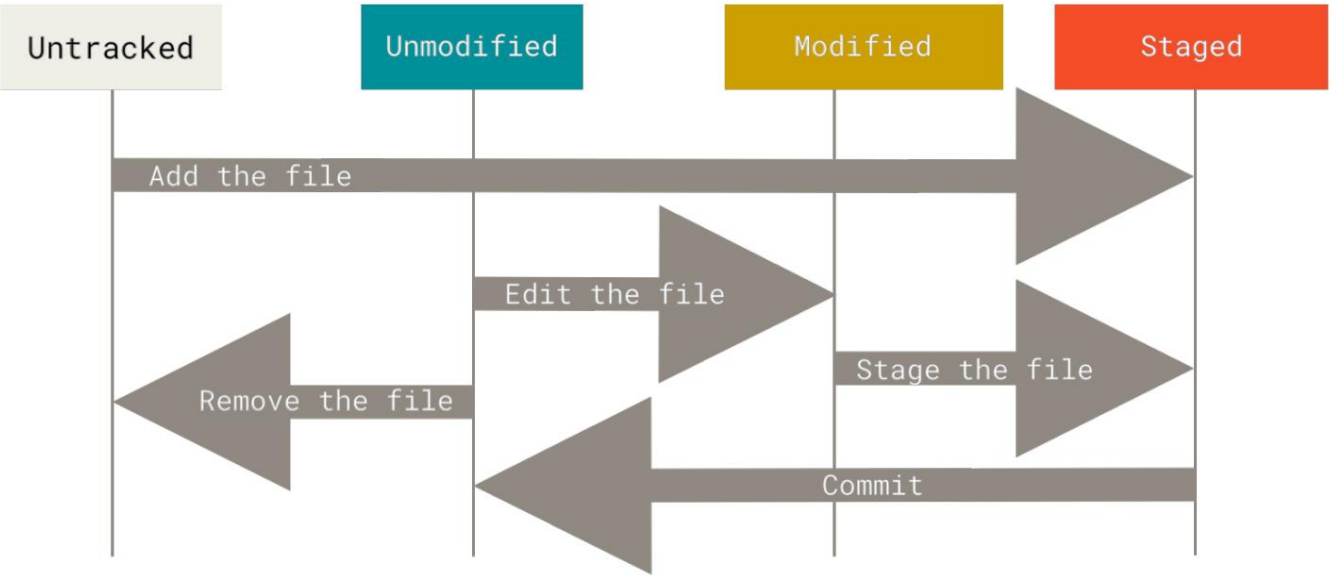


Figura 8. O ciclo de vida do status de seus arquivos

Verificando o status de seus arquivos

A principal ferramenta que você usa para determinar quais arquivos estão em qual estado é o comando git status . Se você executar este comando diretamente após um clone, deverá ver algo como isto:

```
$ git status
On branch master
Sua branch está atualizada com 'origin/master'.
nada a comprometer, árvore de trabalho limpa
```

Isso significa que você tem um diretório de trabalho limpo; em outras palavras, nenhum de seus arquivos rastreados é modificado. O Git também não vê nenhum arquivo não rastreado, ou eles seriam listados aqui. Por fim, o comando informa em qual ramo você está e informa que não divergiu do mesmo

Machine Translated by Google

filial no servidor. Por enquanto, essa ramificação é sempre mestre, que é o padrão; você não vai se preocupar com isso aqui. Git Branching examinará ramificações e referências em detalhes.

ÿ

O GitHub mudou o nome da ramificação padrão de master para main em meados de 2020, e outros hosts do Git seguiram o exemplo. Portanto, você pode descobrir que o nome da ramificação padrão em alguns repositórios recém-criados é principal e não mestre. Além disso, o nome da filial padrão pode ser alterado (como você viu em Seu nome de filial padrão), então você pode ver um nome diferente para a filial padrão.

No entanto, o próprio Git ainda usa master como padrão, então vamos usá-lo ao longo do livro.

Digamos que você adicione um novo arquivo ao seu projeto, um simples arquivo README . Se o arquivo não existia antes e você executar git status, verá seu arquivo não rastreado da seguinte forma:

```
$ echo 'Meu Projeto' > README $
git status On branch master Seu
branch está atualizado com 'origin/
master'.
Arquivos não
rastreados: (use "git add <arquivo>..." para incluir no que será confirmado)

Leia-me


nada adicionado ao commit, mas arquivos não rastreados presentes (use "git add" para rastrear)
```

Você pode ver que seu novo arquivo README não foi rastreado, porque está sob o título “Arquivos não rastreados” em sua saída de status. Untracked basicamente significa que o Git vê um arquivo que você não tinha no instantâneo anterior (commit) e que ainda não foi testado; O Git não começará a incluí-lo em seus instantâneos de confirmação até que você diga explicitamente para fazer isso. Ele faz isso para que você não comece acidentalmente a incluir arquivos binários gerados ou outros arquivos que você não pretendia incluir. Você quer começar a incluir o README, então vamos começar a rastrear o arquivo.

Rastreando Novos Arquivos

Para começar a rastrear um novo arquivo, você usa o comando git add. Para começar a rastrear o arquivo README , você pode executar isto:

```
$ git add README
```

Se você executar seu comando status novamente, poderá ver que seu arquivo README agora está rastreado e preparado para ser confirmado:

```
$ git status On
branch master Sua
branch está atualizada com 'origin/master'.
Mudanças a serem confirmadas:
```

Machine Translated by Google

```
(use "git restore --staged <arquivo>..." para remover o estágio)

novo arquivo: README
```

Você pode dizer que é preparado porque está sob o cabeçalho “Mudanças a serem confirmadas”. Se você confirmar neste ponto, a versão do arquivo no momento em que você executou o git add é o que estará no instantâneo histórico subsequente. Você deve se lembrar que, quando executou git init anteriormente, você executou git add <files> — para começar a rastrear arquivos em seu diretório. O comando git add usa um nome de caminho para um arquivo ou diretório; se for um diretório, o comando adiciona todos os arquivos desse diretório recursivamente.

Preparando Arquivos Modificados

Vamos alterar um arquivo que já foi rastreado. Se você alterar um arquivo rastreado anteriormente chamado CONTRIBUTING.md e executar o comando git status novamente, obterá algo parecido com isto:

```
$ git status On
branch master Sua
branch está atualizada com 'origin/master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

novo arquivo: README

Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido)
(use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md
```

O arquivo CONTRIBUTING.md aparece em uma seção denominada “Changes not staged for commit” — o que significa que um arquivo rastreado foi modificado no diretório de trabalho, mas ainda não foi testado. Para testá-lo, execute o comando git add . git add é um comando multiuso — você o usa para começar a rastrear novos arquivos, preparar arquivos e fazer outras coisas, como marcar arquivos em conflito de mesclagem como resolvidos. Pode ser útil pensar nisso mais como “adicionar precisamente este conteúdo ao próximo commit” em vez de “adicionar este arquivo ao projeto”. Vamos executar git add agora para preparar o arquivo CONTRIBUTING.md e, em seguida, executar git status novamente:

```
$ git add CONTRIBUTING.md $
git status No branch master Seu
branch está atualizado com 'origin/
master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

novo arquivo: README
```

Machine Translated by Google

```
modificado: CONTRIBUTING.md
```

Ambos os arquivos são testados e irão para o seu próximo commit. Neste ponto, suponha que você se lembre de uma pequena alteração que deseja fazer em CONTRIBUTING.md antes de confirmá-la. Você o abre novamente e faz essa alteração e está pronto para confirmar. No entanto, vamos executar git status mais uma vez:

```
$ vim CONTRIBUTING.md
$ git status No branch master
Seu branch está atualizado
com 'origin/master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

novo arquivo: README
modificado: CONTRIBUTING.md

Alterações não testadas para confirmação:
(use "git add <arquivo>..." para atualizar o que será submetido) (use "git checkout
-- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md
```

Que diabos? Agora CONTRIBUTING.md está listado como testado *e* não testado. Como isso é possível? Acontece que o Git prepara um arquivo exatamente como quando você executa o comando git add . Se você confirmar agora, a versão de CONTRIBUTING.md como estava quando você executou o comando git add pela última vez é como ele irá para o commit, não a versão do arquivo que aparece em seu diretório de trabalho quando você executa git commit. Se você modificar um arquivo depois de executar o git add, será necessário executar o git add novamente para testar a versão mais recente do arquivo:

```
$ git add CONTRIBUTING.md $
git status No branch master Seu
branch está atualizado com 'origin/
master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

novo arquivo: README
modificado: CONTRIBUTING.md
```

Status Curto

Embora a saída git status seja bastante abrangente, também é bastante prolixo. O Git também possui um sinalizador de status curto para que você possa ver suas alterações de maneira mais compacta. Se você executar git status -s ou git status --short, obterá uma saída muito mais simplificada do comando:

```
$ git status -s
M LEIA-ME
```

Machine Translated by Google

```
MM Rakefile
A lib/git.rb M
lib/simplegit.rb ??
LICENÇA.txt
```

Novos arquivos que não são rastreados têm um ?? ao lado deles, os novos arquivos que foram adicionados à área de preparação têm um A, os arquivos modificados têm um M e assim por diante. Existem duas colunas para a saída — a coluna da esquerda indica o status da área de preparação e a coluna da direita indica o status da árvore de trabalho. Assim, por exemplo, nessa saída, o arquivo README é modificado no diretório de trabalho, mas ainda não preparado, enquanto o arquivo lib/simplegit.rb é modificado e preparado. O Rakefile foi modificado, testado e depois modificado novamente, portanto, há alterações nele que são testadas e não testadas.

Ignorando arquivos

Freqüentemente, você terá uma classe de arquivos que não deseja que o Git adicione automaticamente ou mesmo mostre como não rastreados. Geralmente, são arquivos gerados automaticamente, como arquivos de log ou arquivos produzidos pelo seu sistema de compilação. Nesses casos, você pode criar um arquivo listando padrões para corresponder a eles, denominado .gitignore. Aqui está um exemplo de arquivo .gitignore :

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

A primeira linha diz ao Git para ignorar quaisquer arquivos que terminem em “.o” ou “.a” — objetos e arquivos compactados que podem ser o produto da construção do seu código. A segunda linha diz ao Git para ignorar todos os arquivos cujos nomes terminam com um til (~), que é usado por muitos editores de texto, como o Emacs, para marcar arquivos temporários. Você também pode incluir um diretório log, tmp ou pid; documentação gerada automaticamente; e assim por diante. Configurar um arquivo .gitignore para seu novo repositório antes de começar geralmente é uma boa ideia para que você não envie acidentalmente arquivos que realmente não deseja em seu repositório Git.

As regras para os padrões que você pode colocar no arquivo .gitignore são as seguintes:

- Linhas em branco ou linhas começando com # são ignoradas.
- Padrões glob padrão funcionam e serão aplicados recursivamente ao longo de todo o trabalho árvore.
- Você pode iniciar os padrões com uma barra (/) para evitar a recursividade.
- Você pode terminar os padrões com uma barra (/) para especificar um diretório.
- Você pode negar um padrão iniciando-o com um ponto de exclamação (!).

Padrões glob são como expressões regulares simplificadas que os shells usam. Um asterisco (*) corresponde a zero ou mais caracteres; [abc] corresponde a qualquer caractere dentro dos colchetes (neste caso a, b ou c); um ponto de interrogação (?) corresponde a um único caractere; e colchetes envolvendo caracteres separados por um hífen ([0-9]) corresponde a qualquer caractere entre eles (neste caso, de 0 a 9). Você também pode usar dois asteriscos para corresponder a diretórios aninhados; a/**/z corresponderia a a/z, a/b/z, a/b/c/z e assim por diante.

Machine Translated by Google

Aqui está outro exemplo de arquivo .gitignore :

```
# ignora todos os arquivos .a
*.a

# mas rastreie lib.a, mesmo que você esteja ignorando os arquivos .a acima de !lib.a


# ignora apenas o arquivo TODO no diretório atual, não subdir/TODO
/TODO

# ignora todos os arquivos em qualquer diretório chamado build
build/

# ignora doc/notes.txt, mas não doc/server/arch.txt doc/*.txt


# ignora todos os arquivos .pdf no diretório doc/ e qualquer um de seus subdiretórios doc/**/*.pdf
```

ÿ

O GitHub mantém uma lista bastante abrangente de bons exemplos de arquivos .gitignore para dezenas de projetos e idiomas em https://github.com/github/gitignore se você quer um ponto de partida para o seu projeto.

ÿ

No caso simples, um repositório pode ter um único arquivo .gitignore em seu diretório raiz, que se aplica recursivamente a todo o repositório. No entanto, também é possível ter arquivos .gitignore adicionais em subdiretórios. As regras nesses arquivos .gitignore aninhados se aplicam apenas aos arquivos no diretório em que estão localizados. O repositório fonte do kernel Linux tem 206 arquivos .gitignore .

Está além do escopo deste livro entrar nos detalhes de vários arquivos .gitignore ; veja man gitignore para mais detalhes.

Visualizando suas alterações preparadas e não preparadas

Se o comando git status for muito vago para você — você quer saber exatamente o que mudou, não apenas quais arquivos foram alterados — você pode usar o comando git diff . Abordaremos o git diff com mais detalhes posteriormente, mas você provavelmente o usará com mais frequência para responder a estas duas perguntas: O que você mudou, mas ainda não preparou? E o que você encenou que está prestes a cometer? Embora o git status responda a essas perguntas de maneira geral, listando os nomes dos arquivos, o git diff mostra as linhas exatas adicionadas e removidas — o patch, por assim dizer.

Digamos que você edite e teste o arquivo LEIA-ME novamente e, em seguida, edite o arquivo CONTRIBUTING.md sem prepará-lo. Se você executar seu comando git status , mais uma vez verá algo assim:

```
$ git status
No mestre da filial
```

Machine Translated by Google

```
Sua filial está atualizada com 'origin/master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

modificado: README


Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido)
(use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md
```

Para ver o que você mudou, mas ainda não preparou, digite git diff sem outros argumentos:

```
$ git diff diff
--git a/CONTRIBUTING.md b/CONTRIBUTING.md index
8ebb991..643e24f 100644 --- a/CONTRIBUTING.md +++ b/
CONTRIBUTING.md @@ -65,7 +65,8 @ @ ramo
diretamente, as coisas podem ficar confusas.

Inclua uma boa descrição de suas alterações ao enviar seu PR; se tivermos que ler todo o diff
para descobrir por que você está contribuindo em primeiro lugar, é menos provável que você
receba feedback e tenha sua alteração mesclada. Além disso, divida suas alterações em partes
abrangentes se seu patch tem mais de uma dúzia de linhas.



Se você está começando a trabalhar em uma área específica, sinta-se à vontade para enviar um
PR que destaque seu trabalho em andamento (e observe no título do PR que é
```

Esse comando compara o que está em seu diretório de trabalho com o que está em sua área de preparação. O resultado informa as alterações feitas que ainda não foram testadas.

Se você quiser ver o que preparou para o próximo commit, você pode usar git diff --staged. Este comando compara suas alterações preparadas com seu último commit:

```
$ git diff --staged diff --
git a/README b/README novo
modo de arquivo 100644 índice
0000000..03902a1 --- /dev/null +++
b/README @@ -0,0 +1 @@ +My
Projeto
```

É importante observar que o git diff por si só não mostra todas as alterações feitas desde seu último commit — apenas as alterações que ainda não foram testadas. Se você preparou todas as suas alterações, o git diff não fornecerá saída.

Machine Translated by Google

Para outro exemplo, se você preparar o arquivo CONTRIBUTING.md e depois editá-lo, poderá usar git diff para ver as alterações no arquivo que são testadas e as alterações que não são testadas. Se nosso ambiente estiver assim:

```
$ git add CONTRIBUTING.md $
echo '# test line' >> CONTRIBUTING.md $ git status
On branch master Seu branch está atualizado com
'origin/master'.

Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

modificado: CONTRIBUTING.md

Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido)
(use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md
```

Agora você pode usar o git diff para ver o que ainda não foi testado:

```
$ git diff diff
--git a/CONTRIBUTING.md b/CONTRIBUTING.md index
643e24f..87f08c8 100644
--- a/CONTRIBUTING.md +
++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ no ##
Starter Projects

Veja nossa [lista de
projetos](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md). +# linha de teste
```

e git diff --cached para ver o que você preparou até agora (--staged e --cached são sinônimos):

```
$ 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 @@ ramo diretamente, as coisas podem ficar complicadas.
Inclua uma boa descrição de suas alterações ao enviar seu PR; se tivermos que ler todo o diff para
descobrir por que você está contribuindo em primeiro lugar, é menos provável que você receba
feedback e tenha sua alteração mesclada. Além disso, divida suas alterações em partes abrangentes
se seu patch tem mais de uma dúzia de linhas.
```

Machine Translated by Google

Se você está começando a trabalhar em uma área específica, sinta-se à vontade para enviar um PR que destaque seu trabalho em andamento (e observe no título do PR que é

ÿ

Git Diff em uma ferramenta externa

Continuaremos a usar o comando git diff de várias maneiras ao longo do livro. Existe outra maneira de ver essas diferenças se você preferir um programa gráfico ou externo de visualização de diferenças. Se você executar git difftool em vez de git diff, poderá visualizar qualquer um desses diffs em software como emerge, vimdiff e muitos mais (incluindo produtos comerciais). Execute git difftool --tool-help para ver o que está disponível em seu sistema.

Confirmando suas alterações

Agora que sua área de preparação está configurada da maneira que você deseja, você pode confirmar suas alterações. Lembre-se de que tudo o que ainda não foi testado - quaisquer arquivos que você criou ou modificou e não executou o git add desde que os editou - não entrará neste commit. Eles permanecerão como arquivos modificados em seu disco. Nesse caso, digamos que na última vez que você executou o git status, você viu que tudo estava preparado, então você está pronto para fazer o commit de suas alterações. A maneira mais simples de confirmar é digitar git commit:

```
$ git commit
```

Fazer isso inicia o editor de sua escolha.

ÿ

Isso é definido pela variável de ambiente EDITOR do seu shell — geralmente vim ou emacs, embora você possa configurá-lo com o que quiser usando o comando git config --global core.editor , conforme visto em Introdução.

O editor exibe o seguinte texto (este exemplo é uma tela do Vim):

```
# Por favor, insira a mensagem de confirmação para suas alterações. Linhas começando
com # com '#' serão ignoradas, e uma mensagem vazia aborta o commit.
# On branch master #
Sua branch está atualizada com 'origin/master'. # # Mudanças a
serem confirmadas: # novo arquivo: README # modificado:
CONTRIBUTING.md



#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Você pode ver que a mensagem de confirmação padrão contém a saída mais recente do comando git status comentada e uma linha vazia no topo. Você pode remover esses comentários e digitar seu commit

Machine Translated by Google

mensagem, ou você pode deixá-los lá para ajudá-lo a lembrar o que está cometendo.

ÿ

Para um lembrete ainda mais explícito do que você modificou, você pode passar a opção -v para git commit. Fazer isso também coloca o diff de sua alteração no editor para que você possa ver exatamente quais alterações está confirmando.

Quando você sai do editor, o Git cria seu commit com aquela mensagem de commit (com os comentários e diff removidos).

Alternativamente, você pode digitar sua mensagem de commit em linha com o comando commit , especificando-o após um sinalizador -m , assim:

```
$ git commit -m "Story 182: fix benchmarks for speed" [master
463dc4f] Story 182: fix benchmarks for speed 2 arquivos alterados,
2 inserções(+) modo de criação 100644 README
```

Agora você criou seu primeiro commit! Você pode ver que o commit forneceu algumas saídas sobre si mesmo: em qual branch você fez o commit (mestre), qual soma de verificação SHA-1 o commit tem (463dc4f), quantos arquivos foram alterados e estatísticas sobre as linhas adicionadas e removidas no comprometer-se.

Lembre-se de que o commit registra o instantâneo que você configurou em sua área de preparação. Qualquer coisa que você não encenou ainda está lá modificada; você pode fazer outro commit para adicioná-lo ao seu histórico. Toda vez que você executa uma confirmação, está gravando um instantâneo do seu projeto que pode ser revertido ou comparado posteriormente.

Pular a área de preparação

Embora possa ser incrivelmente útil para criar commits exatamente como você deseja, a área de preparação às vezes é um pouco mais complexa do que você precisa em seu fluxo de trabalho. Se você quiser pular a área de preparação, o Git fornece um atalho simples. Adicionar a opção -a ao comando git commit faz com que o Git organize automaticamente todos os arquivos que já estão rastreados antes de fazer o commit, permitindo que você pule a parte git add :

```
$ git status On
branch master Sua
branch está atualizada com 'origin/master'.
Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido)
(use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md


nenhuma alteração adicionada ao commit (use "git add" e/ou "git commit -a") $
git commit -a -m 'Adicionar novos benchmarks' [master 83e38c7] Adicionar novos
benchmarks
```

Machine Translated by Google

```
1 arquivo alterado, 5 inserções (+), 0 exclusões (-)
```

Observe como você não precisa executar git add no arquivo CONTRIBUTING.md neste caso antes de confirmar. Isso ocorre porque o sinalizador -a inclui todos os arquivos alterados. Isso é conveniente, mas tenha cuidado; às vezes, esse sinalizador fará com que você inclua alterações indesejadas.

Removendo arquivos

Para remover um arquivo do Git, você deve removê-lo de seus arquivos rastreados (mais precisamente, removê-lo de sua área de teste) e então confirmar. O comando git rm faz isso e também remove o arquivo do seu diretório de trabalho para que você não o veja como um arquivo não rastreado na próxima vez.

Se você simplesmente remover o arquivo do seu diretório de trabalho, ele aparecerá na área "Changes not staged for commit" (isto é, *não testado)* da saída do git status :

```
$ rm PROJECTS.md
$ git status On branch
master Sua branch
está atualizada com 'origin/master'.
Alterações não testadas para confirmação:
(use "git add/rm <arquivo>..." para atualizar o que será confirmado) (use "git checkout --
<arquivo>..." para descartar alterações no diretório de trabalho)

        excluído:        PROJETOS.md

nenhuma alteração adicionada ao commit (use "git add" e/ou "git commit -a")
```

Então, se você rodar git rm, ele prepara a remoção do arquivo:

```
$ git rm PROJETOS.md rm
'PROJETOS.md'
$ git status On
branch master Sua
branch está atualizada com 'origin/master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

excluído:        PROJETOS.md
```

Na próxima vez que você confirmar, o arquivo desaparecerá e não será mais rastreado. Se você modificou o arquivo ou já o adicionou à área de staging, deve forçar a remoção com a opção -f . Este é um recurso de segurança para evitar a remoção acidental de dados que ainda não foram gravados em um instantâneo e que não podem ser recuperados do Git.

Outra coisa útil que você pode querer fazer é manter o arquivo em sua árvore de trabalho, mas removê-lo de sua área de teste. Em outras palavras, você pode querer manter o arquivo em seu disco rígido, mas não permitir que o Git o rastreie mais. Isso é particularmente útil se você esqueceu de adicionar algo ao seu .gitignore

Machine Translated by Google

arquivo e o preparou acidentalmente, como um grande arquivo de log ou um monte de arquivos .a compilados. Para fazer isso, use a opção --cached :

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

Você pode passar arquivos, diretórios e padrões de arquivo-glob para o comando git rm . Isso significa que você pode fazer coisas como:

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

Observe a barra invertida (\) na frente do *. Isso é necessário porque o Git faz sua própria expansão de nome de arquivo além da expansão de nome de arquivo do seu shell. Este comando remove todos os arquivos que possuem a extensão .log no diretório log/ . Ou, você pode fazer algo assim:

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

Este comando remove todos os arquivos cujos nomes terminam com ~.

Movendo arquivos

Ao contrário de muitos outros VCSs, o Git não rastreia explicitamente o movimento do arquivo. Se você renomear um arquivo no Git, nenhum metadado será armazenado no Git informando que você renomeou o arquivo. No entanto, o Git é muito inteligente em descobrir isso depois do fato — lidaremos com a detecção do movimento do arquivo um pouco mais tarde.

Portanto, é um pouco confuso que o Git tenha um comando mv . Se você deseja renomear um arquivo no Git, pode executar algo como:

```
$ git mv arquivo_de arquivo_para
```

e funciona bem. Na verdade, se você executar algo assim e observar o status, verá que o Git o considera um arquivo renomeado:

```
$ git mv README.md README $
git status On branch master Sua
branch está atualizada com 'origin/
master'.
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

renomeado:          README.md -> README
```

No entanto, isso é equivalente a executar algo assim:

```
$ mv README.md README
```

Machine Translated by Google

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

O Git descobre que é uma renomeação implícita, então não importa se você renomeia um arquivo dessa forma ou com o comando mv . A única diferença real é que git mv é um comando em vez de três — é uma função de conveniência. Mais importante, você pode usar qualquer ferramenta que desejar para renomear um arquivo e endereçar o add/rm mais tarde, antes de confirmar.

Visualizando o histórico de commits

Depois de criar vários commits, ou se você tiver clonado um repositório com um histórico de commits existente, provavelmente desejará olhar para trás para ver o que aconteceu. A ferramenta mais básica e poderosa para fazer isso é o comando git log .

Esses exemplos usam um projeto muito simples chamado “simplegit”. Para obter o projeto, execute:

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

Ao executar o git log neste projeto, você deve obter uma saída semelhante a esta:

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

Alterar número da versão

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Autor: Scott Chacon <schacon@gee-mail.com> Data:
Sáb 15 de março 16:40:33 2008 -0700

Remover teste desnecessário

cometer a11bef06a3f659402fe7563abf99ad00de2209e6
Autor: Scott Chacon <schacon@gee-mail.com>
Data: Sáb 15 de março 10:31:28 2008 -0700

Confirmação inicial
```

Por padrão, sem argumentos, git log lista os commits feitos naquele repositório em ordem cronológica inversa; ou seja, os commits mais recentes aparecem primeiro. Como você pode ver, este comando lista cada commit com sua soma de verificação SHA-1, o nome e e-mail do autor, a data de gravação e a mensagem do commit.

Um grande número e variedade de opções para o comando git log estão disponíveis para mostrar exatamente o que você está procurando. Aqui, mostraremos alguns dos mais populares.

Machine Translated by Google

Uma das opções mais úteis é -p ou --patch, que mostra a diferença (a saída do *patch)* introduzida em cada commit. Você também pode limitar o número de entradas de log exibidas, como usar -2 para mostrar apenas as duas últimas entradas.

```
$ git log -p -2 commit
ca82a6dff817ec66f44342007202690a93763949 Autor: Scott Chacon
<schacon@gee-mail.com> Data: Seg 17 de março 21:52:11 2008 -0700



Alterar número da versão


diff --git a/Rakefile b/Rakefile index
a874b73..8f94139 100644 --- a/Rakefile +++ b/
Rakefile @@ -5,7 +5,7 @@ require 'rake/
gempackagetask' spec = Gem: :Especificação.novo
do |s|

        s.platform = Gem::Platform::RUBY = "simplegit"
        s.name          s.version = "0.1.0" s.version =
-       "0.1.1" s.author
+
                        = "Scott Chacon"
        e-mail          = "schacon@gee-mail.com"
        s.resumo =              "Uma joia simples para usar Git em código Ruby."

cometer 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Autor: Scott Chacon <schacon@gee-mail.com> Data: Sáb, 15
de março 16:40:33 2008 -0700


Remover teste desnecessário


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 @@
classe SimpleGit end




fim
-
-if $0 == __FILE__ - git =
SimpleGit.new - coloca git.show
-end
```

Esta opção exibe as mesmas informações, mas com uma diferença logo após cada entrada. Isso é muito útil para revisão de código ou para navegar rapidamente pelo que aconteceu durante uma série de commits que um colaborador adicionou. Você também pode usar uma série de opções de resumo com git log. Por exemplo, se você quiser ver algumas estatísticas abreviadas para cada commit, você pode usar a opção --stat :

Machine Translated by Google

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Autor: Scott Chacon <schacon@gee-mail.com>
Data: Seg, 17 de março 21:52:11 2008 -0700

    Alterar número da versão

 Rakefile | 2 +- 1
 arquivo alterado, 1 inserção(+), 1 exclusão(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Autor: Scott Chacon <schacon@gee-mail.com> Data:
Sáb 15 de março 16:40:33 2008 -0700

    Remover teste desnecessário

 lib/simplegit.rb | 5 ----- 1 arquivo
 alterado, 5 exclusões (-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Autor: Scott Chacon <schacon@gee-mail.com> Data:
Sáb 15 de março 10:31:28 2008 -0700

    Confirmação inicial

 README Rakefile | 23 +++++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 arquivos alterados, 54 inserções(+)
```

Como você pode ver, a opção --stat imprime abaixo de cada entrada de confirmação uma lista de arquivos modificados, quantos arquivos foram alterados e quantas linhas nesses arquivos foram adicionadas e removidas. Ele também coloca um resumo das informações no final.

Outra opção realmente útil é --pretty. Esta opção altera a saída de log para formatos diferentes do padrão. Alguns valores de opção predefinidos estão disponíveis para você usar. O valor de uma linha para esta opção imprime cada commit em uma única linha, o que é útil se você estiver olhando para muitos commits. Além disso, os valores short, full e fuller mostram a saída aproximadamente no mesmo formato, mas com menos ou mais informações, respectivamente:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 Alterar número da
versão 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remover teste
desnecessário a11bef06a3f659402fe7563abf99ad060de Commit inicial2209e
```

O valor de opção mais interessante é format, que permite especificar seu próprio formato de saída de log. Isso é especialmente útil quando você está gerando saída para análise de máquina — como você especifica o formato explicitamente, sabe que não mudará com as atualizações do Git:

Machine Translated by Google

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 anos atrás: Alterar número da versão
085bb3b - Scott Chacon, 6 anos atrás: Remover teste desnecessário
a11bef0 - Scott Chacon , há 6 anos : Compromisso inicial
```

Especificadores úteis para git log --pretty=format lista alguns dos especificadores mais úteis que o formato aceita.

Tabela 1. Especificadores úteis para git log --pretty=format

especificador	Descrição da Saída
%H	Commit hash
%h	Hash de commit abreviado
%T	hash de árvore
%t	hash de árvore abreviado
%P	hashes pai
%p	Hashes pai abreviados
%um	Nome do autor
%mas	e-mail do autor
%de Anúncios	Data do autor (o formato respeita a opção --date=)
%Com	Data do autor, parente
%cn	Nome do committer
%Esse	E-mail do remetente
%cd	Data do commiter
%cr	Data do committer, parente
%s	Assunto

Você pode estar se perguntando qual é a diferença entre *autor* e *committer.* O autor é a pessoa que originalmente escreveu o trabalho, enquanto o committer é a pessoa que aplicou o trabalho pela última vez. Portanto, se você enviar um patch para um projeto e um dos membros principais aplicar o patch, ambos receberão o crédito — você como autor e o membro principal como o committer. Abordaremos essa distinção um pouco mais em Distributed Git.

Os valores das opções oneline e format são particularmente úteis com outra opção de log chamada --graph.

Esta opção adiciona um pequeno gráfico ASCII mostrando seu histórico de branch e merge:

```
$ git log --pretty=format:"%h %s" --graph *
2d3acf9 Ignorar erros de SIGCHLD na armadilha
* 5e3ee11 Mesclar ramificação 'mestre' de git://github.com/dustin/grit |\ |
 * 420eac9 Adicionar método para obter a ramificação atual * | 30e367c
 Código de tempo limite e testes
```

Machine Translated by Google

```
* | 5a09431 Adicionar proteção de tempo limite
ao grit * | e1193f8 Suporte para cabeçalhos com barras
|/ * d6016bc Requer tempo para xmlschema * 11d191e
Mesclar branch 'defunkt' no local
```

Esse tipo de saída se tornará mais interessante à medida que passarmos pelas ramificações e mesclagens no próximo capítulo.

Essas são apenas algumas opções simples de formatação de saída para o git log — existem muitas mais. As opções comuns para git log listam as opções que cobrimos até agora, bem como algumas outras opções de formatação comuns que podem ser úteis, junto com como elas alteram a saída do comando log.

Tabela 2. Opções comuns para git log

Opção	Descrição
-p	Mostre o patch introduzido com cada confirmação.
--Estado	Mostrar estatísticas para arquivos modificados em cada confirmação.
--shortstat	Exibe apenas a linha alterada/inserções/exclusões do comando --stat.
--name-only	Mostra a lista de arquivos modificados após as informações do commit.
--nome-status	Mostre também a lista de arquivos afetados com informações adicionadas/modificadas/excluídas.
--abbrev-commit	Mostra apenas os primeiros caracteres da soma de verificação SHA-1 em vez de todos os 40.
--data-relativa	Exiba a data em um formato relativo (por exemplo, “2 semanas atrás”) em vez de usar o formato de data completo.
--gráfico	Exiba um gráfico ASCII da ramificação e do histórico de mesclagem ao lado da saída do log.
--bonito	Mostrar confirmações em um formato alternativo. Os valores das opções incluem uma linha, curto, completo, mais completo e formato (onde você especifica seu próprio formato).
--on-line	Abreviação de --pretty=oneline --abbrev-commit usado em conjunto.

Limitando saída de log

Além das opções de formatação de saída, o git log usa várias opções úteis de limitação; ou seja, opções que permitem mostrar apenas um subconjunto de commits. Você já viu uma dessas opções — a opção -2 , que exibe apenas os dois últimos commits. Na verdade, você pode fazer -<n>, onde n é qualquer número inteiro para mostrar os últimos n commits. Na realidade, é improvável que você use isso com frequência, porque o Git, por padrão, canaliza toda a saída por meio de um pager para que você veja apenas uma página de saída de log por vez.

No entanto, as opções de limitação de tempo, como --since e --until, são muito úteis. Por exemplo, este comando obtém a lista de commits feitos nas últimas duas semanas:

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

Este comando funciona com vários formatos — você pode especificar uma data específica como "2008-01-15" ou uma data relativa como "2 anos 1 dia 3 minutos atrás".

Machine Translated by Google

Você também pode filtrar a lista para confirmações que correspondem a alguns critérios de pesquisa. A opção --author permite filtrar um autor específico, e a opção --grep permite pesquisar palavras-chave nas mensagens de confirmação.

ÿ

Você pode especificar mais de uma instância dos critérios de pesquisa --author e --grep , o que limitará a saída do commit a commits que correspondam *a qualquer* um dos padrões --author e --grep ; no entanto, adicionar a opção --all-match limita ainda mais a saída apenas para os commits que correspondem *a todos os* padrões --grep .

Outro filtro realmente útil é a opção -S (coloquialmente chamada de opção "picareta" do Git), que pega uma string e mostra apenas os commits que alteraram o número de ocorrências dessa string. Por exemplo, se você quiser encontrar o último commit que adicionou ou removeu uma referência a uma função específica, você pode chamar:

```
$ git log -S nome_da_função
```

A última opção realmente útil para passar para o git log como um filtro é um caminho. Se você especificar um diretório ou nome de arquivo, poderá limitar a saída do log a confirmações que introduziram uma alteração nesses arquivos. Esta é sempre a última opção e geralmente é precedida por dois traços (--) para separar os caminhos das opções:

```
$ git log -- caminho/para/arquivo
```

Em Opções para limitar a saída do git log, listaremos essas e algumas outras opções comuns para sua referência.

Tabela 3. Opções para limitar a saída do git log

Opção	Descrição
-<n>	Mostrar apenas os últimos n commits
--desde, --depois	Limite as confirmações àquelas feitas após a data especificada.
--até, --antes	Limite as confirmações àquelas feitas antes da data especificada.
--autor	Mostra apenas os commits nos quais a entrada do autor corresponde à string especificada.
--committer	Mostra apenas os commits nos quais a entrada do committer corresponde à string especificada.
--pegada	Mostrar apenas commits com uma mensagem de commit contendo a string
-S	Mostrar apenas commits adicionando ou removendo código correspondente à string

Por exemplo, se você quiser ver quais commits que modificam arquivos de teste no histórico do código-fonte do Git foram commitados por Junio Hamano no mês de outubro de 2008 e não são commits de mesclagem, você pode executar algo assim:

Machine Translated by Google

```
$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \ --
before="2008-11-01" --no- merges -- t/ 5610e3b - Corrige falha do caso de teste
quando atributos estendidos estão em uso acd3b9e - Aprimorou
hold_lock_file_for_{update,append}() API f563754 - demonstra quebra de checkout
desanexado com link simbólico HEAD d1a43f2 - reset --hard/read-tree - -reset -u:
remove novos caminhos não mesclados 51a94af - Corrige "checkout --track -b
newbranch" no HEAD separado b0ad11e - pull: permite "git pull origin $something:
$current_branch" em um branch não nascido
```

Dos quase 40.000 commits no histórico do código-fonte do Git, este comando mostra os 6 que correspondem a esses critérios.

ÿ

Impedindo a exibição de commits de mesclagem

Dependendo do fluxo de trabalho usado em seu repositório, é possível que uma porcentagem considerável das confirmações em seu histórico de log sejam apenas confirmações de mesclagem, que normalmente não são muito informativas. Para evitar que a exibição de confirmações de mesclagem sobrecarregue seu histórico de log, basta adicionar a opção de log --no-merges.

Desfazendo Coisas

Em qualquer estágio, você pode querer desfazer algo. Aqui, revisaremos algumas ferramentas básicas para desfazer as alterações feitas. Tenha cuidado, porque nem sempre é possível desfazer alguns desses desfazeres. Esta é uma das poucas áreas no Git onde você pode perder algum trabalho se fizer errado.

Um dos desfazeres comuns ocorre quando você faz um commit muito cedo e possivelmente se esquece de adicionar alguns arquivos, ou você confunde sua mensagem de commit. Se você quiser refazer esse commit, faça as alterações adicionais que você esqueceu, teste-as e confirme novamente usando a opção --amend :

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

Este comando pega sua área de teste e a usa para o commit. Se você não fez alterações desde seu último commit (por exemplo, você executou este comando imediatamente após o commit anterior), então seu instantâneo parecerá exatamente o mesmo e tudo o que você mudará será sua mensagem de commit.

O mesmo editor de mensagem de confirmação é acionado, mas já contém a mensagem de sua confirmação anterior. Você pode editar a mensagem como sempre, mas ela sobrescreve seu commit anterior.

Por exemplo, se você fizer um commit e depois perceber que esqueceu de preparar as alterações em um arquivo que deseja adicionar a esse commit, você pode fazer algo assim:

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

Machine Translated by Google

Você acaba com um único commit — o segundo commit substitui os resultados do primeiro.

ÿ

É importante entender que quando você está corrigindo seu último commit, você não está tanto consertando, mas *substituindo* -o inteiramente por um novo commit melhorado que tira o commit antigo do caminho e coloca o novo commit em seu lugar.
Efetivamente, é como se o commit anterior nunca tivesse acontecido e não aparecesse no histórico do seu repositório.

O valor óbvio de corrigir commits é fazer pequenas melhorias em seu último commit, sem sobrecarregar o histórico do seu repositório com mensagens de commit do tipo "Oops, esqueci de adicionar um arquivo" ou "Droga, corrigindo um erro de digitação no último commit".

ÿ

Apenas corrija os commits que ainda são locais e não foram enviados para algum lugar.
Corrigir commits enviados anteriormente e forçar o push do branch causará problemas para seus colaboradores. Para saber mais sobre o que acontece quando você faz isso e como se recuperar se você estiver recebendo, leia The Perils of Rebasing.

Desinstalando um arquivo preparado

As próximas duas seções demonstram como trabalhar com sua área de preparação e mudanças no diretório de trabalho. A parte boa é que o comando que você usa para determinar o estado dessas duas áreas também lembra como desfazer alterações nelas. Por exemplo, digamos que você alterou dois arquivos e deseja confirmá-los como duas alterações separadas, mas digitou acidentalmente git add * e preparou os dois. Como você pode desencenar um dos dois? O comando git status lembra você:

```
$ git add * $
git status
No mestre da filial
Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

renomeado:          README.md -> README
modificado: CONTRIBUTING.md
```

Logo abaixo do texto "Mudanças a serem confirmadas", ele diz use git reset HEAD <arquivo>… para remover o estágio. Então, vamos usar esse conselho para remover o arquivo CONTRIBUTING.md :

```
$ git reset HEAD CONTRIBUTING.md
Mudanças sem preparação após reset: M
CONTRIBUTING.md $ git status On branch
master

Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

renomeado:          README.md -> README
```

Machine Translated by Google

```
Alterações não testadas para confirmação:
(use "git add <arquivo>..." para atualizar o que será submetido) (use "git checkout
-- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md
```

O comando é um pouco estranho, mas funciona. O arquivo CONTRIBUTING.md foi modificado, mas novamente removido.

É verdade que git reset pode ser um comando perigoso, especialmente se você fornecer o sinalizador --hard . No entanto, no cenário descrito acima, o arquivo em seu diretório de trabalho não é tocado, por isso é relativamente seguro.

Por enquanto, essa invocação mágica é tudo o que você precisa saber sobre o comando git reset . Entraremos em muito mais detalhes sobre o que o reset faz e como dominá-lo para fazer coisas realmente interessantes em Reset Demystified.

Desmodificando um arquivo modificado

E se você perceber que não deseja manter suas alterações no arquivo CONTRIBUTING.md ? Como você pode facilmente desmodificá-lo - revertê-lo para o que parecia quando você fez o commit (ou inicialmente clonado, ou como você o colocou em seu diretório de trabalho)? Felizmente, git status diz a você como fazer isso também. No último exemplo de saída, a área não selecionada tem esta aparência:

```
Alterações não testadas para confirmação:
(use "git add <arquivo>..." para atualizar o que será submetido) (use "git checkout
-- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: CONTRIBUTING.md
```

Ele informa explicitamente como descartar as alterações feitas. Vamos fazer o que diz:

```
$ git checkout -- CONTRIBUTING.md $ git
status No branch master Mudanças a serem
confirmadas: (use "git reset HEAD <arquivo>..."
para retirar o estágio)


renomeado:          README.md -> README
```

Você pode ver que as alterações foram revertidas.

É importante entender que git checkout -- <file> é um comando perigoso.
Todas as alterações locais feitas nesse arquivo desapareceram — o Git apenas substituiu esse arquivo pela última versão testada ou confirmada. Nunca use este comando, a menos que você saiba absolutamente que não deseja essas alterações locais não salvas.

Machine Translated by Google

Se você gostaria de manter as alterações feitas nesse arquivo, mas ainda precisa tirá-lo do caminho por enquanto, examinaremos o armazenamento e ramificação no Git Branching; essas são geralmente as melhores maneiras de ir.

Lembre-se, qualquer coisa que seja *confirmada* no Git quase sempre pode ser recuperada. Mesmo os commits que estavam em branches que foram excluídos ou os commits que foram sobrescritos com um --amend commit podem ser recuperado (consulte Recuperação de dados para recuperação de dados). No entanto, qualquer coisa que você perca e nunca tenha sido comprometida provavelmente nunca mais será vista.

Desfazendo coisas com git restore

O Git versão 2.23.0 introduziu um novo comando: git restore. É basicamente uma alternativa ao git reset que acabamos de abordar. A partir da versão 2.23.0 do Git, o Git usará git restore em vez de git reset para muitas operações de desfazer.

Vamos refazer nossos passos e desfazer as coisas com git restore em vez de git reset.

Removendo um arquivo preparado com git restore

As próximas duas seções demonstram como trabalhar com sua área de preparação e mudanças no diretório de trabalho com git restore. A parte boa é que o comando que você usa para determinar o estado dessas duas áreas também lembra como desfazer alterações nelas. Por exemplo, digamos que você alterou dois arquivos e deseja confirmá-los como duas alterações separadas, mas digitou acidentalmente git add * e preparou os dois. Como você pode desencenar um dos dois? O comando git status lembra você:

```
$ git add * $
git status No
branch master
Alterações a serem confirmadas:
(use "git restore --staged <arquivo>..." para remover o estágio)
modificado: CONTRIBUTING.md renomeado:
                    README.md -> README
```

Logo abaixo do texto “Changes to be commited”, ele diz use git restore --staged <file>… to unstage. Então, vamos usar esse conselho para remover o arquivo CONTRIBUTING.md :

```
$ git restore --staged CONTRIBUTING.md $ git
status No branch master

Alterações a serem confirmadas:
(use "git restore --staged <arquivo>..." para remover o estágio)
renomeado: README.md -> README

Alterações não preparadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido) (use
"git restore <arquivo>..." para descartar alterações no diretório de trabalho) modificado:
CONTRIBUTING.md
```

Machine Translated by Google

O arquivo CONTRIBUTING.md foi modificado, mas novamente removido.

Desmodificando um arquivo modificado com git restore

E se você perceber que não deseja manter suas alterações no arquivo CONTRIBUTING.md ? Como você pode facilmente desmodificá-lo - revertê-lo para o que parecia quando você fez o commit (ou inicialmente clonado, ou como você o colocou em seu diretório de trabalho)? Felizmente, git status diz a você como fazer isso também. No último exemplo de saída, a área não selecionada tem esta aparência:

```
Alterações não preparadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será
submetido) (use "git restore <arquivo>..." para descartar alterações no diretório de
trabalho) modificado: CONTRIBUTING.md
```

Ele informa explicitamente como descartar as alterações feitas. Vamos fazer o que diz:

```
$ git restore CONTRIBUTING.md $
git status No branch master
Mudanças a serem confirmadas:
(use "git restore --staged
<arquivo>..." para remover o estágio) renomeado: README.md
-> README
```

ÿ

É importante entender que git restore <file> é um comando perigoso. Todas as alterações locais feitas nesse arquivo desapareceram — o Git apenas substituiu esse arquivo pela última versão testada ou confirmada. Nunca use este comando, a menos que você saiba absolutamente que não deseja essas alterações locais não salvas.

Trabalhando com Controles Remotos

Para poder colaborar em qualquer projeto Git, você precisa saber como gerenciar seus repositórios remotos. Os repositórios remotos são versões do seu projeto que estão hospedadas na Internet ou na rede em algum lugar. Você pode ter vários deles, cada um dos quais geralmente é somente leitura ou leitura/gravação para você. A colaboração com outras pessoas envolve o gerenciamento desses repositórios remotos e o envio e recebimento de dados de e para eles quando você precisa compartilhar o trabalho. Gerenciar repositórios remotos inclui saber como adicionar repositórios remotos, remover remotos que não são mais válidos, gerenciar vários ramos remotos e defini-los como rastreados ou não, e muito mais. Nesta seção, abordaremos algumas dessas habilidades de gerenciamento remoto.

ÿ

Os repositórios remotos podem estar em sua máquina local.

É perfeitamente possível que você esteja trabalhando com um repositório “remoto” que esteja, de fato, no mesmo host que você. A palavra “remoto” não implica necessariamente que o repositório esteja em outro lugar na rede ou na Internet, apenas que está em outro lugar. Trabalhar com tal repositório remoto ainda envolveria todas as operações padrão de empurrar, puxar e buscar como em qualquer outro remoto.

Machine Translated by Google

Mostrando seus controles remotos

Para ver quais servidores remotos você configurou, você pode executar o comando git remote . Ele lista os nomes curtos de cada identificador remoto que você especificou. Se você clonou seu repositório, você deve pelo menos ver origin — esse é o nome padrão que o Git dá ao servidor do qual você clonou:

```
$ git clone https://github.com/schacon/ticgit Cloning
into 'ticgit'... remote: Reutilizando o pacote existente:
1857, pronto. remoto: Total 1857 (delta 0), reutilizado
0 (delta 0)
Recebimento de objetos: 100% (1857/1857), 374,35 KiB | 268,00 KiB/s, concluído.
Resolvendo deltas: 100% (772/772), concluído.
Verificando a conectividade...
concluído. $ cd ticgit $ git origem
remota
```

Você também pode especificar -v, que mostra as URLs que o Git armazenou para o nome curto a ser usado ao ler e gravar naquele controle remoto:

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

Se você tiver mais de um controle remoto, o comando listará todos eles. Por exemplo, um repositório com vários controles remotos para trabalhar com vários colaboradores pode se parecer com isso.

```
$ cd grit $
git remote -v
bakkdoor https://github.com/bakkdoor/grit (buscar) bakkdoor
https://github.com/bakkdoor/grit (push) https://github.com/
github.com/chris45/grit (buscar) chris45 https://github.com/
chris45/grit (push) defunkt https://github.com/
defunkt/grit (buscar) defunkt https://github.com/defunkt/grit
(push) git://github.com/koke/grit.git (buscar)
koke koke git://github.com/koke/grit.git (push) origem
git@github.com:mojombo/grit.git (buscar) origem git@
github.com:mojombo/grit.git (push)
```

Isso significa que podemos obter contribuições de qualquer um desses usuários com bastante facilidade. Além disso, podemos ter permissão para enviar para um ou mais deles, embora não possamos dizer isso aqui.

Observe que esses controles remotos usam uma variedade de protocolos; abordaremos mais sobre isso em Obtendo o Git em um Servidor.

Machine Translated by Google

Adicionando repositórios remotos

Mencionamos e demos algumas demonstrações de como o comando git clone adiciona implicitamente o origin remote para você. Veja como adicionar um novo controle remoto explicitamente. Para adicionar um novo repositório Git remoto como um nome curto que você pode referenciar facilmente, execute git remote add <shortname> <url>:

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

Agora você pode usar a string pb na linha de comando no lugar de toda a URL. Por exemplo, se você quiser buscar todas as informações que Paul tem, mas que você ainda não tem em seu repositório, você pode executar git fetch pb:

```
$ git fetch pb
remote: Contando objetos: 43, concluído.
remote: Comprimindo objetos: 100% (36/36), concluído. remoto:
Total 43 (delta 10), reutilizado 31 (delta 5)
Desempacotando objetos: 100% (43/43), concluído.
De https://github.com/paulboone/ticgit * [nova
ramificação] -> pb/master * [nova ramificação] mestre -> pb/ticgit
ticgit
```

O branch master de Paul agora está acessível localmente como pb/master — você pode fazer merge dele em um de seus branches, ou você pode fazer check-out de um branch local naquele ponto se quiser inspecioná-lo. Veremos o que são branches e como usá-los com muito mais detalhes no Git Branching.

Buscando e puxando de seus controles remotos

Como você acabou de ver, para obter dados de seus projetos remotos, você pode executar:

```
$ git busca <remoto>
```

O comando vai para esse projeto remoto e obtém todos os dados desse projeto remoto que você ainda não possui. Depois de fazer isso, você deve ter referências a todas as ramificações desse controle remoto, nas quais você pode mesclar ou inspecionar a qualquer momento.

Se você clonar um repositório, o comando adiciona automaticamente esse repositório remoto com o nome “origem”. Portanto, git fetch origin busca qualquer novo trabalho que tenha sido enviado para esse servidor desde que você o clonou (ou o buscou pela última vez). É importante observar que o comando git fetch apenas baixa os dados para seu repositório local — ele não os mescla automaticamente com nenhum de seus trabalhos ou modifica o que você está trabalhando atualmente. Você tem que mesclá-lo manualmente em seu trabalho quando

Machine Translated by Google

Você está pronto.

Se sua ramificação atual estiver configurada para rastrear uma ramificação remota (consulte a próxima seção e Git Branching para obter mais informações), você pode usar o comando git pull para buscar automaticamente e, em seguida, mesclar essa ramificação remota em sua ramificação atual. Esse pode ser um fluxo de trabalho mais fácil ou confortável para você; e, por padrão, o comando git clone configura automaticamente sua ramificação mestre local para rastrear a ramificação mestre remota (ou qualquer que seja a ramificação padrão) no servidor do qual você clonou.
A execução de git pull geralmente busca dados do servidor do qual você originalmente clonou e tenta automaticamente mesclá-los no código em que você está trabalhando no momento.

ÿ

A partir da versão 2.27 do git, o git pull emitirá um aviso se a variável pull.rebase não estiver definida. O Git continuará avisando até que você defina a variável.

Se você deseja o comportamento padrão do git (avanço rápido se possível, senão crie um merge commit): git config --global pull.rebase "false"

Se você deseja rebase ao puxar: git config --global pull.rebase "true"

Empurrando para seus controles remotos

Quando você tem seu projeto em um ponto que deseja compartilhar, precisa empurrá-lo rio acima. O comando para isso é simples: git push <remote> <branch>. Se você quiser enviar seu branch master para o servidor de origem (novamente, a clonagem geralmente configura esses dois nomes para você automaticamente), então você pode executar isso para enviar quaisquer commits que você fez de volta para o servidor:

```
$ git push mestre de origem
```

Este comando funciona apenas se você clonou de um servidor ao qual você tem acesso de gravação e se ninguém o enviou nesse meio tempo. Se você e outra pessoa clonarem ao mesmo tempo e eles fizerem push upstream e você fizer push upstream, seu push será rejeitado com razão. Você terá que buscar o trabalho deles primeiro e incorporá-lo ao seu antes de poder empurrar. Consulte Git Branching para obter informações mais detalhadas sobre como enviar para servidores remotos.

Inspecionando um controle remoto

Se quiser ver mais informações sobre um determinado controle remoto, você pode usar o comando git remote show <remote> . Se você executar este comando com um nome curto específico, como origin, obterá algo como isto:

```
$ git remote show origin
* remote origin Fetch URL:
https://github.com/schacon/ticgit Push URL: https://
github.com/schacon/ticgit HEAD branch: master

Ramos remotos:
rastreado master dev-branch rastreado

Ramificação local configurada para 'git pull':
```

Machine Translated by Google

```
mestre mescla com mestre remoto
Referência local configurada para 'git
push': mestre envia para mestre (atualizado)
```

Ele lista a URL para o repositório remoto, bem como as informações do branch de rastreamento. O comando informa que, se você estiver na ramificação master e executar git pull, ele mesclará automaticamente a ramificação master do controle remoto com a local depois de ter sido buscada. Ele também lista todas as referências remotas que ele baixou.

Esse é um exemplo simples que você provavelmente encontrará. No entanto, quando você estiver usando o Git com mais intensidade, poderá ver muito mais informações do git remote show:

```
$ git remote show origin
*
   remote origin URL:
https://github.com/my-org/complex-project Buscar
URL: https://github.com/my-org/complex-project Push URL:
https://github .com/my-org/complex-project HEAD branch:
master
Ramos remotos:
master dev-                    rastreado
branch                         rastreado
markdown-strip                 rastreado
issue-43 issue-45              novo (a próxima busca será armazenada em remotos/
refs/remotes/origin/           origem) novo (a próxima busca será armazenada em
issue-11                       remotos/origem) obsoleto (use 'git remote prune' para remover)
Branches locais configurados para 'git pull': dev-
branch merge com remote dev-branch merge
                com remote master master
Refs locais configuradas para 'git push':
data dev-branch) markdown-strip date)  empurra para o dev-branch          (até
master date)
                                       empurra para a faixa de remarcação  (até

                                       empurra para dominar                (até
```

Este comando mostra para qual ramificação é enviado automaticamente quando você executa git push em determinadas ramificações. Ele também mostra quais branches remotos no servidor você ainda não possui, quais branches remotos você tem que foram removidos do servidor e vários branches locais que podem se fundir automaticamente com seu branch de rastreamento remoto quando você executa o git puxar.

Renomeando e removendo controles remotos

Você pode executar git remote rename para alterar o nome abreviado de um controle remoto. Por exemplo, se você deseja renomear pb para paul, pode fazê-lo com git remote rename:

```
$ git remote renomear pb
paul $ git remote
```

Machine Translated by Google

```
origem
paulo
```

Vale a pena mencionar que isso também altera todos os nomes de filiais de rastreamento remoto. O que costumava ser referenciado em pb/master agora está em paul/master.

Se você deseja remover um controle remoto por algum motivo - você moveu o servidor ou não está mais usando um espelho específico, ou talvez um contribuidor não esteja mais contribuindo - você pode usar git remote remove ou git remote rm :

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

Depois de excluir a referência a um controle remoto dessa maneira, todas as ramificações de rastreamento remoto e definições de configuração associadas a esse controle remoto também serão excluídas.

marcação

Como a maioria dos VCSs, o Git tem a capacidade de marcar pontos específicos no histórico de um repositório como sendo importantes. Normalmente, as pessoas usam essa funcionalidade para marcar pontos de lançamento (v1.0, v2.0 e assim por diante). Nesta seção, você aprenderá como listar tags existentes, como criar e excluir tags e quais são os diferentes tipos de tags.

Listando suas tags

Listar as tags existentes no Git é simples. Basta digitar git tag (com -l ou --list opcional):

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

Este comando lista as tags em ordem alfabética; a ordem em que são exibidos não tem importância real.

Você também pode procurar tags que correspondam a um determinado padrão. O repositório de origem Git, por exemplo, contém mais de 500 tags. Se você estiver interessado apenas em ver a série 1.8.5, execute isto:

```
$ 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
```

Machine Translated by Google

```
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

ÿ

Os curingas de marca de listagem requerem a opção -l ou --list

Se você quiser apenas a lista inteira de tags, a execução do comando git tag assume implicitamente que você deseja uma listagem e fornece uma; o uso de -l ou --list neste caso é opcional.

Se, no entanto, você estiver fornecendo um padrão curinga para corresponder aos nomes das tags, o uso de -l ou --list é obrigatório.

Criando Marcas

O Git oferece suporte a dois tipos de tags: *leves* e *anotadas.*

Uma tag leve é muito parecida com um branch que não muda — é apenas um ponteiro para um commit específico.

Tags anotadas, no entanto, são armazenadas como objetos completos no banco de dados Git. Eles são verificados; conter o nome do tagger, e-mail e data; ter uma mensagem de marcação; e pode ser assinado e verificado com GNU Privacy Guard (GPG). Geralmente é recomendado que você crie tags anotadas para ter todas essas informações; mas se você quiser uma tag temporária ou por algum motivo não quiser manter as outras informações, as tags leves também estão disponíveis.

Marcas anotadas

Criar uma tag anotada no Git é simples. A maneira mais fácil é especificar -a ao executar o comando tag :

```
$ git tag -a v1.4 -m "minha versão 1.4" $ git
tag v0.1

v1.3
v1.4
```

O -m especifica uma mensagem de marcação, que é armazenada com a marca. Se você não especificar uma mensagem para uma tag anotada, o Git iniciará seu editor para que você possa digitá-la.

Você pode ver os dados da tag junto com o commit que foi marcado usando o comando git show :

```
$ git show v1.4
tag v1.4 Marcador:
Ben Straub <ben@straub.cc> Data:
sábado, 3 de maio 20:19:12 2014 -0700

minha versão 1.4
```

Machine Translated by Google

```
cometer ca82a6dff817ec66f44342007202690a93763949
Autor: Scott Chacon <schacon@gee-mail.com>
Data: Seg, 17 de março 21:52:11 2008 -0700

Alterar número da versão
```

Isso mostra as informações do tagger, a data em que o commit foi marcado e a mensagem de anotação antes de mostrar as informações do commit.

Etiquetas Leves

Outra maneira de marcar commits é com uma tag leve. Isso é basicamente a soma de verificação do commit armazenada em um arquivo — nenhuma outra informação é mantida. Para criar uma tag leve, não forneça nenhuma das opções -a, -s ou -m , apenas forneça um nome de tag:

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


v1.4
v1.4-lw
v1.5
```

Desta vez, se você executar git show na tag, não verá as informações extras da tag. O comando apenas mostra o commit:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Autor: Scott Chacon <schacon@gee-mail.com> Data:
Seg, 17 de março 21:52:11 2008 -0700

Alterar número da versão
```

Marcando mais tarde

Você também pode marcar commits depois de passar por eles. Suponha que seu histórico de commits seja assim:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
```

Machine Translated by Google

```
9fceb02d0ae598e95dc970b74767f19372d61af8 Atualizar rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Confirme o todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Atualizar leia-me
```

Agora, suponha que você esqueceu de marcar o projeto em v1.2, que estava no commit “Update rakefile”. Você pode adicioná-lo após o fato. Para marcar esse commit, você especifica a soma de verificação do commit (ou parte dele) no final do comando:

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

Você pode ver que marcou o commit:

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

$ git show v1.2 tag
v1.2 Marcador:
Scott Chacon <schacon@gee-mail.com> Data: segunda-
feira, 9 de fevereiro 15:32:16 2009 -0800

versão 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Autor: Magnus Chacon <mchacon@gee-mail.com> Data:
Dom 27 de abril 20:43:35 2008 -0700

Atualizar rakefile
 ...
```

Compartilhamento de tags

Por padrão, o comando git push não transfere tags para servidores remotos. Você terá que enviar explicitamente as tags para um servidor compartilhado depois de criá-las. Esse processo é como compartilhar ramificações remotas — você pode executar git push origin <tagname>.

```
$ git push origin v1.5
Contando objetos: 14, concluído.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (12/12), concluído.
Objetos de escrita: 100% (14/14), 2,05 KiB | 0 byte/s, pronto.
Total 14 (delta 3), reutilizado 0 (delta 0)
Para git@github.com:schacon/simplegit.git
```

Machine Translated by Google

```
* [nova etiqueta]          v1.5 -> v1.5
```

Se você tiver muitas tags que deseja enviar de uma só vez, também poderá usar a opção --tags para o comando git push . Isso transferirá todas as suas tags para o servidor remoto que ainda não estão lá.

```
$ git push origin --tags
Contando objetos: 1, concluído.
Objetos de escrita: 100% (1/1), 160 bytes | 0 byte/s, pronto.
Total 1 (delta 0), reutilizado 0 (delta 0)
Para git@github.com:schacon/simplegit.git *
[nova tag] * [nova tag]    v1.4 -> v1.4
                           v1.4-lw -> v1.4-lw
```

Agora, quando outra pessoa clona ou extrai de seu repositório, ela também obterá todas as suas tags.

ÿ

git push *empurra ambos os tipos de tags*

git push <remote> --tags enviará tags leves e anotadas. Atualmente, não há opção para enviar apenas tags leves, mas se você usar git push <remote> --follow-tags, apenas tags anotadas serão enviadas para o controle remoto.

Excluindo tags

Para excluir uma tag em seu repositório local, você pode usar git tag -d <tagname>. Por exemplo, poderíamos remover nossa tag leve acima da seguinte maneira:

```
$ git tag -d v1.4-lw Tag
excluída 'v1.4-lw' (era e7d5add)
```

Observe que isso não remove a tag de nenhum servidor remoto. Existem duas variações comuns para excluir uma tag de um servidor remoto.

A primeira variação é git push <remote> :refs/tags/<tagname>:

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

A maneira de interpretar o acima é lê-lo como o valor nulo antes que os dois pontos sejam enviados para o nome da tag remota, excluindo-o efetivamente.

A segunda (e mais intuitiva) maneira de excluir um tag remoto é com:

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

Machine Translated by Google

Verificando tags

Se você quiser visualizar as versões dos arquivos para os quais uma tag está apontando, você pode fazer um git checkout dessa tag, embora isso coloque seu repositório no estado “HEAD separado”, o que tem alguns efeitos colaterais ruins:

```
$ git checkout v2.0.0 Nota:
mudando para 'v2.0.0'.

Você está no estado 'cabeça separada'. Você pode olhar em volta, fazer alterações
experimentais e confirmá-las, e pode descartar quaisquer confirmações feitas nesse estado
sem afetar nenhuma ramificação realizando outro check-out.

Se você deseja criar uma nova ramificação para reter os commits que criou, pode fazê-lo
(agora ou mais tarde) usando -c com o comando switch. Exemplo:

git switch -c <nome da nova ramificação>

Ou desfaça esta operação com:

git switch -

Desative este aviso definindo a variável de configuração warning.detachedHead como false

HEAD está agora em 99ada87... Merge pull request #89 de schacon/appendix-final

$ git checkout v2.0-beta-0.1 A
posição HEAD anterior era 99ada87... Mesclar pull request #89 de schacon/apêndice final HEAD está
agora em df3f601... Adicionar atlas.json e imagem de capa
```

No estado “detached HEAD”, se você fizer alterações e depois criar um commit, a tag permanecerá a mesma, mas seu novo commit não pertencerá a nenhum branch e ficará inacessível, exceto pelo hash exato do commit. Portanto, se você precisar fazer alterações — digamos que está corrigindo um bug em uma versão mais antiga, por exemplo — geralmente desejará criar uma ramificação:

```
$ git checkout -b version2 v2.0.0 Alterado
para um novo branch 'version2'
```

Se você fizer isso e fizer um commit, sua ramificação da versão 2 será um pouco diferente da sua tag v2.0.0 , pois ela seguirá em frente com as novas alterações, portanto, tenha cuidado.

Git Aliases

Antes de passarmos para o próximo capítulo, queremos apresentar um recurso que pode tornar sua experiência com o Git mais simples, fácil e familiar: aliases. Para fins de clareza, não os usaremos em nenhum outro lugar neste livro, mas se você continuar a usar o Git com alguma regularidade, os aliases são algo que você deve conhecer.

Machine Translated by Google

O Git não infere automaticamente seu comando se você o digitar parcialmente. Se não quiser digitar todo o texto de cada um dos comandos do Git, você pode configurar facilmente um alias para cada comando usando git config. Aqui estão alguns exemplos que você pode querer configurar:

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

Isso significa que, por exemplo, ao invés de digitar git commit, basta digitar git ci. À medida que você continuar usando o Git, provavelmente também usará outros comandos com frequência; não hesite em criar novos aliases.

Essa técnica também pode ser muito útil na criação de comandos que você acha que deveriam existir. Por exemplo, para corrigir o problema de usabilidade encontrado ao desinstalar um arquivo, você pode adicionar seu próprio alias de desativação ao Git:

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

Isso torna os dois comandos a seguir equivalentes:

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

Isso parece um pouco mais claro. Também é comum adicionar um último comando, como este:

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

Dessa forma, você pode ver o último commit facilmente:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Autor: Josh Goebel <dreamer3@example.com> Data:
Ter 26 de agosto 19:48:51 2008 +0800

Teste para cabeçote atual

Assinado por: Scott Chacon <schacon@example.com>
```

Como você pode ver, o Git simplesmente substitui o novo comando por qualquer alias que você usar. No entanto, talvez você queira executar um comando externo, em vez de um subcomando do Git. Nesse caso, você inicia o comando com um ! personagem. Isso é útil se você escrever suas próprias ferramentas que funcionam com um repositório Git. Podemos demonstrar criando um alias git visual para executar o gitk:

Machine Translated by Google

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

Resumo

Neste ponto, você pode fazer todas as operações locais básicas do Git — criar ou clonar um repositório, fazer alterações, testar e confirmar essas alterações e visualizar o histórico de todas as alterações pelas quais o repositório passou. Em seguida, abordaremos o recurso matador do Git: seu modelo de ramificação.

Machine Translated by Google

Ramificação do Git

Quase todo VCS tem alguma forma de suporte de ramificação. Ramificar significa que você diverge da linha principal de desenvolvimento e continua a trabalhar sem interferir nessa linha principal. Em muitas ferramentas VCS, esse é um processo um tanto caro, geralmente exigindo que você crie uma nova cópia do diretório do código-fonte, o que pode levar muito tempo para projetos grandes.

Algumas pessoas se referem ao modelo de ramificação do Git como seu “recurso matador” e certamente diferencia o Git na comunidade VCS. Por que é tão especial? A maneira como o Git ramifica é incrivelmente leve, tornando as operações de ramificação quase instantâneas e alternando entre as ramificações geralmente com a mesma rapidez. Ao contrário de muitos outros VCSs, o Git incentiva fluxos de trabalho que se ramificam e se mesclam com frequência, até mesmo várias vezes ao dia. Compreender e dominar esse recurso oferece uma ferramenta poderosa e única e pode mudar totalmente a maneira como você se desenvolve.

Ramos em poucas palavras

Para realmente entender como o Git faz ramificações, precisamos dar um passo para trás e examinar como o Git armazena seus dados.

Como você deve se lembrar de O que é o Git?, o Git não armazena dados como uma série de conjuntos de alterações ou diferenças, mas sim como uma série de *instantâneos.*

Quando você faz uma confirmação, o Git armazena um objeto de confirmação que contém um ponteiro para o instantâneo do conteúdo que você testou. Este objeto também contém o nome e endereço de e-mail do autor, a mensagem que você digitou e ponteiros para o commit ou commits que vieram diretamente antes deste commit (seu pai ou pais): zero pais para o commit inicial, um pai para um commit normal e vários pais para uma confirmação que resulta de uma mesclagem de duas ou mais ramificações.

Para visualizar isso, vamos supor que você tenha um diretório contendo três arquivos, e que você faça o stage de todos eles e confirme. A preparação dos arquivos calcula uma soma de verificação para cada um (o hash SHA-1 que mencionamos em O que é Git?), armazena essa versão do arquivo no repositório Git (o Git se refere a eles como *blobs)* e adiciona essa soma de verificação à preparação área:

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

Quando você cria o commit executando git commit, o Git verifica a soma de cada subdiretório (neste caso, apenas o diretório raiz do projeto) e os armazena como um objeto de árvore no repositório Git. O Git então cria um objeto commit que tem os metadados e um ponteiro para a árvore raiz do projeto para que ele possa recriar esse instantâneo quando necessário.

Seu repositório Git agora contém cinco objetos: três *blobs* (cada um representando o conteúdo de um dos três arquivos), uma *árvore* que lista o conteúdo do diretório e especifica quais nomes de arquivo são armazenados como quais blobs e um *commit* com o ponteiro para aquela árvore raiz e todos os metadados do commit.

Machine Translated by Google

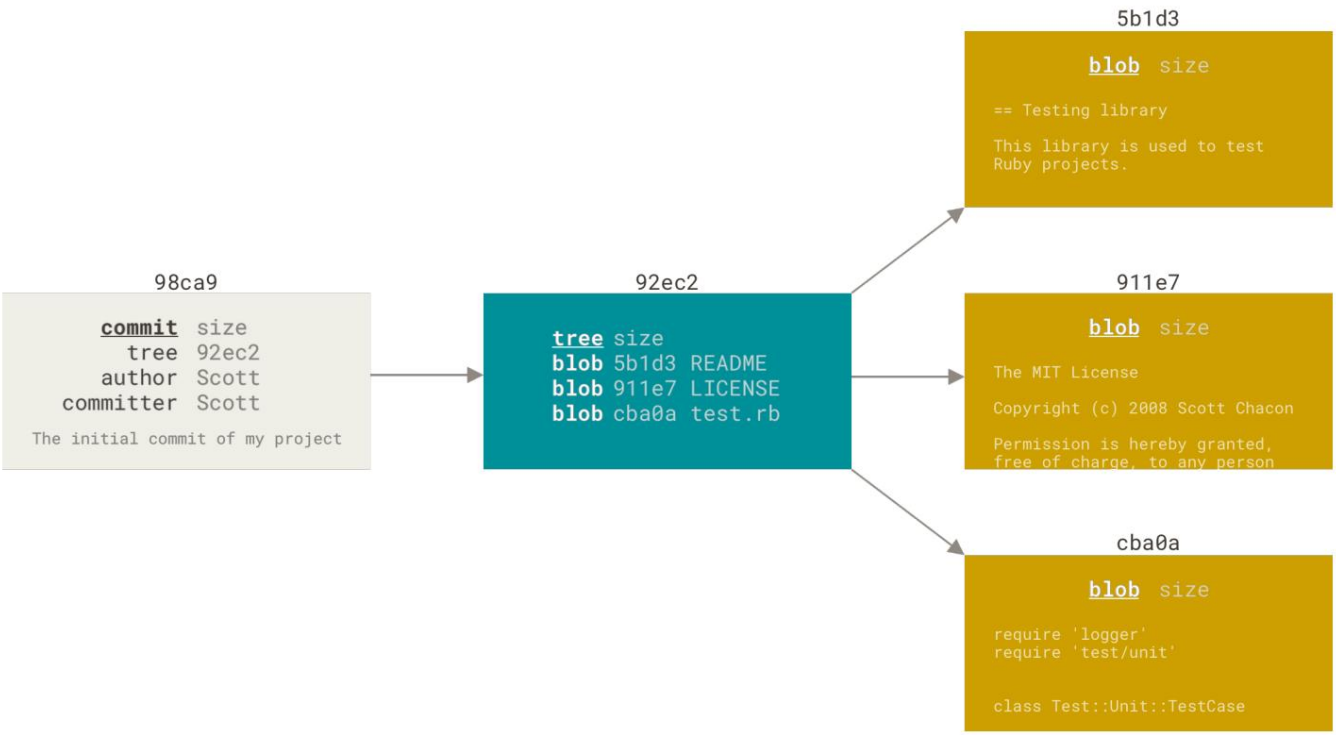


Figura 9. Um commit e sua árvore

Se você fizer algumas alterações e confirmar novamente, o próximo commit armazenará um ponteiro para o commit que veio imediatamente antes dele.

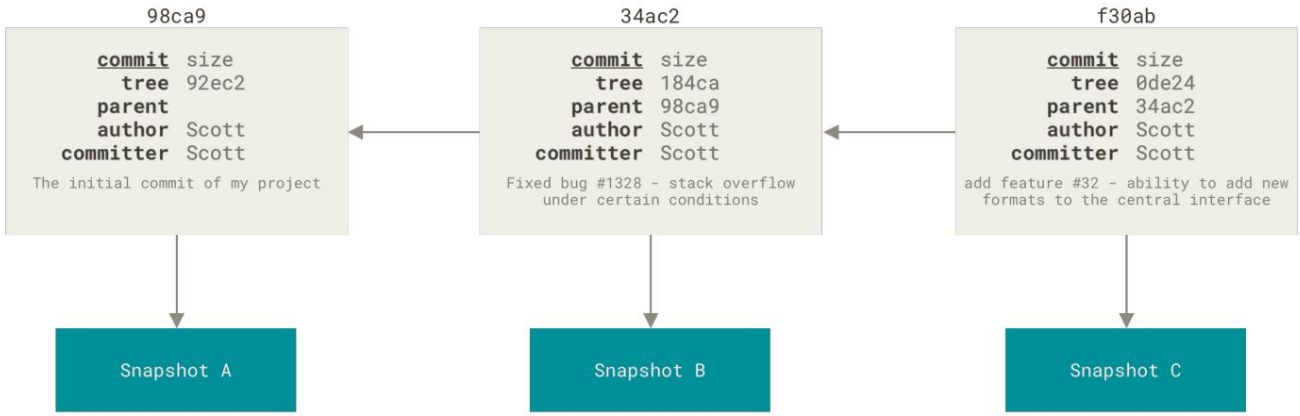


Figura 10. Commits e seus pais

Uma ramificação no Git é simplesmente um ponteiro leve e móvel para um desses commits. O nome padrão do branch no Git é master. Conforme você começa a fazer commits, você recebe um branch master que aponta para o último commit que você fez. Toda vez que você faz um commit, o ponteiro do branch master avança automaticamente.

ÿ

O branch “mestre” no Git não é um branch especial. É exatamente como qualquer outro ramo. A única razão pela qual quase todo repositório tem um é que o comando git init o cria por padrão e a maioria das pessoas não se preocupa em alterá-lo.

Machine Translated by Google

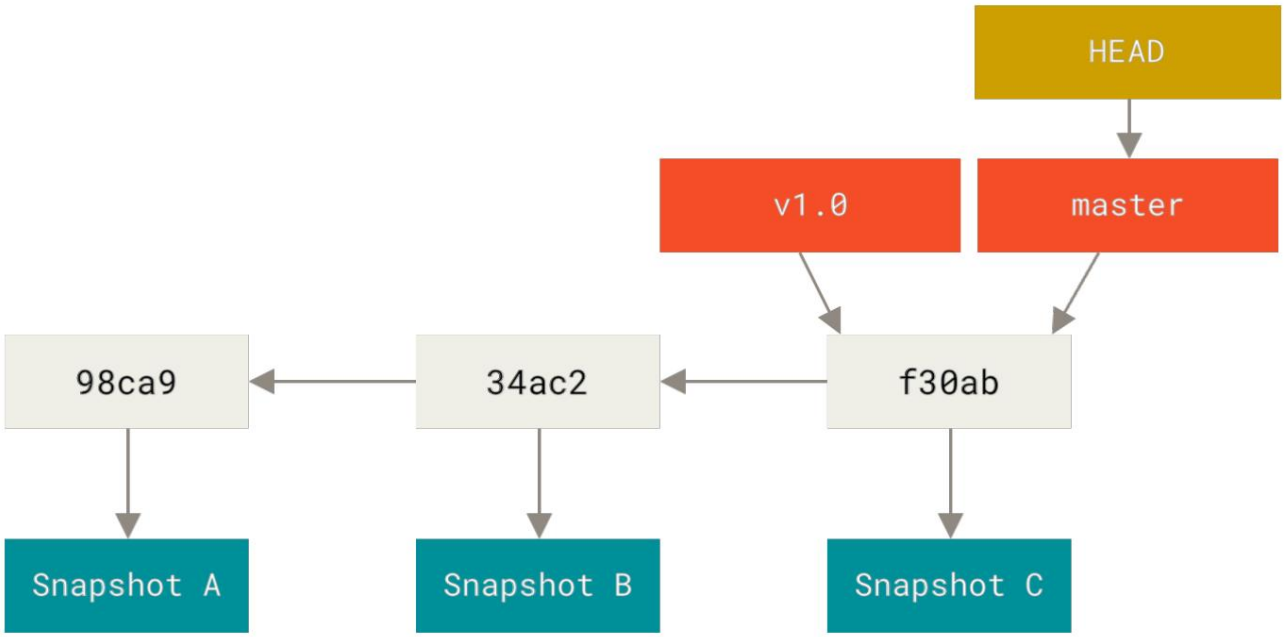


Figura 11. Um branch e seu histórico de commits

Criando uma Nova Filial

O que acontece quando você cria uma nova ramificação? Bem, isso cria um novo ponteiro para você se mover. Digamos que você queira criar um novo branch chamado testing. Você faz isso com o comando git branch :

```
teste de ramificação $ git
```

Isso cria um novo ponteiro para o mesmo commit em que você está atualmente.

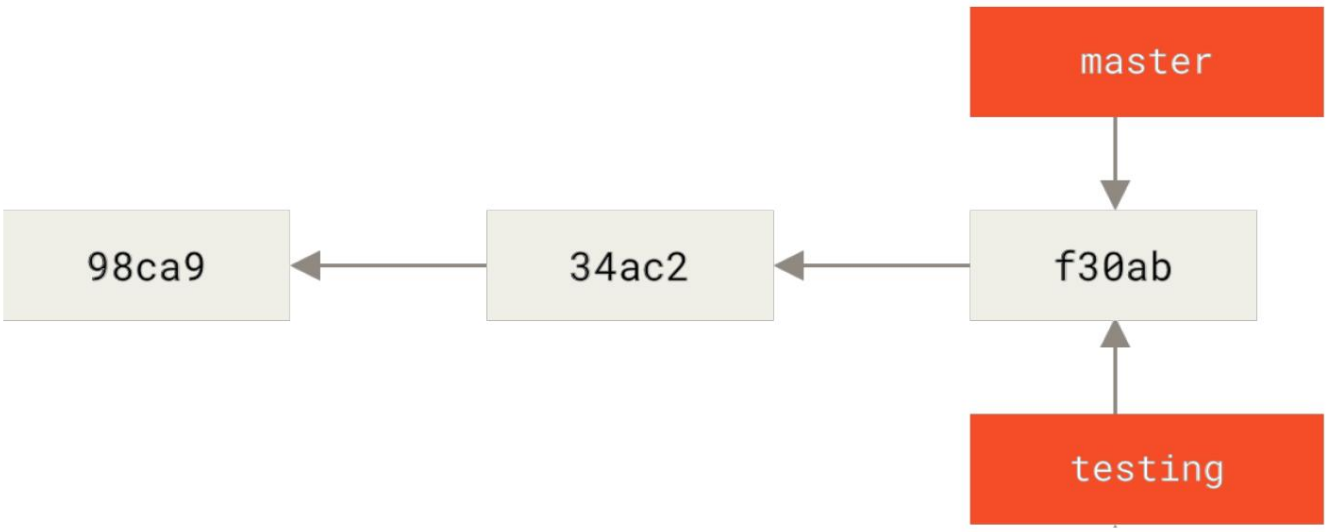


Figura 12. Duas ramificações apontando para a mesma série de commits

Como o Git sabe em qual ramo você está atualmente? Ele mantém um ponteiro especial chamado HEAD. Observe que isso é muito diferente do conceito de HEAD em outros VCSs aos quais você pode estar acostumado, como Subversion ou CVS. No Git, este é um ponteiro para a ramificação local em que você está atualmente. Neste caso, você ainda está no mestre. O comando git branch apenas *criou* um novo branch - não mudou para aquele

Machine Translated by Google

filial.

Figura 13. HEAD apontando para uma ramificação

Você pode ver isso facilmente executando um simples comando git log que mostra para onde os ponteiros de ramificação estão apontando. Esta opção é chamada --decorate.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Adicionar recurso #32 - capacidade de adicionar novos
formatos à interface central
34ac2 Fix bug #1328 - estouro de pilha sob certas condições 98ca9
Commit inicial
```

Você pode ver as ramificações master e de teste que estão bem ao lado do commit f30ab .

Mudando de Ramos

Para alternar para uma ramificação existente, execute o comando git checkout . Vamos mudar para o novo ramo de teste :

```
teste de checkout $ git
```

Isso move HEAD para apontar para o ramo de teste .

Machine Translated by Google

Figura 14. HEAD aponta para o branch atual

Qual é o significado disso? Bem, vamos fazer outro commit:

```
$ vim test.rb
$ git commit -a -m 'fez uma alteração'
```

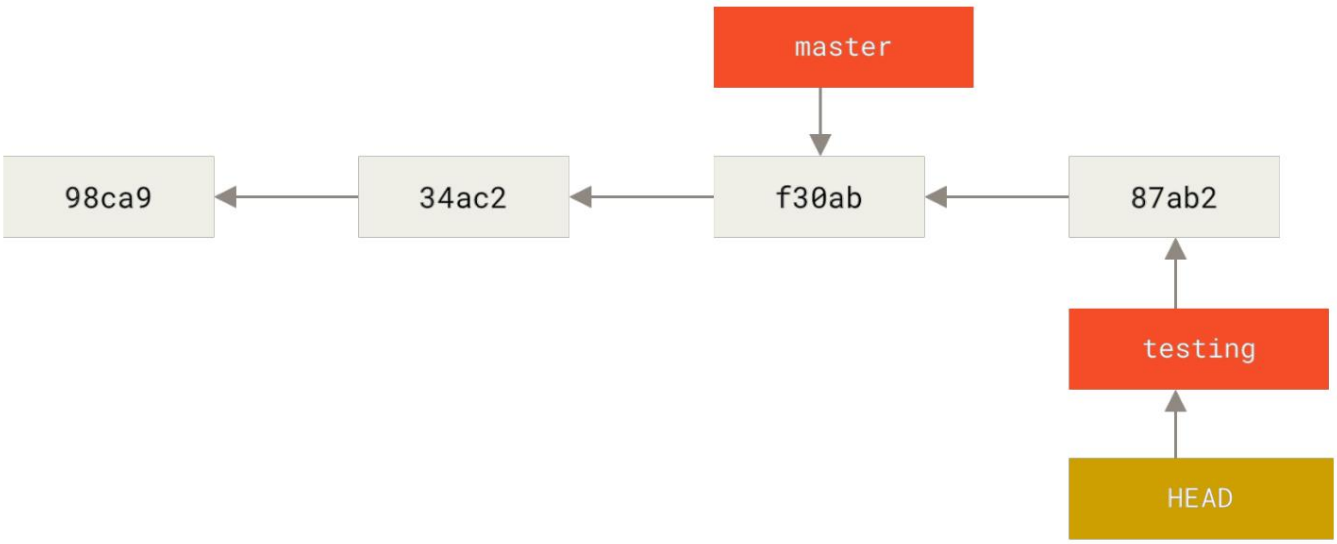


Figura 15. A ramificação HEAD avança quando um commit é feito

Isso é interessante, porque agora seu branch de teste avançou, mas seu branch master ainda aponta para o commit em que você estava quando executou git checkout para alternar os branches. Vamos voltar para o branch master :

```
$ git checkout mestre
```

ÿ git log *não mostra todos os branches o tempo todo*

Machine Translated by Google

Se você executasse o git log agora, poderia se perguntar para onde foi o branch "testing" que acabou de criar, pois ele não apareceria na saída.

O galho não desapareceu; O Git simplesmente não sabe que você está interessado nesse ramo e está tentando mostrar a você no que acha que você está interessado. Em outras palavras, por padrão, o git log mostrará apenas o histórico de commits abaixo do ramo que você check-out.

Para mostrar o histórico de commits para o branch desejado, você deve especificá-lo explicitamente: git log testing. Para mostrar todas as ramificações, adicione --all ao seu comando git log .

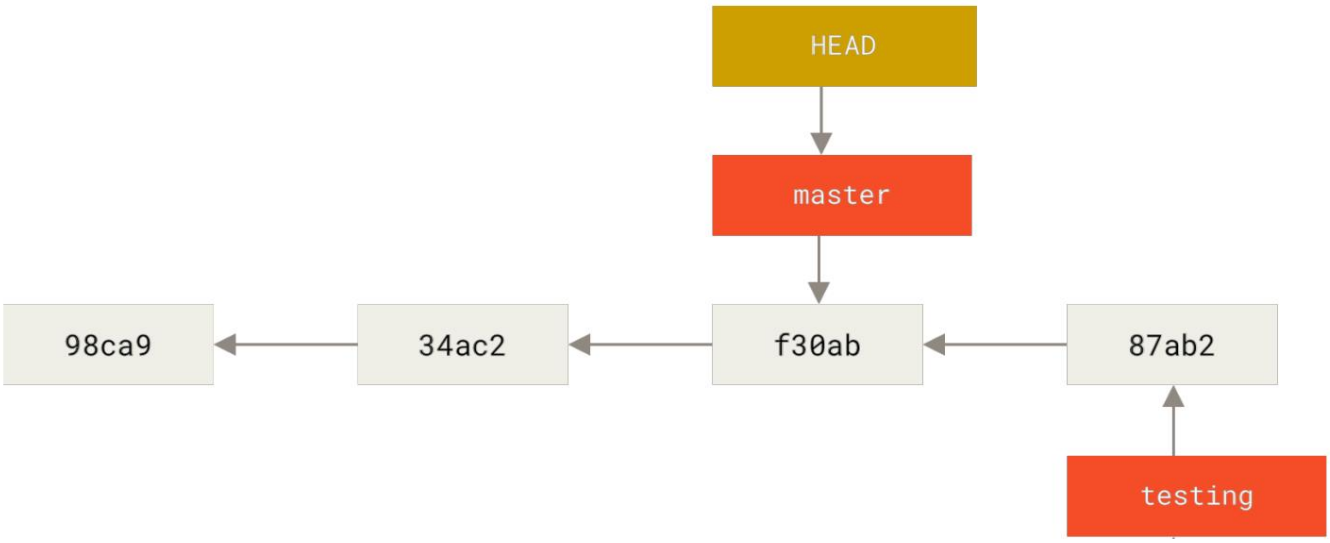


Figura 16. HEAD se move quando você faz o checkout

Esse comando fez duas coisas. Ele moveu o ponteiro HEAD de volta para apontar para a ramificação mestre e reverteu os arquivos em seu diretório de trabalho de volta para o instantâneo para o qual o mestre aponta. Isso também significa que as alterações feitas a partir deste ponto serão diferentes de uma versão mais antiga do projeto. Ele basicamente rebobina o trabalho que você fez em seu branch de teste para que você possa ir em uma direção diferente.

ÿ

A troca de ramificações altera os arquivos em seu diretório de trabalho

É importante observar que quando você troca de branches no Git, os arquivos em seu diretório de trabalho serão alterados. Se você mudar para uma ramificação mais antiga, seu diretório de trabalho será revertido para se parecer com a última vez que você fez commit naquela ramificação. Se o Git não puder fazer isso de forma limpa, ele não permitirá que você alterne.

Vamos fazer algumas alterações e confirmar novamente:

```
$ vim test.rb
$ git commit -a -m 'fez outras alterações'
```

Agora o histórico do seu projeto divergiu (consulte Histórico divergente). Você criou e mudou para um branch, fez algum trabalho nele e depois voltou para o branch principal e fez outro trabalho. Ambas as alterações são isoladas em branches separados: você pode alternar entre os branches e mesclá-los quando estiver pronto. E você fez tudo isso com um ramo simples,

Machine Translated by Google

comandos de check-out e commit .

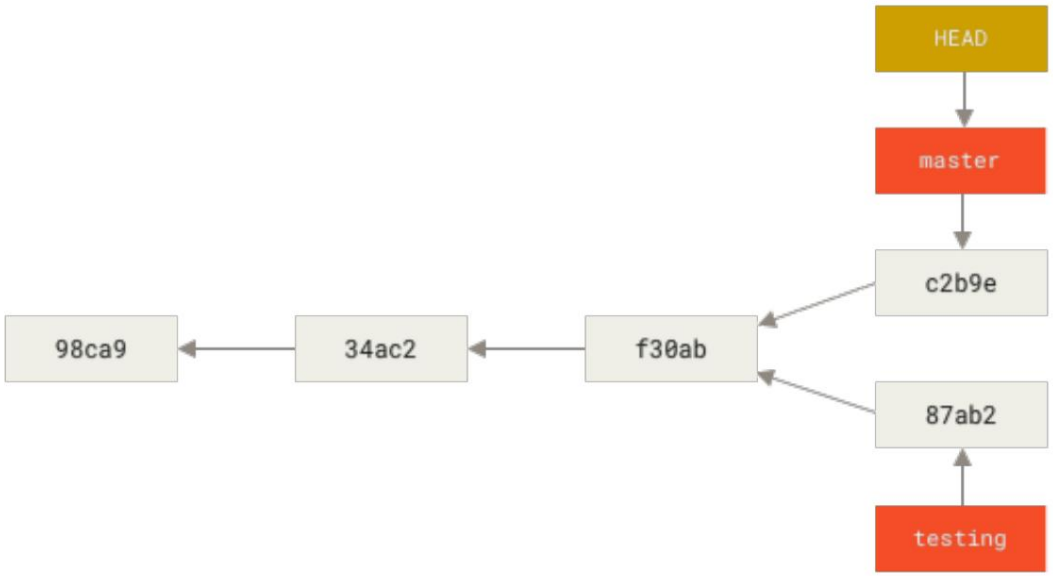


Figura 17. Histórico divergente

Você também pode ver isso facilmente com o comando git log . Se você executar git log --oneline --decorate --graph --all, ele imprimirá o histórico de seus commits, mostrando onde estão seus ponteiros de ramificação e como seu histórico divergiu.

```
$ git log --oneline --decorate --graph --all * c2b9e
(HEAD, master) Outras alterações feitas | *
 87ab2 (teste) Fez uma alteração |/ * f30ab
 Adicionar recurso #32 - capacidade de adicionar
novos formatos à interface central * 34ac2 Corrigir bug #1328 - estouro de pilha sob
certas condições * 98ca9 commit inicial do meu projeto
```

Como uma ramificação no Git é na verdade um arquivo simples que contém a soma de verificação SHA-1 de 40 caracteres do commit para o qual aponta, as ramificações são baratas de criar e destruir. Criar uma nova ramificação é tão rápido e simples quanto escrever 41 bytes em um arquivo (40 caracteres e uma nova linha).

Isso contrasta fortemente com a ramificação da maioria das ferramentas VCS mais antigas, que envolve a cópia de todos os arquivos do projeto em um segundo diretório. Isso pode levar vários segundos ou até minutos, dependendo do tamanho do projeto, enquanto no Git o processo é sempre instantâneo. Além disso, como estamos gravando os pais quando fazemos o commit, encontrar uma base de mesclagem adequada para mesclar é feito automaticamente para nós e geralmente é muito fácil de fazer. Esses recursos ajudam a incentivar os desenvolvedores a criar e usar branches com frequência.

Vamos ver por que você deve fazer isso.

Machine Translated by Google

ÿ

Criando uma nova ramificação e alternando para ela ao mesmo tempo

É típico criar um novo branch e querer mudar para esse novo branch ao mesmo tempo — isso pode ser feito em uma operação com git checkout -b <newbranchname>.

ÿ

A partir da versão 2.23 do Git, você pode usar git switch em vez de git checkout para:

- Alternar para um branch existente: git switch testing-branch.
- Crie um novo branch e mude para ele: git switch -c new-branch. A bandeira -c significa criar, você também pode usar o sinalizador completo: --create.
- Retorne ao branch verificado anteriormente: git switch -.

Ramificação e fusão básicas

Vejamos um exemplo simples de ramificação e fusão com um fluxo de trabalho que você pode usar no mundo real. Você seguirá estas etapas:

1. Faça algum trabalho em um site.
2. Crie uma ramificação para uma nova história de usuário na qual você está trabalhando.
3. Faça algum trabalho nesse ramo.

Nesse estágio, você receberá uma ligação informando que outro problema é crítico e que você precisa de um hotfix. Você fará o seguinte:

1. Mude para o seu ramo de produção.
2. Crie uma ramificação para adicionar o hotfix.
3. Depois de testado, mescle a ramificação do hotfix e envie para produção.
4. Volte para sua história de usuário original e continue trabalhando.

Ramificação Básica

Primeiro, digamos que você esteja trabalhando em seu projeto e já tenha alguns commits no branch master .

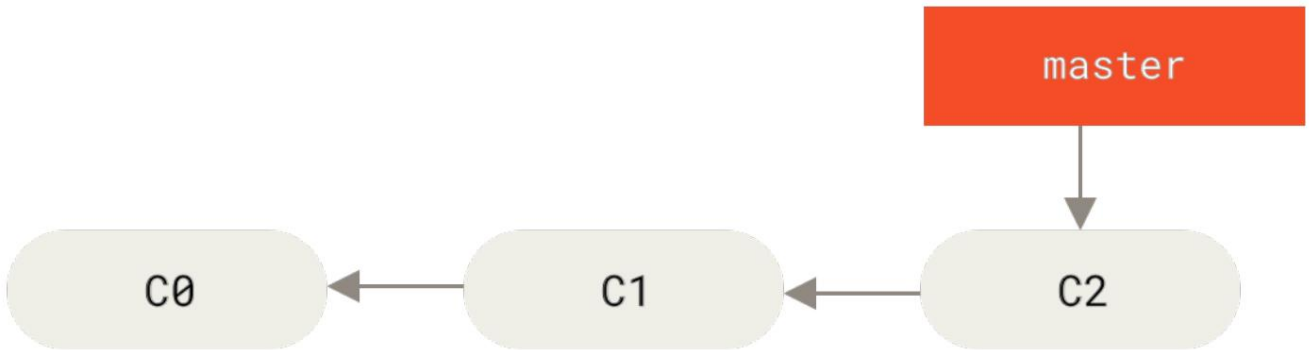


Figura 18. Um histórico de confirmação simples

Machine Translated by Google

Você decidiu que vai trabalhar no problema nº 53 em qualquer sistema de rastreamento de problemas que sua empresa usa. Para criar uma nova ramificação e alternar para ela ao mesmo tempo, você pode executar o comando git checkout com a opção -b :

```
$ git checkout -b iss53
Mudou para uma nova ramificação "iss53"
```

Isso é uma abreviação para:

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

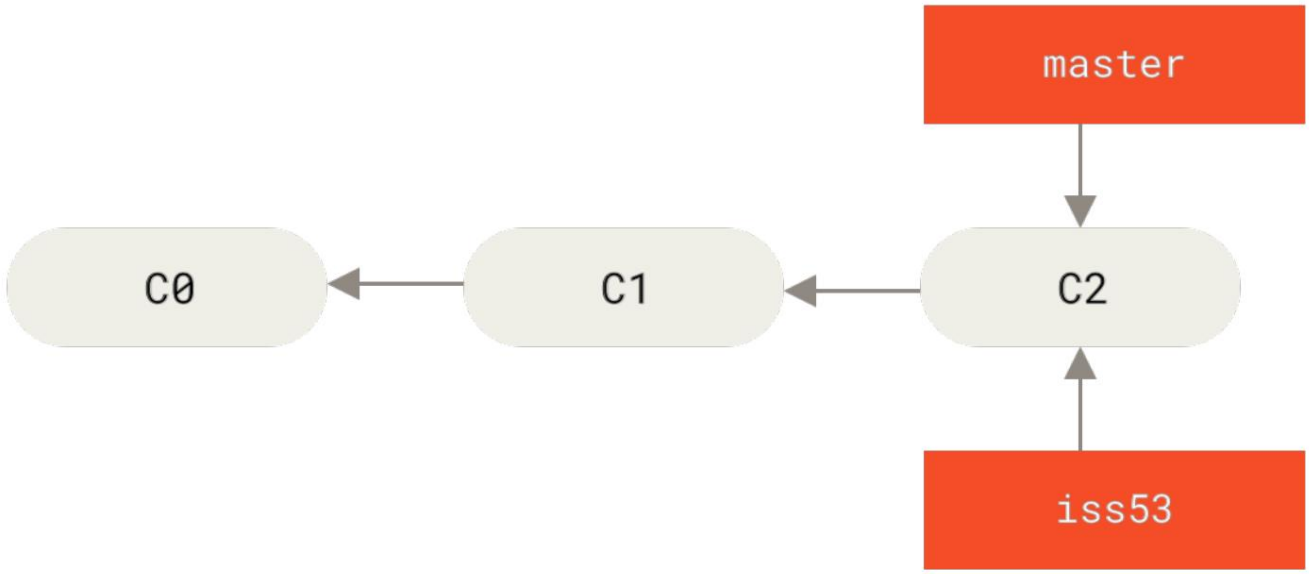


Figura 19. Criando um novo ponteiro de ramificação

Você trabalha em seu site e faz alguns commits. Fazer isso move o ramo iss53 para frente, porque você o verificou (ou seja, seu HEAD está apontando para ele):

```
$ vim index.html $ git
commit -a -m 'Criar novo rodapé [questão 53]'
```

Machine Translated by Google

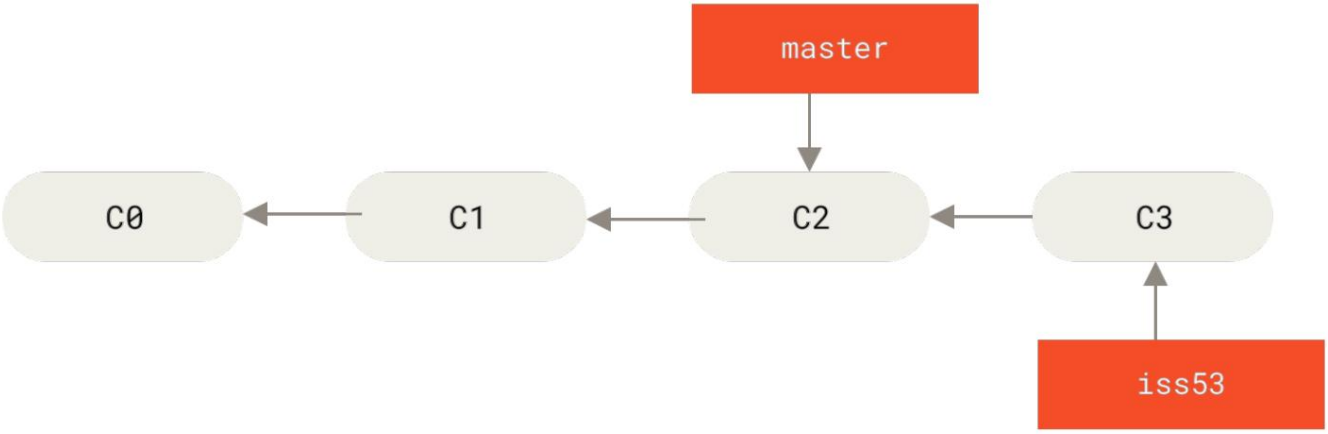


Figura 20. A ramificação iss53 *avançou com seu trabalho*

Agora você recebe a ligação de que há um problema com o site e precisa corrigi-lo imediatamente. Com Git, você não precisa implantar sua correção junto com as alterações iss53 que você fez e não precisa se esforçar muito para reverter essas alterações antes de trabalhar na aplicação de sua correção ao que está em Produção. Tudo o que você precisa fazer é voltar para o branch master .

No entanto, antes de fazer isso, observe que, se seu diretório de trabalho ou área de preparação tiver alterações não confirmadas que entrem em conflito com a ramificação que você está verificando, o Git não permitirá que você troque de ramificações. É melhor ter um estado de trabalho limpo ao trocar de branch. Existem maneiras de contornar isso (ou seja, ocultar e alterar commits) que abordaremos mais adiante, em Armazenamento e limpeza. Por enquanto, vamos supor que você tenha feito commit de todas as suas alterações, então você pode voltar para seu branch master :

```
$ git checkout mestre
Comutado para ramificação 'mestre'
```

Neste ponto, o diretório de trabalho do projeto está exatamente como estava antes de você começar a trabalhar no problema nº 53 e você pode se concentrar no hotfix. Este é um ponto importante a ser lembrado: quando você troca de branches, o Git redefine seu diretório de trabalho para se parecer com a última vez que você fez commit naquele branch. Ele adiciona, remove e modifica arquivos automaticamente para garantir que sua cópia de trabalho seja como a ramificação estava em seu último commit.

Em seguida, você tem um hotfix para fazer. Vamos criar uma ramificação de hotfix na qual trabalhar até que seja concluída:

```
$ git checkout -b hotfix Mudou
para um novo branch 'hotfix' $ vim index.html
$ git commit -a -m 'Corrigir endereço de e-
mail quebrado' [hotfix 1fb7853] Corrigir endereço de e-mail
quebrado 1 arquivo alterado, 2 inserções (+)
```

Machine Translated by Google

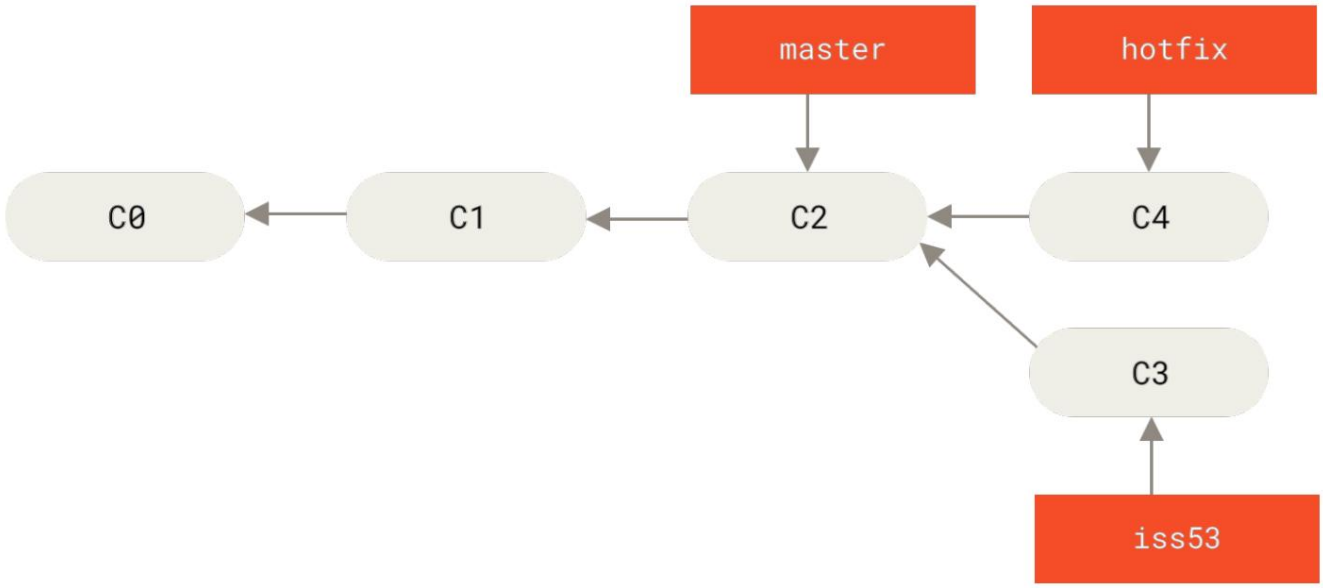


Figura 21. Ramificação de hotfix baseada no mestre

Você pode executar seus testes, certificar-se de que o hotfix é o que você deseja e, finalmente, mesclar a ramificação do hotfix de volta à ramificação principal para implantar na produção. Você faz isso com o comando git merge :

```
$ git checkout master
$ git merge hotfix
Atualizando f42c576..3a0874c
Avanço rápido index.html |
2 ++ 1 arquivo alterado, 2
inserções (+)
```

Você notará a frase “avanço rápido” nessa mesclagem. Como o commit C4 apontado pelo hotfix do branch no qual você fez o merge estava diretamente à frente do commit C2 em que você está, o Git simplesmente move o ponteiro para frente. Em outras palavras, quando você tenta mesclar um commit com um commit que pode ser alcançado seguindo o histórico do primeiro commit, o Git simplifica as coisas movendo o ponteiro para frente porque não há trabalho divergente para mesclar — isso é chamado de “ avanço rápido.”

Sua alteração agora está no instantâneo do commit apontado pela ramificação master e você pode implantar a correção.

Machine Translated by Google

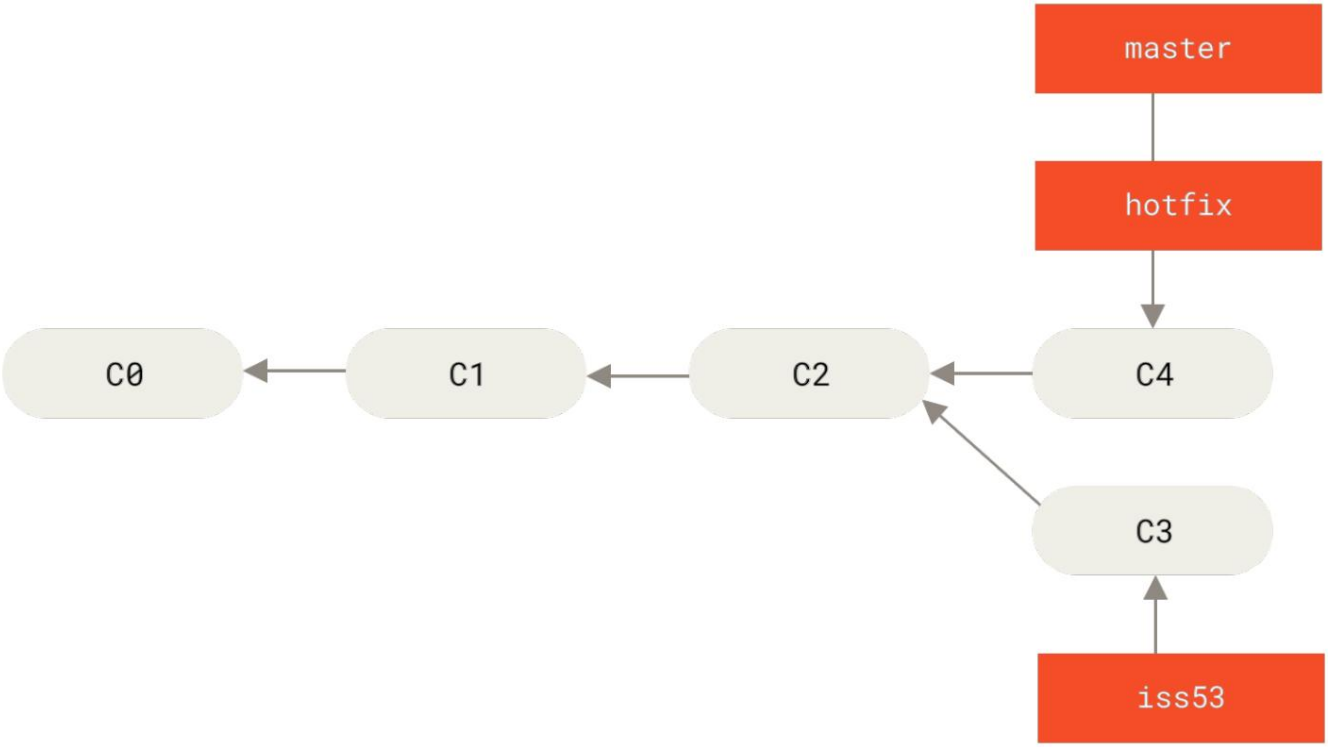


Figura 22. master *é avançado para* hotfix

Depois que sua correção super importante for implantada, você estará pronto para voltar ao trabalho que estava fazendo antes de ser interrompido. No entanto, primeiro você excluirá a ramificação do hotfix , porque não precisa mais dela - a ramificação principal aponta para o mesmo local. Você pode excluí-lo com a opção -d para git branch:

```
$ git branch -d hotfix
Hotfix de ramificação excluído (3a0874c).
```

Agora você pode voltar para sua ramificação de trabalho em andamento na edição nº 53 e continuar trabalhando nela.

```
$ git checkout iss53
Alterado para ramificação
"iss53" $ vim index.html $ git
commit -a -m 'Concluir o novo rodapé [problema 53]' [iss53
ad82d7a] Finalizar o novo rodapé [problema 53] 1 arquivo
alterado, 1 inserção (+)
```

Machine Translated by Google

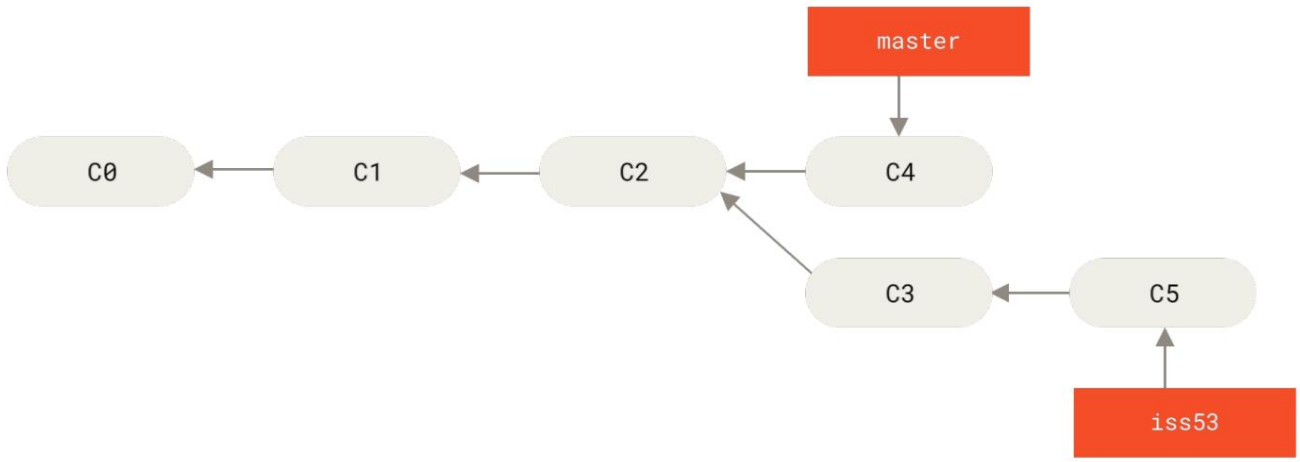


Figura 23. O trabalho continua no iss53

Vale a pena notar aqui que o trabalho que você fez em sua ramificação de hotfix não está contido nos arquivos em sua ramificação iss53 . Se precisar fazer o pull, você pode mesclar sua ramificação master com sua ramificação iss53 executando git merge master, ou pode esperar para integrar essas alterações até decidir puxar a ramificação iss53 de volta para master mais tarde.

Mesclagem Básica

Suponha que você tenha decidido que seu trabalho de número 53 está completo e pronto para ser mesclado em sua ramificação master . Para fazer isso, você mesclará sua ramificação iss53 com master, da mesma forma que você fundiu sua ramificação hotfix anteriormente. Tudo o que você precisa fazer é verificar a ramificação na qual deseja mesclar e executar o comando git merge :

```
$ git checkout master
Comutado para branch
'master' $ git merge iss53
Mesclagem feita pela estratégia 'recursiva'.
index.html | 1 + 1 inserção(+)
                                  alterado, 1
```

Isso parece um pouco diferente da mesclagem de hotfix que você fez anteriormente. Nesse caso, seu histórico de desenvolvimento divergiu de algum ponto mais antigo. Como o commit no branch em que você está não é um ancestral direto do branch no qual você está fazendo o merge, o Git precisa fazer algum trabalho. Nesse caso, o Git faz uma mesclagem simples de três vias, usando os dois instantâneos apontados pelas dicas de ramificação e o ancestral comum dos dois.

Machine Translated by Google

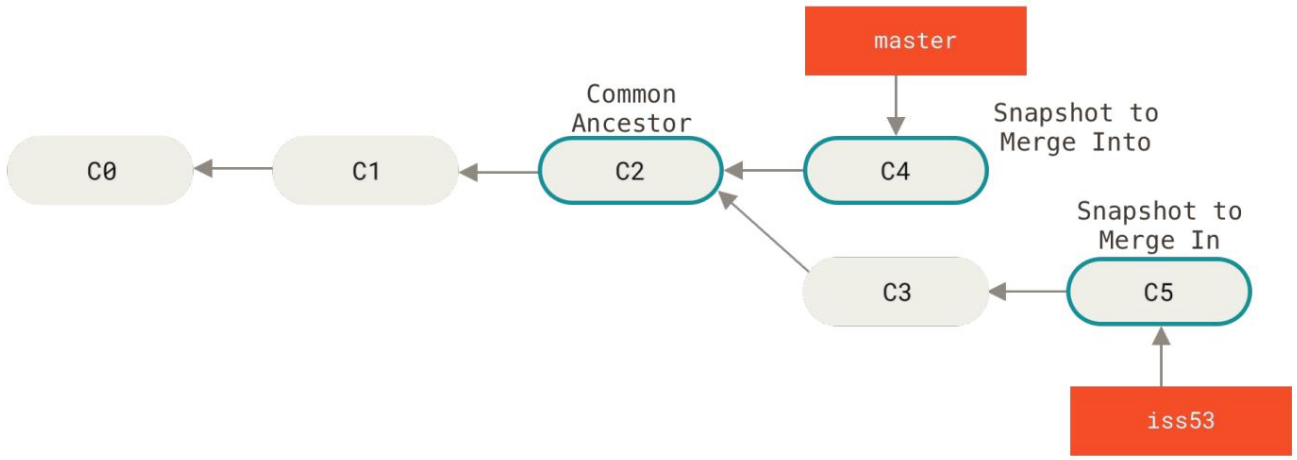


Figura 24. Três instantâneos usados em uma mesclagem típica

Em vez de apenas mover o ponteiro do branch para frente, o Git cria um novo instantâneo que resulta dessa mesclagem de três vias e cria automaticamente um novo commit que aponta para ele. Isso é chamado de confirmação de mesclagem e é especial porque tem mais de um pai.

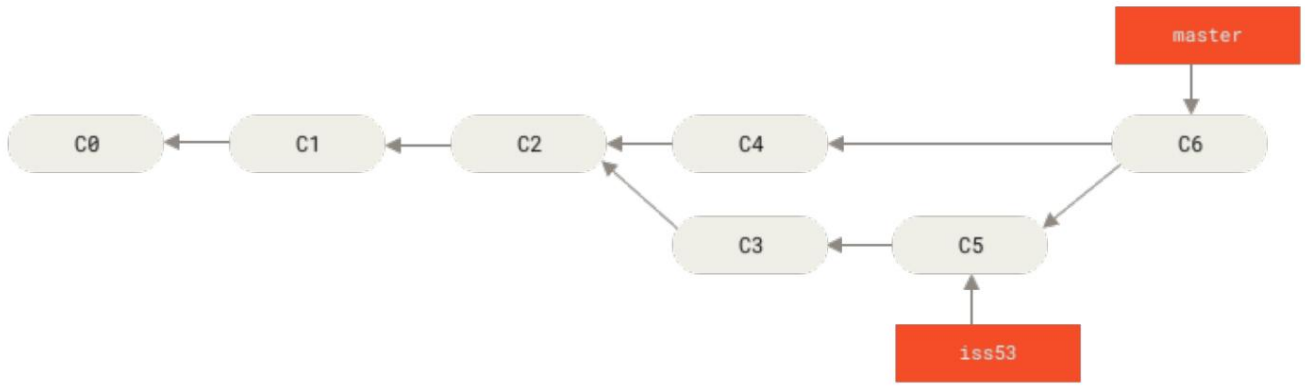


Figura 25. Uma confirmação de mesclagem

Agora que seu trabalho foi mesclado, você não precisa mais da ramificação iss53 . Você pode fechar o problema em seu sistema de rastreamento de problemas e excluir a ramificação:

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

Conflitos Básicos de Mesclagem

Ocasionalmente, esse processo não ocorre sem problemas. Se você alterou a mesma parte do mesmo arquivo de maneira diferente nas duas ramificações que está mesclando, o Git não conseguirá mesclá-las de forma limpa. Se a correção para o problema nº 53 modificou a mesma parte de um arquivo que a ramificação do hotfix , você obterá um conflito de mesclagem semelhante a este:

Machine Translated by Google

```
$ git merge iss53
Mesclagem automática
index.html CONFLICT (conteúdo): Conflito de mesclagem em
index.html Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

O Git não criou automaticamente um novo commit de mesclagem. Ele pausou o processo enquanto você resolve o conflito. Se você quiser ver quais arquivos não foram mesclados a qualquer momento após um conflito de mesclagem, execute git status:

```
$ git status On
branch master Você
tem caminhos não mesclados.
(corrija os conflitos e execute "git commit")

Caminhos não
mesclados: (use "git add <arquivo>..." para marcar a resolução)

ambos modificados:                index.html

nenhuma alteração adicionada ao commit (use "git add" e/ou "git commit -a")
```

Qualquer coisa que tenha conflitos de mesclagem e não tenha sido resolvida é listada como não mesclada. O Git adiciona marcadores de resolução de conflito padrão aos arquivos que têm conflitos, para que você possa abri-los manualmente e resolver esses conflitos. Seu arquivo contém uma seção que se parece com isto:

```
<<<<<<< HEAD:index.html
<div id="footer">contato: email.support@github.com</div>
=======
<div id="footer"> entre
em contato conosco em support@github.com </div>
>>>>>>> iss53:index.html
```

Isso significa que a versão em HEAD (seu branch master , porque foi isso que você verificou quando executou o comando merge) é a parte superior desse bloco (tudo acima de =======), enquanto a versão em sua ramificação iss53 se parece com tudo na parte inferior. Para resolver o conflito, você deve escolher um lado ou outro ou mesclar os conteúdos você mesmo. Por exemplo, você pode resolver esse conflito substituindo o bloco inteiro por isto:

```
<div id="rodapé">
entre em contato conosco pelo e-mail.support@github.com
</div>
```

Esta resolução tem um pouco de cada seção, e as linhas <<<<<<<, ======= e >>>>>>> foram completamente removidas. Depois de resolver cada uma dessas seções em cada arquivo em conflito, execute git add

Machine Translated by Google

em cada arquivo para marcá-lo como resolvido. A preparação do arquivo o marca como resolvido no Git.

Se quiser usar uma ferramenta gráfica para resolver esses problemas, você pode executar git mergetool, que ativa uma ferramenta de mesclagem visual apropriada e orienta você nos conflitos:

```
$ git mergetool

Esta mensagem é exibida porque 'merge.tool' não está configurado.
Veja 'git mergetool --tool-help' ou 'git help config' para mais detalhes. 'git mergetool' agora
 tentará usar uma das seguintes ferramentas: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge
gvimdiff difuso diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge Merging: index.html




Conflito de mesclagem normal para 'index.html':
{local}: arquivo modificado {remote}: arquivo
modificado Pressione return para iniciar a ferramenta
de resolução de mesclagem (opendiff):
```

Se você quiser usar uma ferramenta de mesclagem diferente da padrão (Git escolheu opendiff neste caso porque o comando foi executado em um Mac), você pode ver todas as ferramentas suportadas listadas na parte superior após “uma das seguintes ferramentas”. Basta digitar o nome da ferramenta que você prefere usar.

ÿ

Se você precisar de ferramentas mais avançadas para resolver conflitos complicados de mesclagem, abordaremos mais sobre mesclagem em Mesclagem Avançada.

Depois de sair da ferramenta de mesclagem, o Git pergunta se a mesclagem foi bem-sucedida. Se você disser ao script que sim, ele preparará o arquivo para marcá-lo como resolvido para você. Você pode executar git status novamente para verificar se todos os conflitos foram resolvidos:

```
$ git status On
branch master Todos
os conflitos corrigidos, mas você ainda está mesclando. (use
"git commit" para concluir a mesclagem)

Mudanças a serem confirmadas:

modificado: index.html
```

Se você estiver satisfeito com isso e verificar se tudo o que tinha conflitos foi testado, digite git commit para finalizar o merge commit. A mensagem de confirmação por padrão se parece com isto:

```
Mesclar ramificação 'iss53'

Conflitos:
```

Machine Translated by Google

```
index.html #

# Parece que você pode estar cometendo uma mesclagem.
# Se não estiver correto, remova o arquivo # .git/MERGE_HEAD
# e tente novamente.



# Por favor, insira a mensagem de confirmação para suas alterações. Linhas
começando com # com '#' serão ignoradas, e uma mensagem vazia aborta o commit.
# On branch master #
Todos os conflitos corrigidos, mas você ainda está mesclando.
#
# Mudanças a serem confirmadas:
# modificado: index.html
#
```

Se você acha que seria útil para outras pessoas olhando para esta fusão no futuro, você pode modificar esta mensagem de confirmação com detalhes sobre como você resolveu a fusão e explicar por que você fez as mudanças que você fez se elas não forem óbvias.

Gerenciamento de Filiais

Agora que você criou, fundiu e excluiu algumas ramificações, vamos ver algumas ferramentas de gerenciamento de ramificações que serão úteis quando você começar a usar ramificações o tempo todo.

O comando git branch faz mais do que apenas criar e excluir branches. Se você executá-lo sem argumentos, obterá uma listagem simples de suas ramificações atuais:

```
$ git branch
iss53
* mestre
testando
```

Observe o caractere * que precede o branch master : ele indica o branch que você está verificando atualmente (ou seja, o branch para o qual HEAD aponta). Isso significa que, se você confirmar neste ponto, a ramificação master será movida para frente com seu novo trabalho. Para ver o último commit em cada branch, você pode executar git branch -v:

```
$ git branch -v
iss53 93b412c Fix javascript issue master
* 7a98805 Merge branch 'iss53' testing 782fd34
Adicione scott à lista de autores no readme
```

As opções úteis --merged e --no-merged podem filtrar esta lista para branches que você tem ou ainda não fez merge no branch em que você está atualmente. Para ver quais branches já estão mesclados no branch em que você está, você pode executar git branch --merged:

Machine Translated by Google

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

Como você já fez o merge no iss53 anteriormente, você o vê em sua lista. Ramos nesta lista sem o * na frente deles geralmente podem ser excluídos com git branch -d; você já incorporou o trabalho deles em outro ramo, então não vai perder nada.

Para ver todos os branches que contêm trabalhos nos quais você ainda não fez o merge, você pode executar git branch --no -merged:

```
$ git branch --teste sem
mesclagem
```

Isso mostra seu outro ramo. Como contém trabalho que ainda não foi mesclado, tentar excluí-lo com git branch -d falhará:

```
$ git branch -d testing
error: O branch 'testing' não está totalmente mesclado.
Se tiver certeza de que deseja excluí-lo, execute 'git branch -D testing'.
```

Se você realmente deseja excluir o branch e perder esse trabalho, pode forçá-lo com -D, como indica a mensagem útil.

ÿ

As opções descritas acima, --merged e --no-merged , se não for dado um nome de commit ou branch como argumento, mostrarão a você o que é, respectivamente, mesclado ou não mesclado em seu branch *atual* .

Você sempre pode fornecer um argumento adicional para perguntar sobre o estado de mesclagem em relação a alguma outra ramificação sem verificar essa outra ramificação primeiro, como em, o que não foi mesclado na ramificação principal ?

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

Alteração do nome de uma filial

ÿ

Não renomeie branches que ainda estão em uso por outros colaboradores. Não renomeie uma ramificação como master/main/mainline sem ter lido a seção "Alterando o nome da ramificação master".

Suponha que você tenha uma ramificação chamada bad-branch-name e queira alterá-la para corrigida

Machine Translated by Google

branch-name, mantendo todo o histórico. Você também deseja alterar o nome da ramificação no controle remoto (GitHub, GitLab, outro servidor). Como você faz isso?

Renomeie a ramificação localmente com o comando git branch --move :

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

Isso substitui seu nome de ramificação incorreto pelo nome de ramificação corrigido, mas essa alteração é apenas local por enquanto. Para permitir que outras pessoas vejam o branch corrigido no controle remoto, pressione-o:

```
$ git push --set-upstream origem nome-da-filial corrigida
```

Agora vamos dar uma breve olhada em onde estamos agora:

```
$ git branch --all *
nome da ramificação corrigido
principais
remotos/origem/comandos com nome de
ramificação inválido/origem/comandos com nome de
ramificação corrigido/origem/principal
```

Observe que você está na ramificação nome-da-filial corrigida e está disponível no controle remoto. No entanto, a ramificação com o nome incorreto também está presente, mas você pode excluí-la executando o seguinte comando:

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

Agora, o nome incorreto da ramificação foi totalmente substituído pelo nome da ramificação corrigida.

Alterando o nome da ramificação principal

ÿ

Mudar o nome de um branch como master/main/mainline/default quebrará as integrações, serviços, utilitários auxiliares e scripts de build/release que seu repositório usa. Antes de fazer isso, certifique-se de consultar seus colaboradores. Além disso, certifique-se de fazer uma pesquisa completa em seu repositório e atualizar todas as referências ao antigo nome da ramificação em seu código e scripts.

Renomeie sua ramificação master local para main com o seguinte comando:

```
$ git branch --move mestre principal
```

Não há mais ramificação master local , porque ela foi renomeada para ramificação principal .

Para permitir que outras pessoas vejam a nova ramificação principal , você precisa enviá-la para o controle remoto. Isso torna o renomeado

Machine Translated by Google

filial disponível no controle remoto.

```
$ git push --set-upstream origem principal
```

Agora terminamos com o seguinte estado:

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

Sua ramificação principal local se foi, pois foi substituída pela ramificação principal . A ramificação principal está presente no controle remoto. No entanto, a ramificação master antiga ainda está presente no controle remoto. Outros colaboradores continuarão usando o branch master como base de seu trabalho, até que você faça mais algumas alterações.

Agora você tem mais algumas tarefas à sua frente para concluir a transição:

- Qualquer projeto que dependa deste precisará atualizar seu código e/ou configuração.
- Atualize todos os arquivos de configuração do executor de teste.
- Ajuste os scripts de compilação e lançamento.
- Redirecione as configurações em seu host de repositório para coisas como ramificação padrão do repositório, regras de mesclagem e outras coisas que correspondem aos nomes dos ramos.
- Atualize as referências à ramificação antiga na documentação.
- Feche ou mescle qualquer pull request que tenha como alvo a ramificação antiga.

Depois de concluir todas essas tarefas e ter certeza de que o branch principal funciona exatamente como o branch master , você pode excluir o branch master :

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

Fluxos de trabalho de ramificação

Agora que você tem o básico de ramificação e fusão, o que pode ou deve fazer com eles? Nesta seção, abordaremos alguns fluxos de trabalho comuns que essa ramificação leve possibilita, para que você possa decidir se deseja incorporá-los em seu próprio ciclo de desenvolvimento.

Filiais de Longa Duração

Como o Git usa uma mesclagem simples de três vias, geralmente é fácil fazer a mesclagem de uma ramificação para outra várias vezes durante um longo período. Isso significa que você pode ter várias ramificações que estão sempre abertas e que você usa para diferentes etapas do seu ciclo de desenvolvimento; você pode mesclar regularmente de alguns deles para outros.

Machine Translated by Google

Muitos desenvolvedores Git têm um fluxo de trabalho que adota essa abordagem, como ter apenas código totalmente estável em sua ramificação principal — possivelmente apenas código que foi ou será lançado. Eles têm outra ramificação paralela chamada develop ou next da qual trabalham ou usam para testar a estabilidade - nem sempre é estável, mas sempre que chega a um estado estável, pode ser mesclado em master.
Ele é usado para puxar ramificações de tópico (ramificações de curta duração, como sua ramificação iss53 anterior) quando estão prontas, para garantir que passem em todos os testes e não introduzam bugs.

Na realidade, estamos falando de ponteiros subindo na linha de commits que você está fazendo. Os branches estáveis estão mais abaixo na linha do seu histórico de commits, e os branches mais avançados estão mais avançados no histórico.

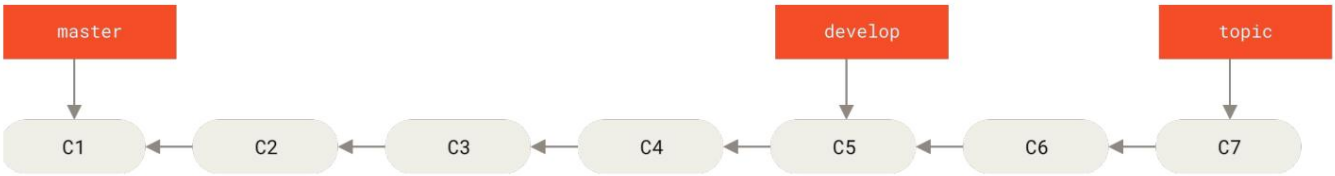


Figura 26. Uma visão linear da ramificação de estabilidade progressiva

Geralmente é mais fácil pensar neles como silos de trabalho, onde conjuntos de commits se transformam em um silo mais estável quando são totalmente testados.

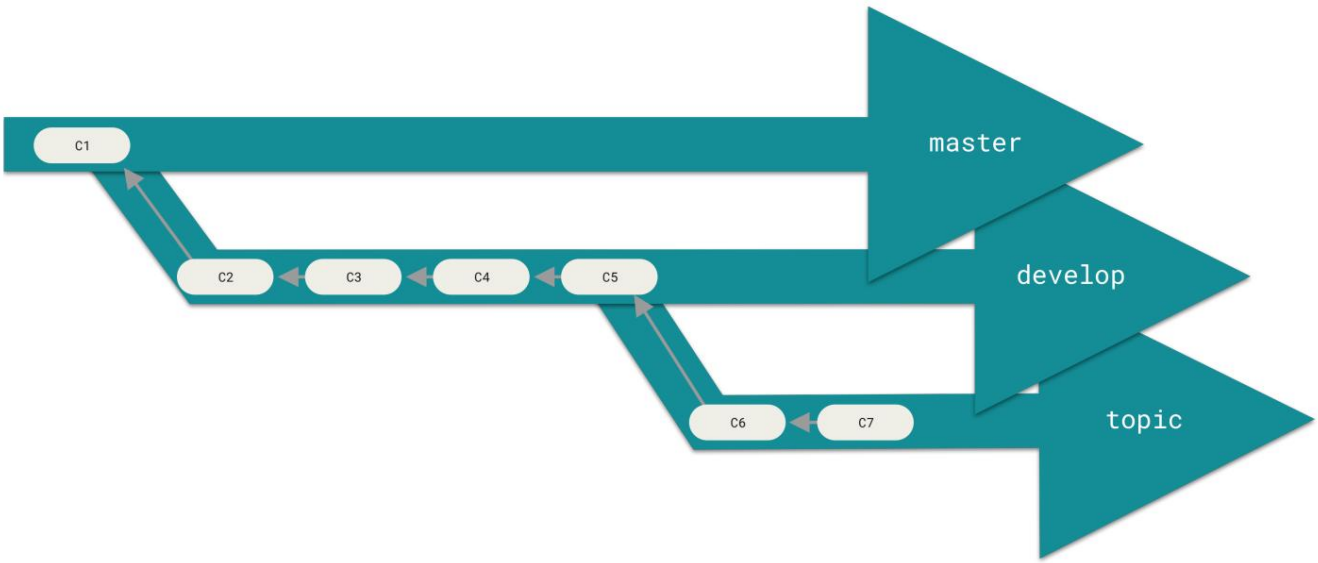


Figura 27. Uma visão de "silo" da ramificação de estabilidade progressiva

Você pode continuar fazendo isso por vários níveis de estabilidade. Alguns projetos maiores também têm uma ramificação proposta ou pu (atualizações propostas) que possui ramificações integradas que podem não estar prontas para ir para a próxima ramificação principal . A ideia é que suas ramificações estejam em vários níveis de estabilidade; quando atingem um nível mais estável, são fundidos no ramo acima deles. Novamente, não é necessário ter várias ramificações de execução longa, mas geralmente é útil, especialmente quando você está lidando com projetos muito grandes ou complexos.

Ramificações de Tópicos

As ramificações de tópicos, no entanto, são úteis em projetos de qualquer tamanho. Uma ramificação de tópico é uma ramificação de curta duração que você cria e usa para um único recurso específico ou trabalho relacionado. Isso é algo que você provavelmente nunca fez com um VCS antes porque geralmente é muito caro criar e mesclar

Machine Translated by Google

galhos. Mas no Git é comum criar, trabalhar, mesclar e excluir branches várias vezes ao dia.

Você viu isso na última seção com os ramos iss53 e hotfix que você criou. Você fez alguns commits neles e os apagou diretamente depois de mesclá-los em seu branch principal. Essa técnica permite que você alterne o contexto de forma rápida e completa — como seu trabalho é separado em silos onde todas as mudanças naquele ramo têm a ver com aquele tópico, é mais fácil ver o que aconteceu durante a revisão de código e tal. Você pode manter as alterações lá por minutos, dias ou meses e mesclá-las quando estiverem prontas, independentemente da ordem em que foram criadas ou trabalhadas.

Considere um exemplo de fazer algum trabalho (no master), ramificar para um problema (iss91), trabalhar um pouco nele, ramificar o segundo branch para tentar outra maneira de lidar com a mesma coisa (iss91v2), voltar ao seu branch master e trabalhando lá por um tempo, e depois se ramificando para fazer algum trabalho que você não tem certeza se é uma boa ideia (ramificação idiota). Seu histórico de commits será mais ou menos assim:

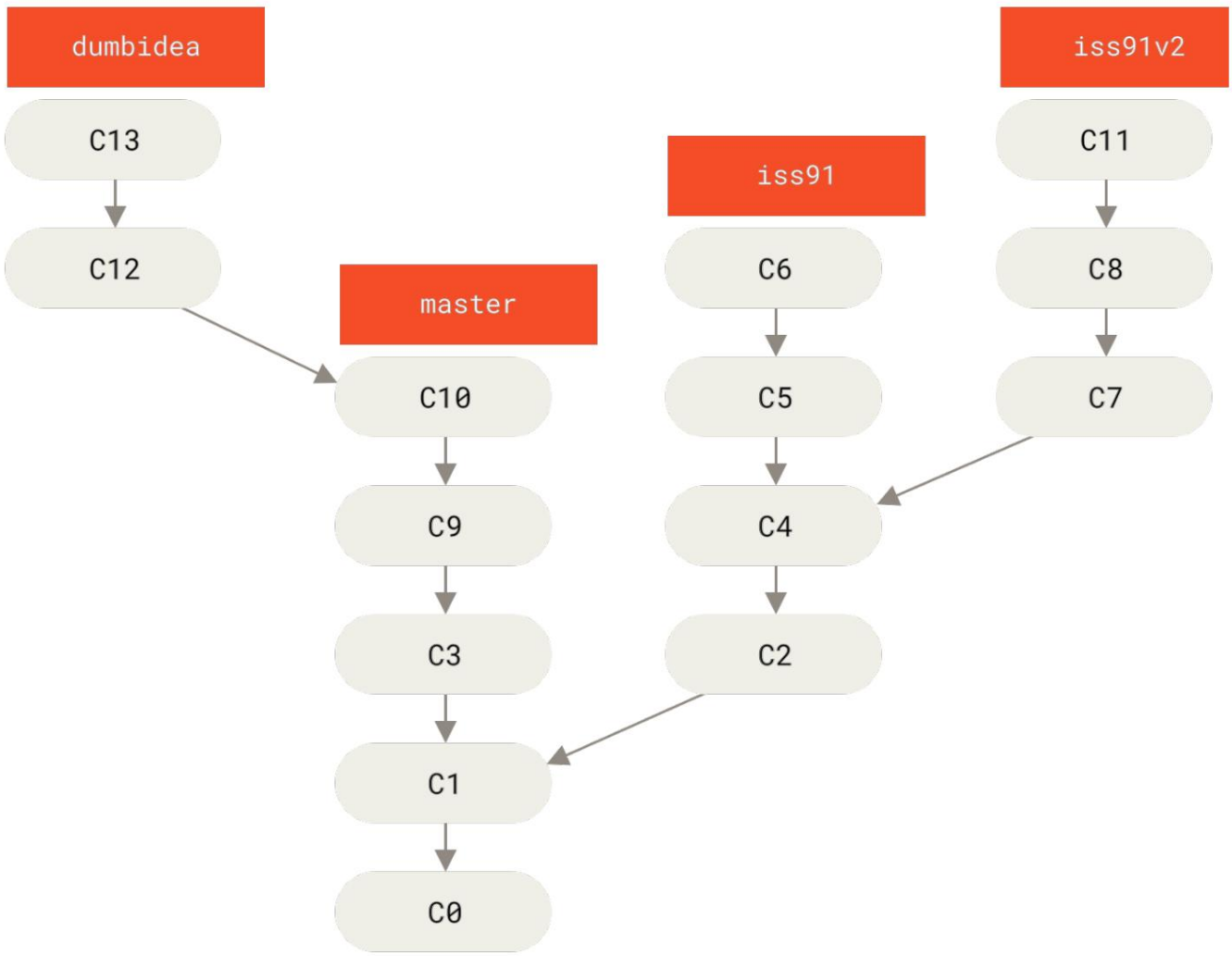


Figura 28. Múltiplas ramificações de tópicos

Agora, digamos que você decida que prefere a segunda solução para o seu problema (iss91v2); e você mostrou o ramo dumbidea para seus colegas de trabalho, e acabou sendo genial. Você pode jogar fora o ramo iss91 original (perdendo os commits C5 e C6) e mesclar os outros dois. Seu histórico fica assim:

Machine Translated by Google

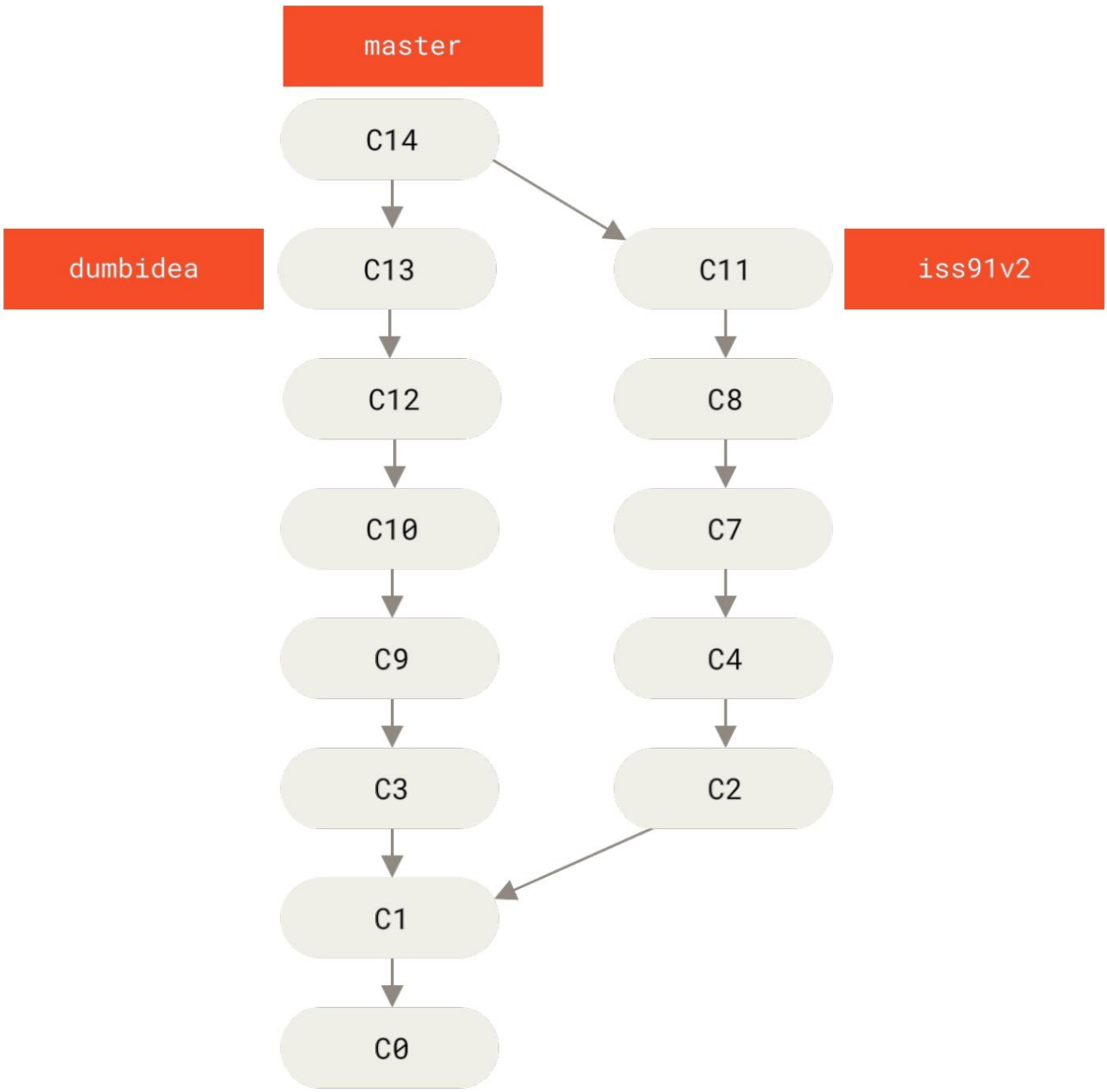


Figura 29. Histórico após a fusão de dumbidea *e* iss91v2

Entraremos em mais detalhes sobre os vários fluxos de trabalho possíveis para seu projeto Git no Distributed Git, portanto, antes de decidir qual esquema de ramificação seu próximo projeto usará, certifique-se de ler esse capítulo.

É importante lembrar quando você está fazendo tudo isso que essas ramificações são completamente locais. Quando você está ramificando e mesclando, tudo está sendo feito apenas em seu repositório Git — não há comunicação com o servidor.

Filiais remotas

Referências remotas são referências (ponteiros) em seus repositórios remotos, incluindo branches, tags e assim por diante. Você pode obter uma lista completa de referências remotas explicitamente com git ls-remote <remote> ou git remote show <remote> para branches remotos, bem como mais informações. No entanto, uma maneira mais comum é aproveitar as ramificações de rastreamento remoto.

Machine Translated by Google

As ramificações de rastreamento remoto são referências ao estado das ramificações remotas. São referências locais que você não pode mover; O Git os move para você sempre que você fizer qualquer comunicação de rede, para garantir que eles representem com precisão o estado do repositório remoto. Pense neles como favoritos, para lembrá-lo de onde as ramificações em seus repositórios remotos estavam na última vez em que você se conectou a elas.

Os nomes das ramificações de rastreamento remoto assumem o formato <remote>/<branch>. Por exemplo, se você quiser ver a aparência da ramificação master em seu controle remoto de origem na última vez em que você se comunicou com ela, verifique a ramificação origin/master . Se você estivesse trabalhando em um problema com um parceiro e ele enviasse uma ramificação iss53 , você poderia ter sua própria ramificação local iss53 , mas a ramificação no servidor seria representada pela ramificação de rastreamento remoto origin/iss53.

Isso pode ser um pouco confuso, então vamos ver um exemplo. Digamos que você tenha um servidor Git em sua rede em git.ourcompany.com. Se você clonar a partir disso, o comando clone do Git nomeia automaticamente sua origem para você, puxa todos os seus dados, cria um ponteiro para onde está sua ramificação master e a nomeia origin/master localmente. O Git também oferece a você sua própria ramificação master local começando no mesmo local que a ramificação master da origem , para que você tenha algo para trabalhar.

ÿ

“origem” não é especial

Assim como o nome da ramificação “mestre” não tem nenhum significado especial no Git, “origem” também não. Enquanto “master” é o nome padrão para uma ramificação inicial quando você executa git init , que é a única razão pela qual é amplamente usado, “origin” é o nome padrão para um controle remoto quando você executa git clone. Se você executar git clone -o booyah , terá booyah/master como sua ramificação remota padrão.

Machine Translated by Google

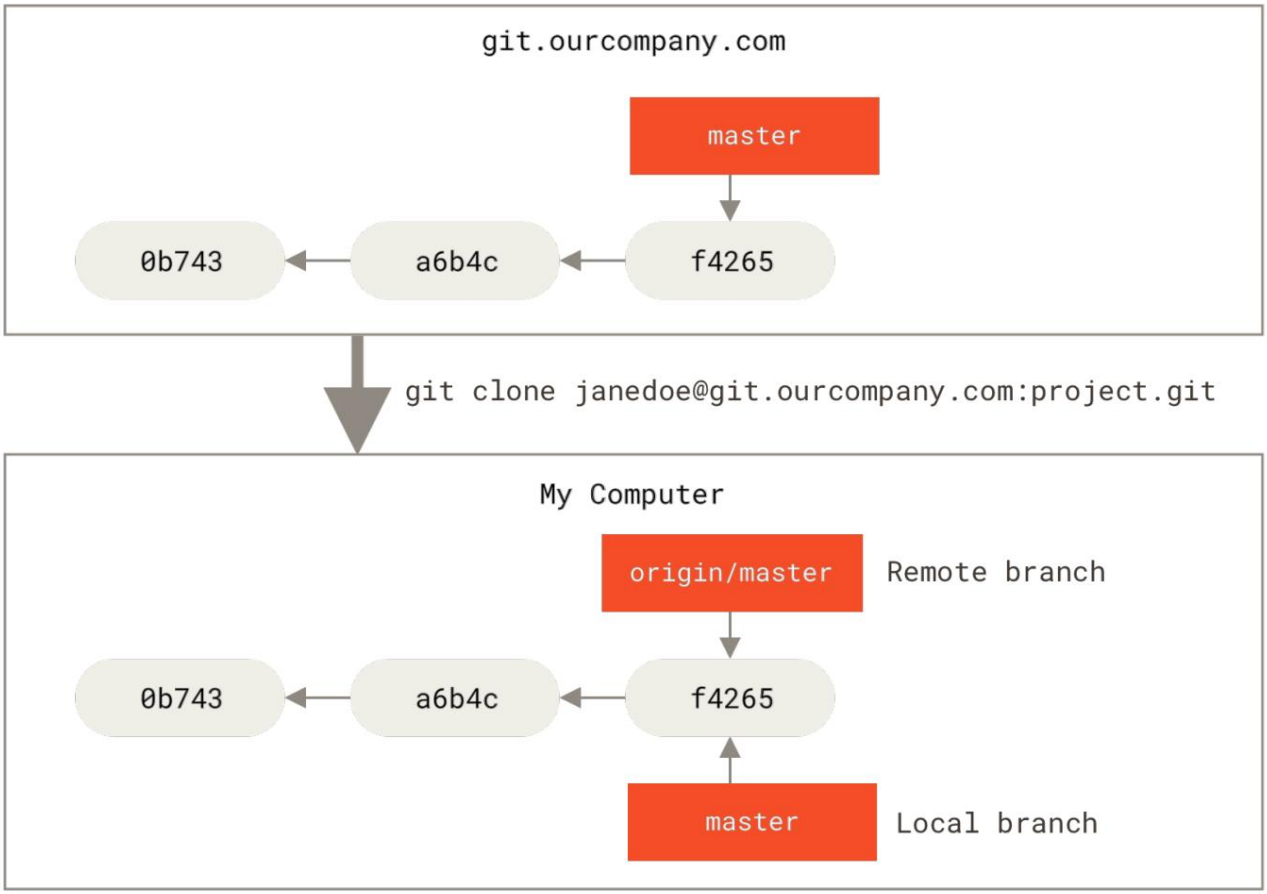


Figura 30. Servidor e repositórios locais após a clonagem

Se você fizer algum trabalho em sua ramificação principal local e, enquanto isso, outra pessoa enviar para git.ourcompany.com e atualizar sua ramificação principal , seus históricos avançarão de maneira diferente. Além disso, desde que você fique fora de contato com seu servidor de origem , seu ponteiro de origem/mestre não mover.

Machine Translated by Google

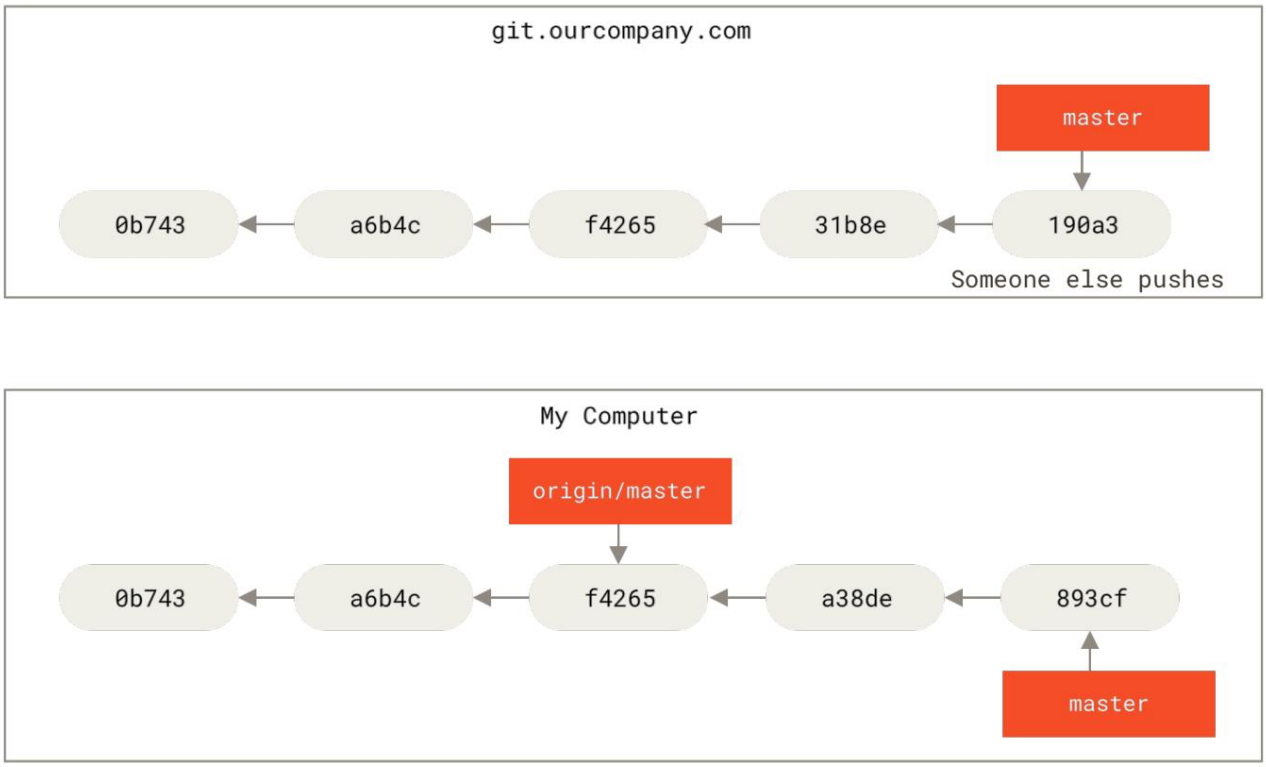


Figura 31. Trabalho local e remoto podem divergir

Para sincronizar seu trabalho com um determinado controle remoto, execute um comando git fetch <remote> (no nosso caso, git fetch origin). Este comando procura qual é a "origem" do servidor (neste caso, é git.ourcompany.com), busca todos os dados dele que você ainda não possui e atualiza seu banco de dados local, movendo seu ponteiro de origem/mestre para sua posição nova e mais atualizada.

Machine Translated by Google

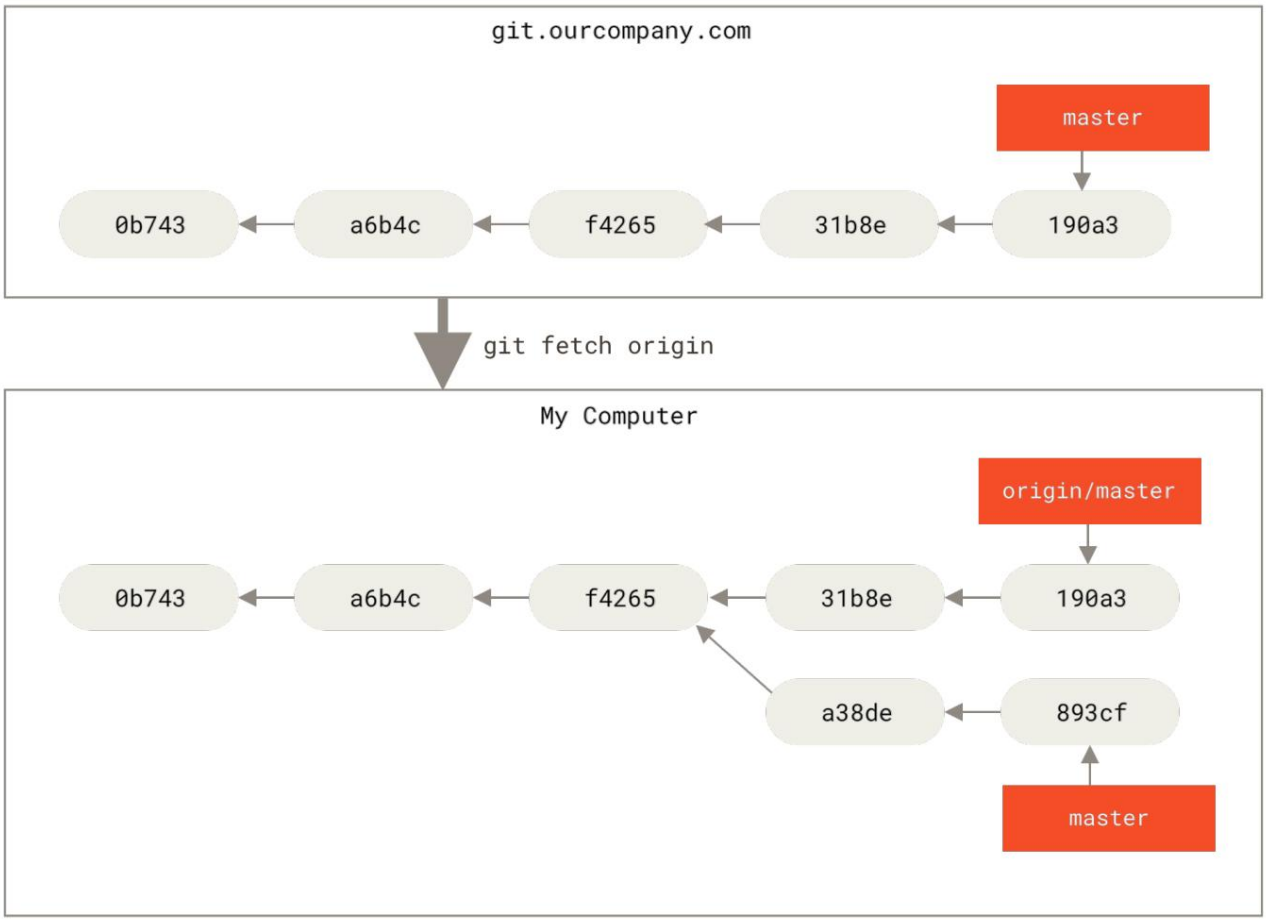


Figura 32. git fetch atualiza suas ramificações de rastreamento remoto

Para demonstrar a existência de vários servidores remotos e como são as ramificações remotas desses projetos remotos, vamos supor que você tenha outro servidor Git interno que seja usado apenas para desenvolvimento por uma de suas equipes de sprint. Este servidor está em git.team1.ourcompany.com. Você pode adicioná-lo como uma nova referência remota ao projeto em que está trabalhando atualmente, executando o comando git remote add conforme abordamos no Git Basics. Nomeie esse time remoto como um, que será seu nome curto para todo o URL.

Machine Translated by Google

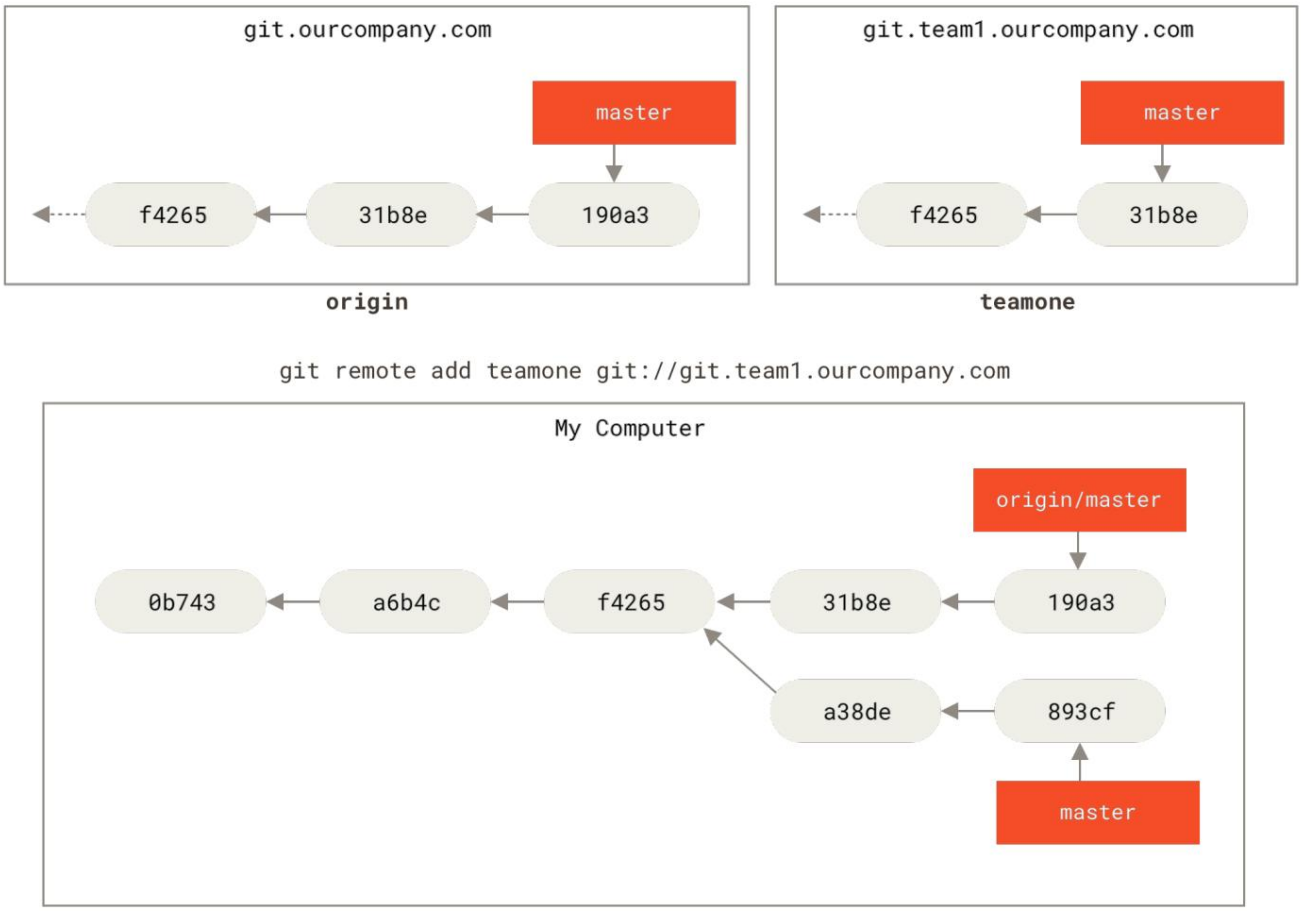


Figura 33. Adicionando outro servidor como remoto

Agora, você pode executar git fetch teamone para buscar tudo o que o servidor teamone remoto possui e que você ainda não possui. Como esse servidor tem um subconjunto dos dados que seu servidor de origem tem agora, o Git não busca dados, mas define um branch de rastreamento remoto chamado teamone/master para apontar para o commit que teamone tem como seu branch master .

Machine Translated by Google

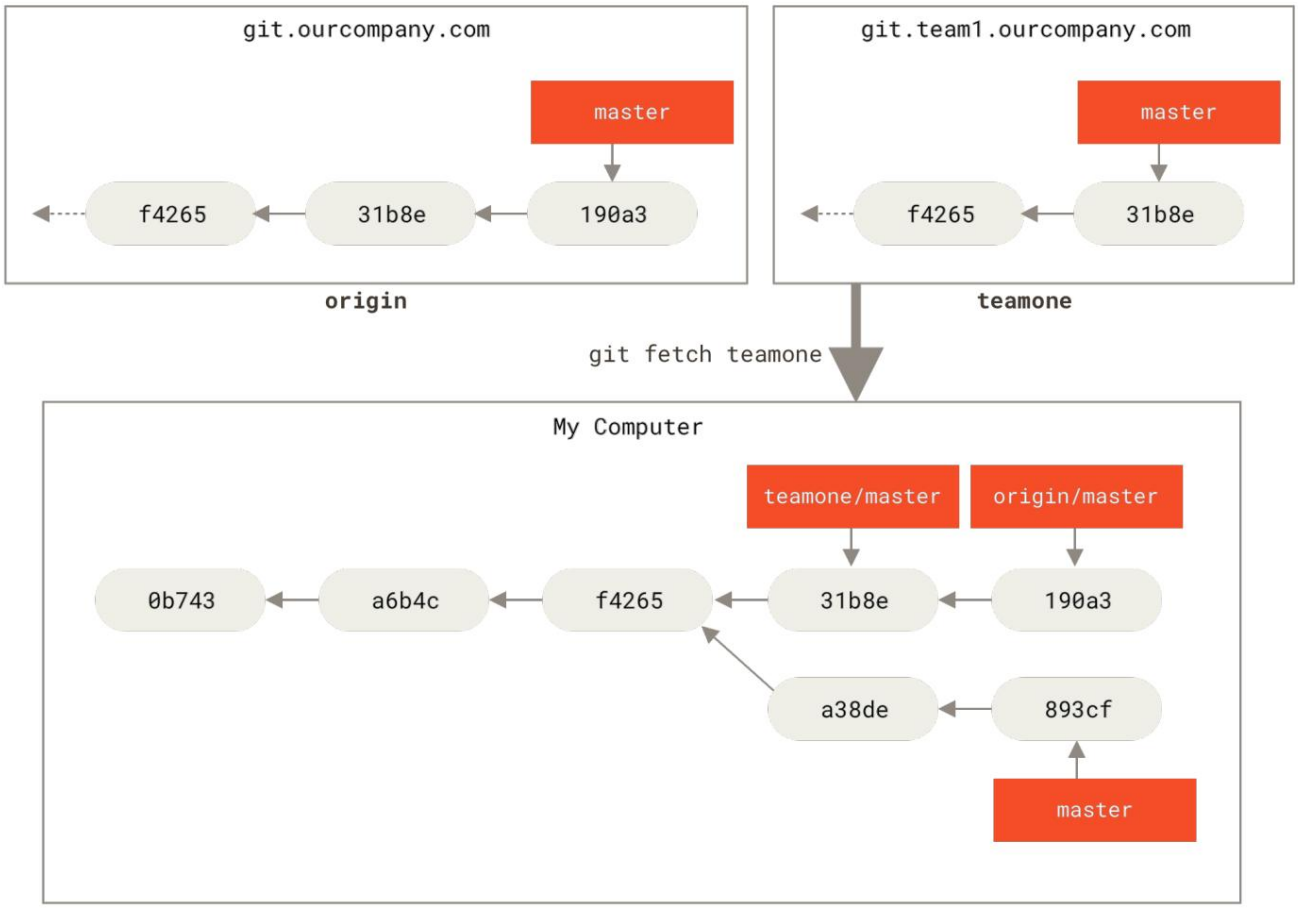


Figura 34. Ramal de rastreamento remoto para teamone/master

Empurrando

Quando você deseja compartilhar uma ramificação com o mundo, precisa enviá-la para um controle remoto ao qual tenha acesso de gravação. Suas ramificações locais não são sincronizadas automaticamente com os remotos para os quais você escreve - você precisa enviar explicitamente as ramificações que deseja compartilhar. Dessa forma, você pode usar ramificações privadas para o trabalho que não deseja compartilhar e enviar apenas as ramificações de tópico nas quais deseja colaborar.

Se você tiver uma ramificação chamada serverfix na qual deseja trabalhar com outras pessoas, poderá enviá-la da mesma forma que enviou sua primeira ramificação. Execute git push <remote> <branch>:

```
$ git push origin serverfix
Contando objetos: 24, concluído.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (15/15), concluído.
Objetos de escrita: 100% (24/24), 1,91 KiB | 0 byte/s, pronto.
Total 24 (delta 2), reutilizado 0 (delta 0)
Para https://github.com/schacon/simplegit *
[novo ramo]                serverfix -> serverfix
```

Este é um pequeno atalho. O Git expande automaticamente o nome da ramificação serverfix para refs/heads/serverfix:refs/heads/serverfix, o que significa: “Pegue minha ramificação local serverfix e envie-a para atualizar a ramificação serverfix remota ”. Veremos os refs/heads/ part em detalhes no Git

Machine Translated by Google

Internals, mas geralmente você pode deixá-lo desligado. Você também pode fazer git push origin serverfix:serverfix, que faz a mesma coisa - diz: "Pegue meu serverfix e torne-o o serverfix remoto". Você pode usar esse formato para enviar uma ramificação local para uma ramificação remota com nome diferente. Se você não quiser que seja chamado de serverfix no remoto, você pode executar git push origin serverfix:awesomebranch para enviar seu branch serverfix local para o branch awesomebranch no projeto remoto.

ÿ

Não digite sua senha toda vez

Se você estiver usando um URL HTTPS para enviar, o servidor Git solicitará seu nome de usuário e senha para autenticação. Por padrão, ele solicitará essas informações no terminal para que o servidor possa informar se você tem permissão para enviar.

Se você não quiser digitá-lo toda vez que pressionar, poderá configurar um “cache de credenciais”. O mais simples é apenas mantê-lo na memória por alguns minutos, o que você pode configurar facilmente executando git config --global credential.helper cache.

Para obter mais informações sobre as várias opções de cache de credenciais disponíveis, consulte Armazenamento de credenciais.

Na próxima vez que um de seus colaboradores fizer uma busca no servidor, eles obterão uma referência de onde a versão do serverfix do servidor está na ramificação remota origin/serverfix:

```
$ git fetch origin remote:
Contando objetos: 7, concluído. remote:
Comprimindo objetos: 100% (2/2), concluído. remoto: Total 3
(delta 0), reutilizado 3 (delta 0)
Desempacotando objetos: 100% (3/3), concluído.
De https://github.com/schacon/simplegit * [nova filial]
serverfix                                  -> origem/serverfix
```

É importante observar que quando você faz uma busca que desativa novas ramificações de rastreamento remoto, você não tem automaticamente cópias locais editáveis delas. Em outras palavras, neste caso, você não tem um novo branch serverfix — você tem apenas um ponteiro origin/serverfix que não pode ser modificado.

Para mesclar este trabalho em sua ramificação de trabalho atual, você pode executar git merge origin/serverfix. Se você deseja sua própria ramificação serverfix na qual possa trabalhar, pode baseá-la em sua ramificação de rastreamento remoto:

```
$ git checkout -b serverfix origin/serverfix Branch serverfix
configurado para rastrear o branch remoto serverfix desde a origem.
Mudou para uma nova ramificação 'serverfix'
```

Isso fornece uma ramificação local na qual você pode trabalhar e começa onde origin/serverfix está.

Filiais de Rastreamento

Fazer check-out de uma ramificação local de uma ramificação de rastreamento remoto cria automaticamente o que é chamado de

Machine Translated by Google

"ramo de rastreamento" (e o ramo que ele rastreia é chamado de "ramo upstream"). Filiais de rastreamento são filiais locais que têm um relacionamento direto com uma filial remota. Se você estiver em um branch de rastreamento e digitar git pull, o Git saberá automaticamente de qual servidor buscar e em qual branch fazer o merge.

Quando você clona um repositório, ele geralmente cria automaticamente uma ramificação mestre que rastreia origem/mestre. No entanto, você pode configurar outras ramificações de rastreamento, se desejar - aquelas que rastreiam ramificações em outros remotos ou não rastreiam a ramificação principal . O caso simples é o exemplo que você acabou de ver, executando git checkout -b <branch> <remote>/<branch>. Esta é uma operação comum o suficiente para que o Git forneça a abreviatura --track :

```
$ git checkout --track origin/serverfix Branch serverfix
configurado para rastrear branch remoto serverfix desde a origem.
Mudou para uma nova ramificação 'serverfix'
```

Na verdade, isso é tão comum que existe até um atalho para esse atalho. Se o nome do branch que você está tentando verificar (a) não existir e (b) corresponder exatamente a um nome em apenas um controle remoto, o Git criará um branch de rastreamento para você:

```
$ git checkout serverfix Branch
serverfix configurado para rastrear o branch remoto serverfix desde a origem.
Mudou para uma nova ramificação 'serverfix'
```

Para configurar uma ramificação local com um nome diferente da ramificação remota, você pode facilmente usar a primeira versão com um nome de ramificação local diferente:

```
$ git checkout -b sf origin/serverfix Branch sf
configurado para rastrear serverfix de branch remoto desde a origem.
Mudou para um novo ramo 'sf'
```

Agora, seu branch local sf irá puxar automaticamente de origin/serverfix.

Se você já tem uma ramificação local e deseja defini-la como uma ramificação remota que acabou de baixar ou deseja alterar a ramificação upstream que está rastreando, pode usar a opção -u ou --set- upstream - to para git branch para defini-lo explicitamente a qualquer momento.

```
$ git branch -u origin/serverfix Branch
serverfix configurado para rastrear o branch remoto serverfix desde a origem.
```

ÿ

abreviação upstream

Quando você tem uma ramificação de rastreamento configurada, pode referenciar sua ramificação upstream com a abreviação @{upstream} ou @{u} . Portanto, se você estiver na ramificação master e estiver rastreando origin/master, poderá dizer algo como git merge @{u} em vez de git merge origin/master , se desejar.

Machine Translated by Google

Se você quiser ver quais branches de rastreamento você configurou, você pode usar a opção -vv para git branch. Isso listará suas filiais locais com mais informações, incluindo o que cada filial está rastreando e se sua filial local está à frente, atrás ou ambos.

```
$ git branch -vv
iss53 7e424c3 [origin/iss53: à frente 2] Adicionar colchetes esquecidos
mestre 1ae2a45 [origin/master] Implantar correção de índice *
serverfix f8674d9 [teamone/server-fix-good: à frente 3, atrás de 1] Isso deve fazer o teste
5ea463a Tente algo novo
```

Então, aqui podemos ver que nossa ramificação iss53 está rastreando origin/iss53 e está "à frente" por dois, o que significa que temos dois commits localmente que não são enviados ao servidor. Também podemos ver que nossa ramificação master está rastreando origin/master e está atualizada. Em seguida, podemos ver que nosso branch serverfix está rastreando o branch server-fix-good em nosso servidor teamone e está à frente por três e atrás por um, o que significa que há um commit no servidor que ainda não foi mesclado e três commits localmente que não enviamos. Por fim, podemos ver que nosso branch de teste não está rastreando nenhum branch remoto.

É importante observar que esses números são apenas desde a última vez que você obteve de cada servidor. Este comando não alcança os servidores, ele informa sobre o que foi armazenado em cache desses servidores localmente. Se você quiser números totalmente atualizados à frente e atrás, você precisará buscar em todos os seus controles remotos antes de executá-lo. Você poderia fazer isso assim:

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

Puxar

Embora o comando git fetch busque todas as alterações no servidor que você ainda não possui, ele não modificará seu diretório de trabalho. Ele simplesmente obterá os dados para você e permitirá que você mesmo os mescle. No entanto, existe um comando chamado git pull que é essencialmente um git fetch imediatamente seguido por um git merge na maioria dos casos. Se você tiver uma ramificação de rastreamento configurada conforme demonstrado na última seção, configurando-a explicitamente ou criando-a para você pelos comandos clone ou checkout , git pull procurará qual servidor e ramificação sua ramificação atual está rastreando, busque desse servidor e, em seguida, tente mesclar nessa ramificação remota.

Geralmente, é melhor simplesmente usar os comandos de busca e mesclagem explicitamente, pois a mágica do git pull pode ser confusa.

Excluindo filiais remotas

Suponha que você tenha terminado com uma ramificação remota — digamos que você e seus colaboradores terminaram um recurso e o mesclaram na ramificação mestre do seu controle remoto (ou em qualquer ramificação em que sua linha de código estável esteja). Você pode excluir uma ramificação remota usando a opção --delete para git push. Se você deseja excluir sua ramificação serverfix do servidor, execute o seguinte:

```
$ git push origin --delete serverfix
```

Machine Translated by Google

```
Para https://github.com/schacon/simplegit
- [excluído]             serverfix
```

Basicamente, tudo o que isso faz é remover o ponteiro do servidor. O servidor Git geralmente mantém os dados lá por um tempo até que uma coleta de lixo seja executada, portanto, se foi excluído acidentalmente, geralmente é fácil recuperá-los.

Rebase

No Git, existem duas formas principais de integrar as alterações de um branch para outro: o merge e o rebase. Nesta seção, você aprenderá o que é rebase, como fazê-lo, por que é uma ferramenta incrível e em quais casos você não vai querer usá-la.

O rebase básico

Se você voltar a um exemplo anterior do Basic Merging, verá que divergiu seu trabalho e fez commits em duas ramificações diferentes.

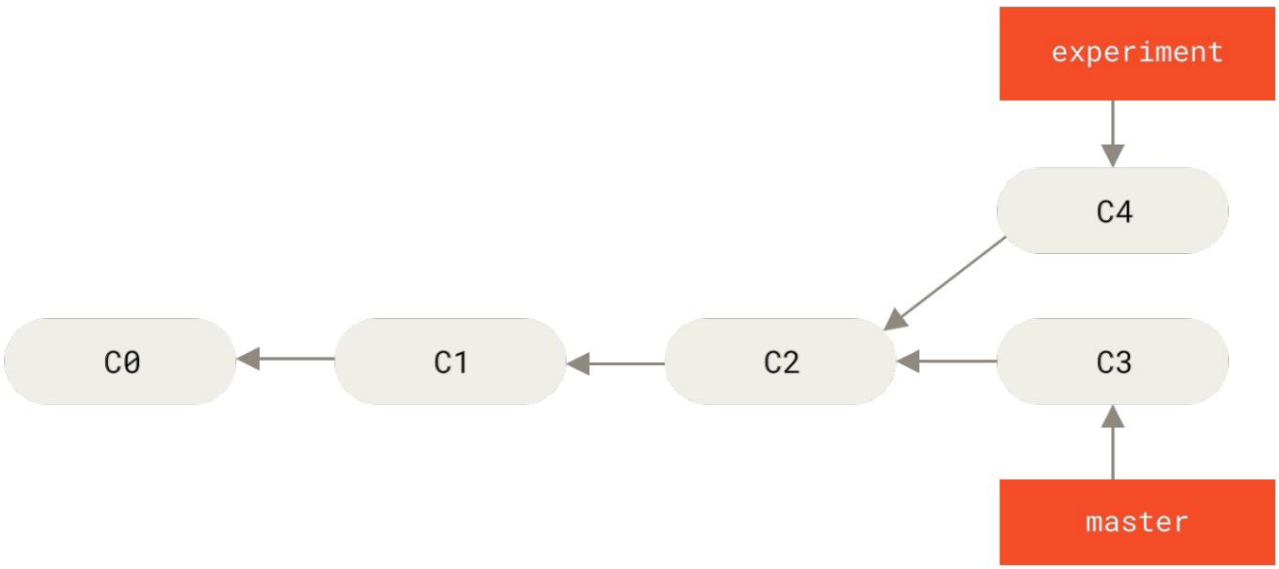


Figura 35. Histórico divergente simples

A maneira mais fácil de integrar as ramificações, como já abordamos, é o comando merge . Ele executa uma mesclagem de três vias entre os dois instantâneos de ramificação mais recentes (C3 e C4) e o ancestral comum mais recente dos dois (C2), criando um novo instantâneo (e confirmação).

Machine Translated by Google

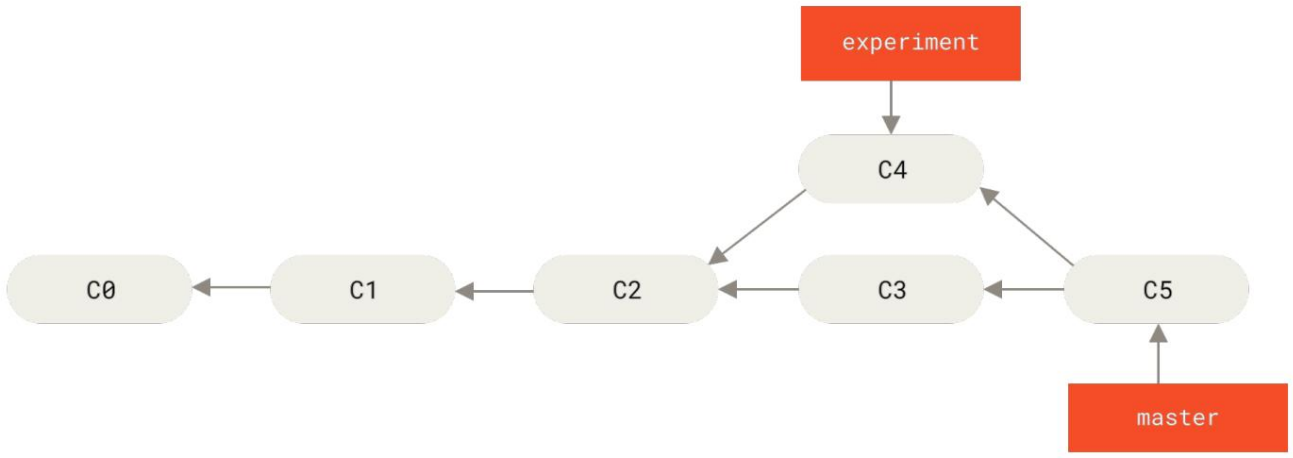


Figura 36. Fusão para integrar histórico de trabalho divergente

No entanto, existe outra maneira: você pode pegar o patch da alteração que foi introduzida no C4 e reaplicá-lo em cima do C3. No Git, isso é chamado de *rebase.* Com o comando rebase , você pode pegar todas as alterações que foram confirmadas em uma ramificação e reproduzi-las em uma ramificação diferente.

Para este exemplo, você verificaria a ramificação experimental e, em seguida, rebaseia-a na ramificação principal da seguinte maneira:

```
$ git checkout experiment $ git
rebase master Primeiro,
rebobinando a cabeça para reproduzir seu trabalho em cima dela...
Aplicando: adicionado comando staged
```

Esta operação funciona indo para o ancestral comum das duas ramificações (aquela em que você está e aquela em que está rebaseando), obtendo o diff introduzido por cada confirmação da ramificação em que você está, salvando essas diferenças em arquivos temporários arquivos, redefinindo a ramificação atual para o mesmo commit da ramificação na qual você está rebaseando e, finalmente, aplicando cada alteração por vez.

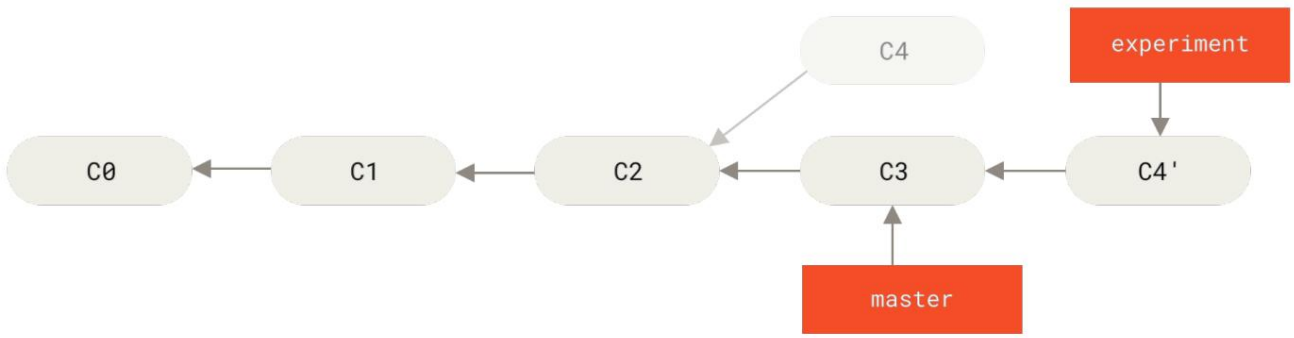


Figura 37. Rebase da alteração introduzida em C4 para C3

Neste ponto, você pode voltar para a ramificação master e fazer uma mesclagem de avanço rápido.

```
$ git checkout master $ git
merge experimento
```

Machine Translated by Google

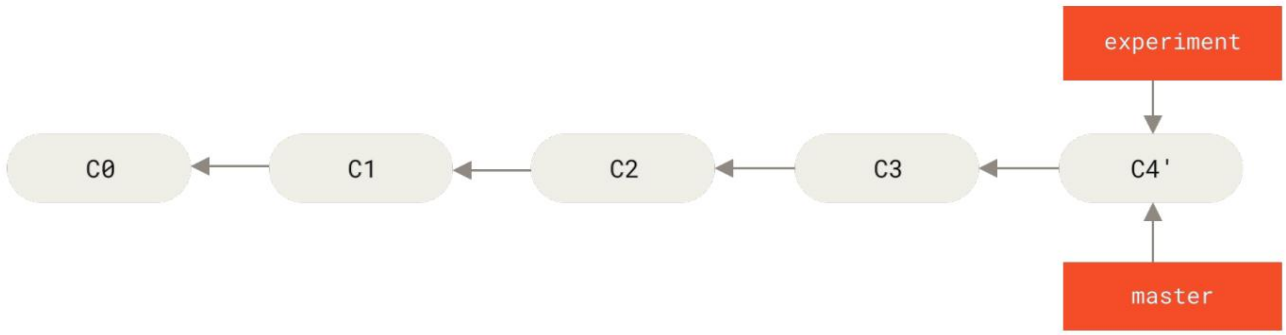


Figura 38. Avanço rápido do branch master

Agora, o snapshot apontado por C4' é exatamente o mesmo que foi apontado por C5 no exemplo de mesclagem. Não há diferença no produto final da integração, mas o rebase torna o histórico mais limpo. Se você examinar o log de um branch rebaseado, ele se parecerá com um histórico linear: parece que todo o trabalho aconteceu em série, mesmo quando originalmente aconteceu em paralelo.

Freqüentemente, você fará isso para garantir que seus commits sejam aplicados corretamente em um branch remoto — talvez em um projeto para o qual você está tentando contribuir, mas que não mantém. Nesse caso, você faria seu trabalho em uma ramificação e, em seguida, faria o rebase de seu trabalho em origin/master quando estivesse pronto para enviar seus patches para o projeto principal. Dessa forma, o mantenedor não precisa fazer nenhum trabalho de integração — apenas um avanço rápido ou uma aplicação limpa.

Observe que o instantâneo apontado pelo commit final com o qual você termina, seja o último dos commits rebaseados para um rebase ou o commit de mesclagem final após uma mesclagem, é o mesmo instantâneo — é apenas o histórico que é diferente. O rebase reproduz as alterações de uma linha de trabalho para outra na ordem em que foram introduzidas, enquanto a mesclagem pega os pontos de extremidade e os mescla.

Rebases mais interessantes

Você também pode ter seu rebase replay em algo diferente da ramificação de destino do rebase. Pegue um histórico como Um histórico com uma ramificação de tópico de outra ramificação de tópico, por exemplo. Você ramifica uma ramificação de tópico (servidor) para adicionar alguma funcionalidade do lado do servidor ao seu projeto e fez um commit. Em seguida, você ramificou isso para fazer as alterações do lado do cliente (cliente) e fez o commit algumas vezes. Por fim, você voltou para a ramificação do servidor e fez mais alguns commits.

Machine Translated by Google

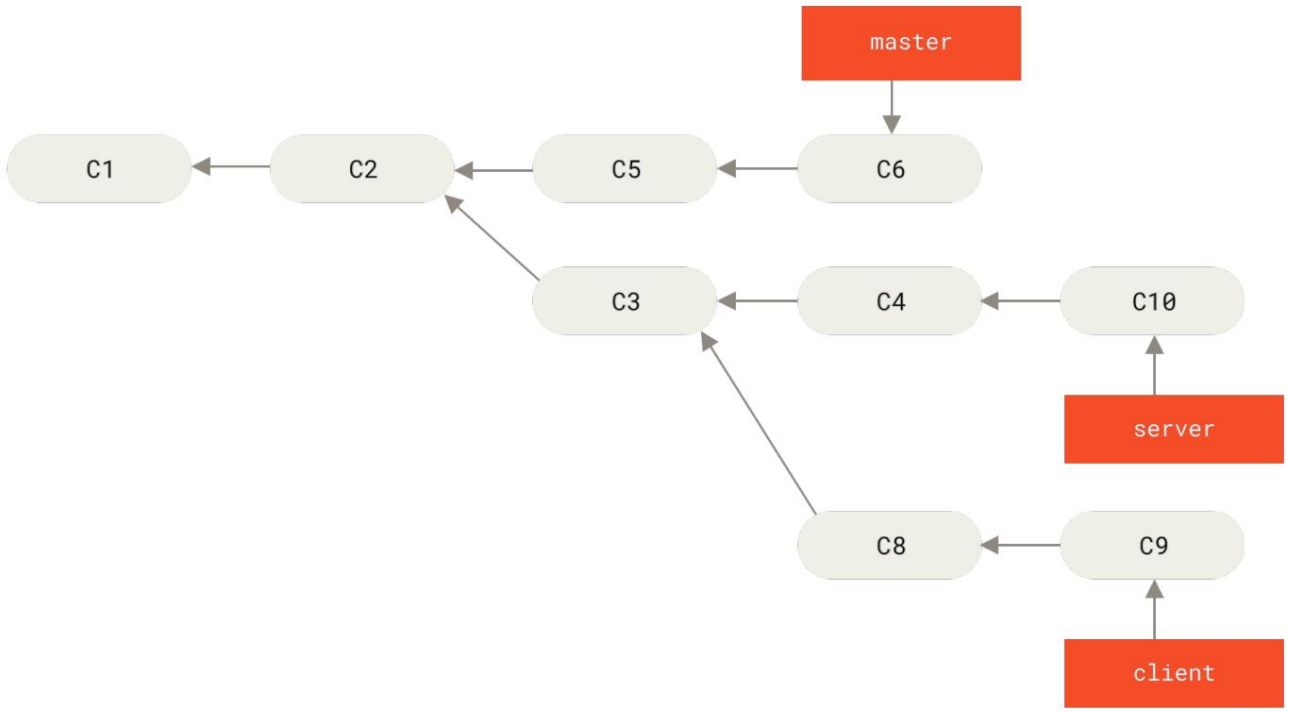


Figura 39. Um histórico com uma ramificação de tópico de outra ramificação de tópico

Suponha que você decida que deseja mesclar as alterações do lado do cliente em sua linha principal para um lançamento, mas deseja adiar as alterações do lado do servidor até que sejam testadas posteriormente. Você pode pegar as alterações no cliente que não estão no servidor (C8 e C9) e reproduzi-las em sua ramificação master usando a opção --onto do git rebase:

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

Isso basicamente diz: "Pegue o branch do cliente , descubra os patches desde que ele divergiu do branch do servidor e reproduza esses patches no branch do cliente como se fossem baseados diretamente no branch master ". É um pouco complexo, mas o resultado é bem legal.

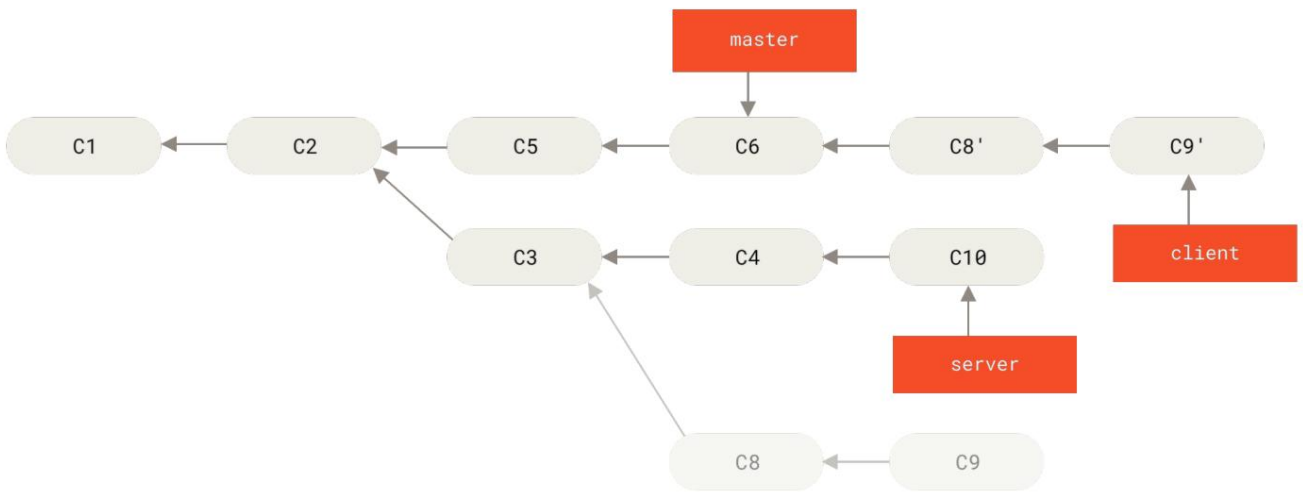


Figura 40. Rebaseando uma ramificação de tópico de outra ramificação de tópico

Agora você pode avançar rapidamente seu branch master (consulte Avanço rápido de seu branch master para incluir as alterações do branch do cliente):

Machine Translated by Google

```
$ git checkout master
$ git merge cliente
```

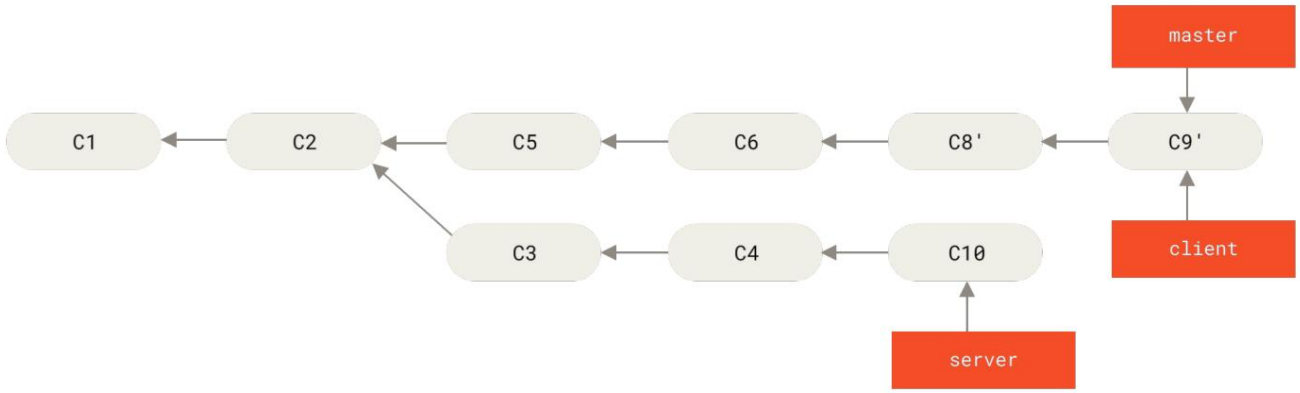


Figura 41. Avanço rápido de sua ramificação principal *para incluir as alterações da ramificação do cliente*

Digamos que você também decida puxar sua ramificação de servidor. Você pode rebasear o branch do servidor no branch master sem ter que fazer check-out primeiro executando git rebase <basebranch> <topicbranch> — que verifica o branch do tópico (neste caso, servidor) para você e o repete no branch base (mestre):

```
$ git rebase servidor mestre
```

Isso reproduz seu trabalho de servidor em cima de seu trabalho principal , conforme mostrado em Rebaseando sua ramificação de servidor em cima de sua ramificação principal .

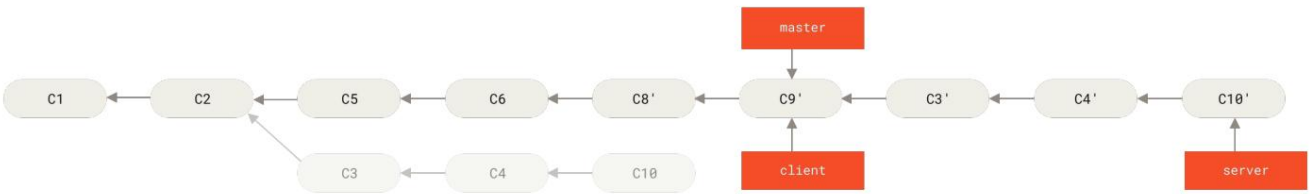


Figura 42. Rebaseando sua ramificação de servidor sobre sua ramificação principal

Em seguida, você pode avançar a ramificação base (mestre):

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

Você pode remover as ramificações do cliente e do servidor porque todo o trabalho é integrado e você não precisa mais deles, deixando seu histórico para todo esse processo com a aparência de Final commit history:

```
$ git branch -d cliente
$ git branch -d servidor
```

Machine Translated by Google

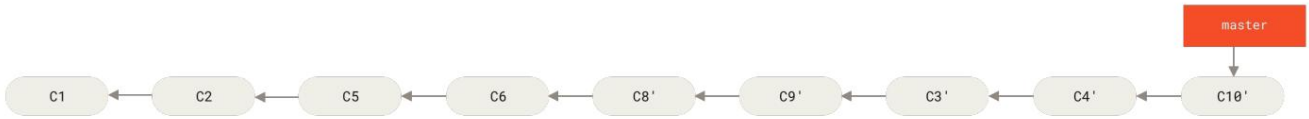


Figura 43. Histórico de confirmação final

Os perigos do rebase

Ahh, mas a felicidade do rebase tem suas desvantagens, que podem ser resumidas em uma única linha:

Não rebase commits que existam fora do seu repositório e nos quais as pessoas possam ter baseado o trabalho.

Se você seguir essa diretriz, ficará bem. Se não o fizer, as pessoas vão odiá-lo e você será desprezado por amigos e familiares.

Quando você rebase coisas, você está abandonando os commits existentes e criando novos que são semelhantes, mas diferentes. Se você enviar commits para algum lugar e outros os puxarem para baixo e basearem o trabalho neles, e então você reescrever esses commits com git rebase e enviá-los novamente, seus colaboradores terão que mesclar novamente seus trabalhos e as coisas ficarão confusas quando você tentar puxe o trabalho deles de volta para o seu.

Vejamos um exemplo de como o trabalho de rebase que você tornou público pode causar problemas.
Suponha que você clone de um servidor central e faça algum trabalho com isso. Seu histórico de commits se parece com isso:

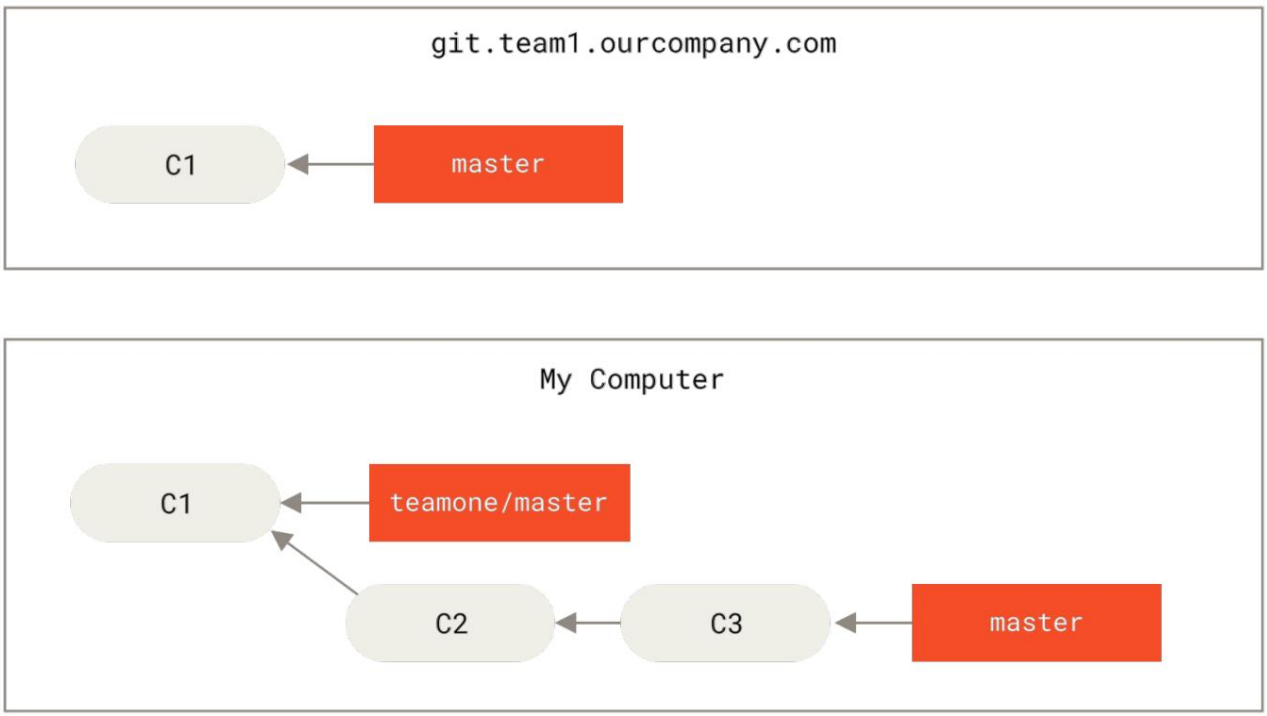


Figura 44. Clone um repositório e baseie algum trabalho nele

Agora, outra pessoa faz mais trabalho que inclui uma mesclagem e envia esse trabalho para o servidor central. Você o busca e faz o merge do novo branch remoto em seu trabalho, fazendo com que seu histórico fique mais ou menos assim:

Machine Translated by Google

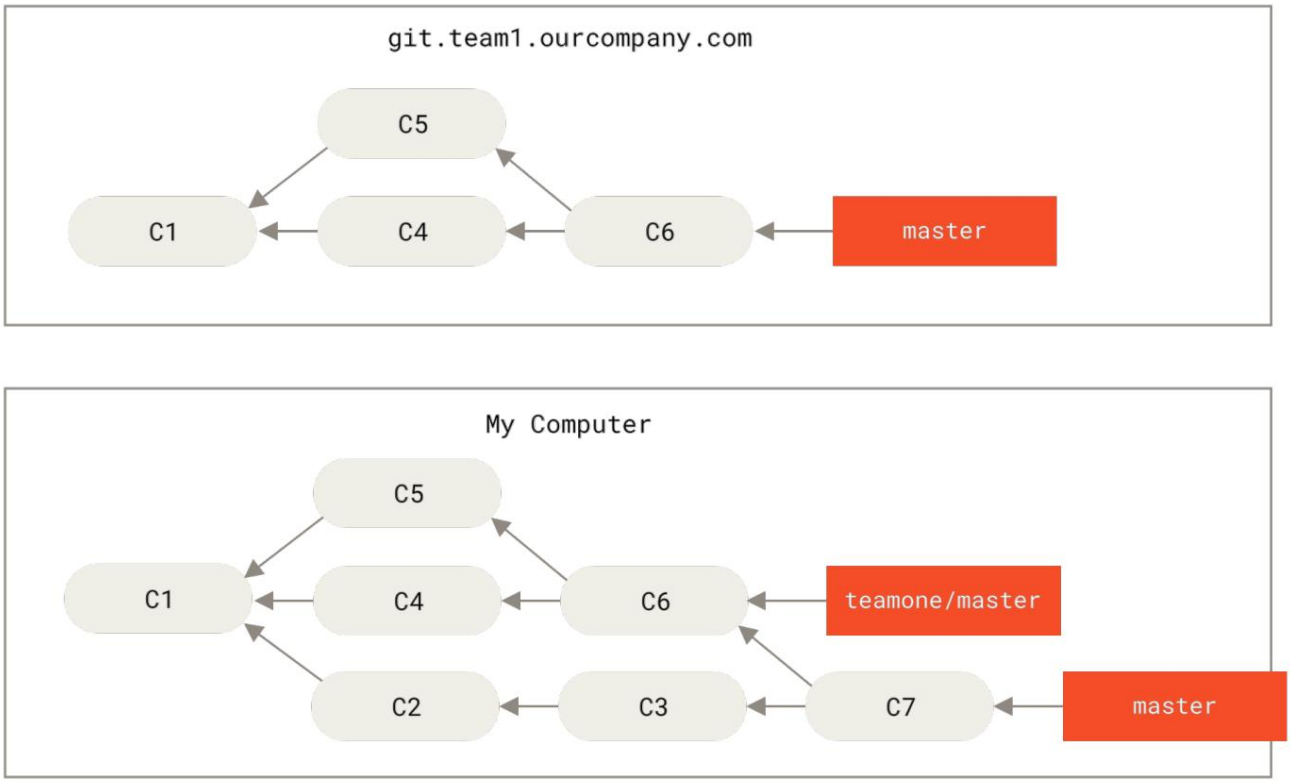


Figura 45. Busque mais commits e mescle-os em seu trabalho

Em seguida, a pessoa que enviou o trabalho mesclado decide voltar e rebasear seu trabalho; eles fazem um git push --force para sobrescrever o histórico no servidor. Você então busca a partir desse servidor, derrubando os novos commits.

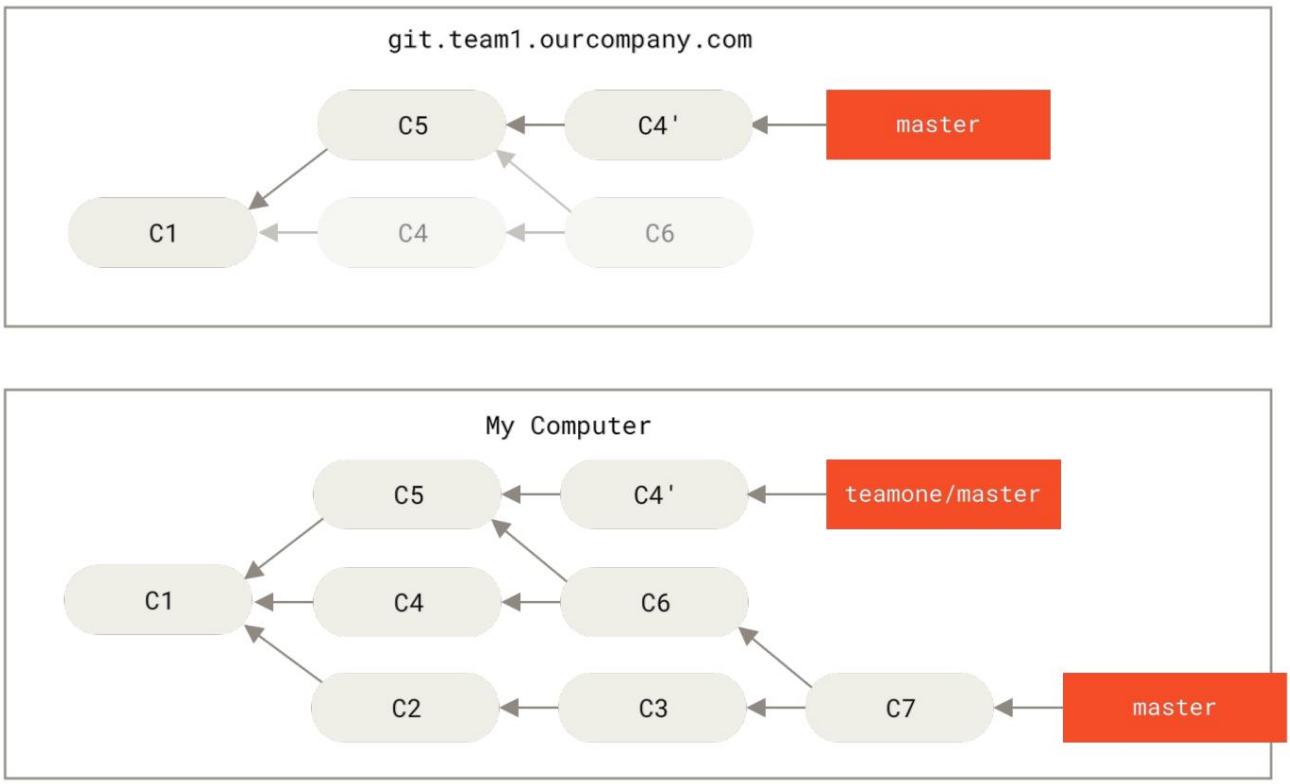


Figura 46. Alguém envia commits rebaseados, abandonando commits nos quais você baseou seu trabalho

Machine Translated by Google

Agora vocês dois estão em apuros. Se você fizer um git pull, criará um merge commit que inclui ambas as linhas do histórico, e seu repositório ficará assim:

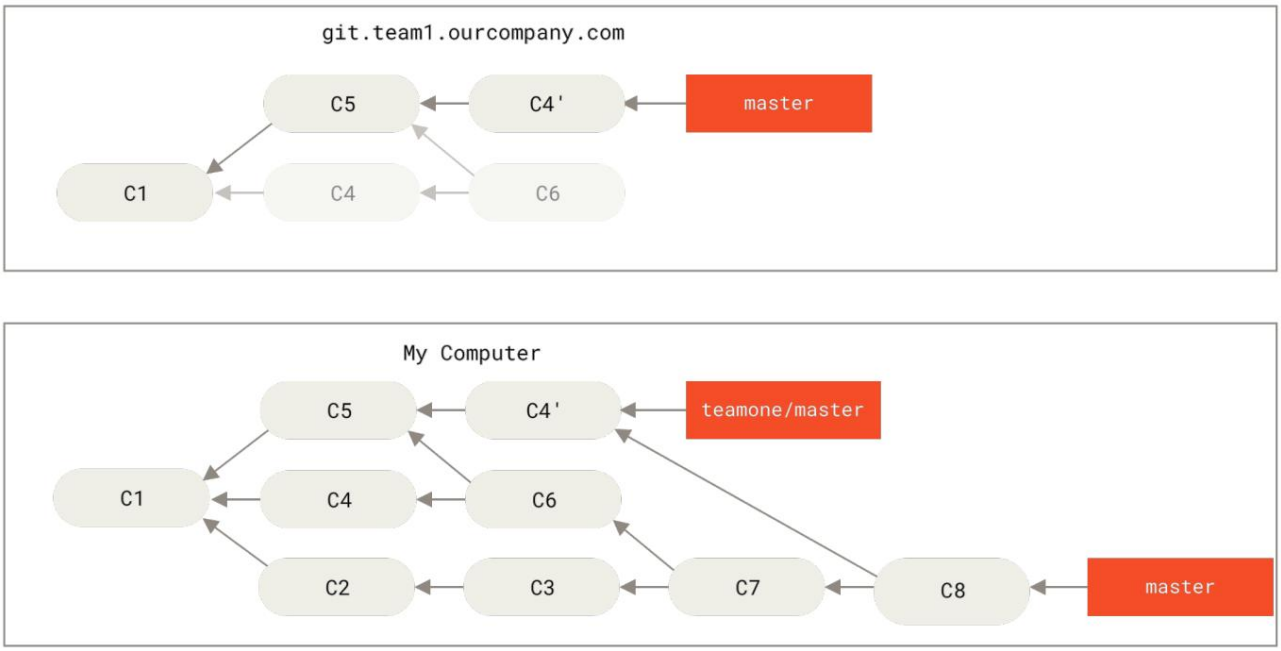


Figura 47. Você mescla no mesmo trabalho novamente em um novo commit de mesclagem

Se você executar um git log quando seu histórico estiver assim, verá dois commits com o mesmo autor, data e mensagem, o que será confuso. Além disso, se você enviar esse histórico de volta para o servidor, reintroduzirá todos os commits rebaseados no servidor central, o que pode confundir ainda mais as pessoas. É bastante seguro presumir que o outro desenvolvedor não deseja que C4 e C6 estejam no histórico; é por isso que eles se mudaram em primeiro lugar.

Rebase quando você rebase

Se você **se** encontrar em uma situação como essa, o Git tem mais alguma mágica que pode ajudá-lo. Se alguém da sua equipe forçar alterações que substituem o trabalho no qual você se baseou, seu desafio é descobrir o que é seu e o que eles reescreveram.

Acontece que, além da soma de verificação SHA-1 do commit, o Git também calcula uma soma de verificação baseada apenas no patch introduzido com o commit. Isso é chamado de “id do patch”.

Se você puxar para baixo o trabalho que foi reescrito e rebaseá-lo em cima dos novos commits de seu parceiro, o Git pode frequentemente descobrir com sucesso o que é exclusivamente seu e aplicá-los de volta no topo do novo branch.

Por exemplo, no cenário anterior, se em vez de fazer uma mesclagem quando estamos em Alguém envia commits rebaseados, abandonando os commits nos quais você baseou seu trabalho, executamos git rebase teamone/master, o Git irá:

- Determinar qual trabalho é exclusivo de nossa filial (C2, C3, C4, C6, C7)
- Determinar quais não são commits de mesclagem (C2, C3, C4)
- Determine quais não foram reescritos na ramificação de destino (apenas C2 e C3, já que C4 é o mesmo patch como C4')

Machine Translated by Google

- Aplicar esses commits no topo de teamone/master

Portanto, em vez do resultado que vemos em You merge no mesmo trabalho novamente em um novo merge commit, teríamos algo mais como Rebase no topo do trabalho de rebase forçado.

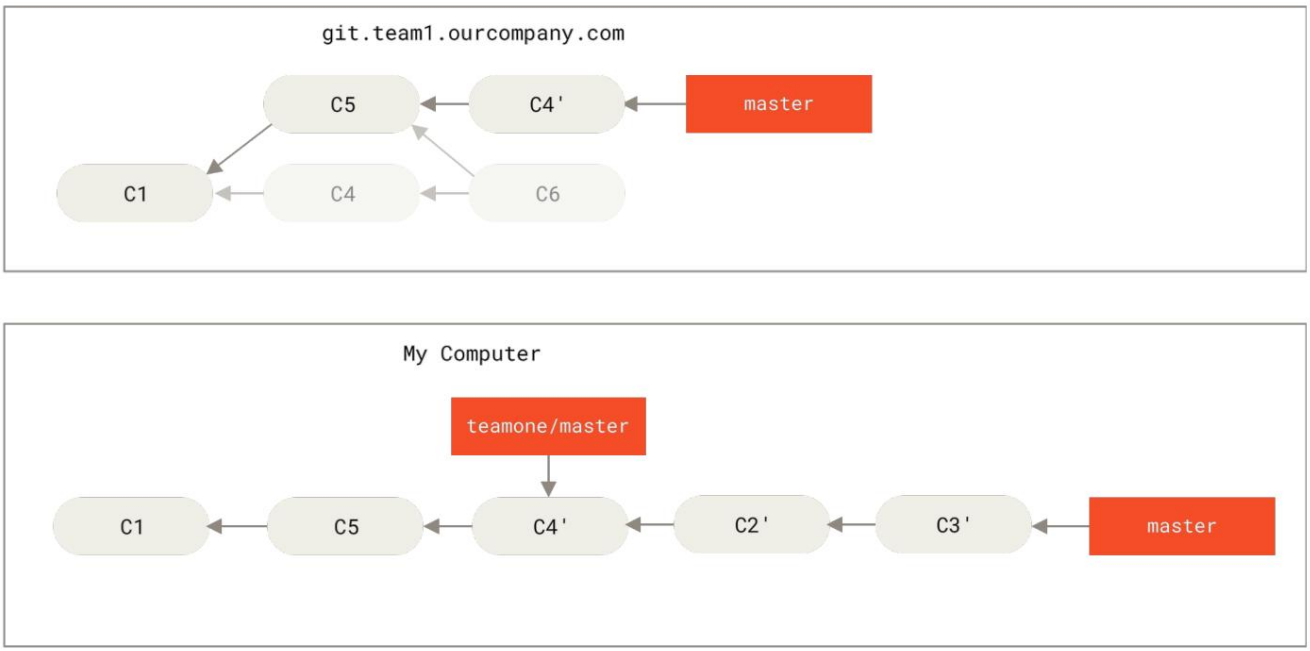


Figura 48. Rebase sobre o trabalho de rebase forçado

Isso só funciona se C4 e C4' que seu parceiro fez forem quase exatamente o mesmo patch. Caso contrário, o rebase não será capaz de dizer que é uma duplicata e adicionará outro patch semelhante ao C4 (que provavelmente não será aplicado de forma limpa, pois as alterações já estariam pelo menos um pouco lá).

Você também pode simplificar isso executando um git pull --rebase em vez de um git pull normal . Ou você pode fazer isso manualmente com um git fetch seguido por um git rebase teamone/master neste caso.

Se você estiver usando git pull e quiser fazer --rebase o padrão, você pode definir o valor de configuração pull.rebase com algo como git config --global pull.rebase true.

Se você apenas rebase commits que nunca saíram do seu computador, você ficará bem. Se você rebase commits que foram enviados, mas dos quais ninguém mais baseou commits, você também ficará bem. Se você rebase commits que já foram enviados publicamente, e as pessoas podem ter baseado o trabalho nesses commits, então você pode ter alguns problemas frustrantes, e o desprezo de seu companheiros de equipe.

Se você ou um parceiro achar necessário em algum momento, certifique-se de que todos saibam executar git pull --rebase para tentar tornar a dor depois que ela acontecer um pouco mais simples.

Rebase vs. Mesclar

Agora que você viu o rebase e a fusão em ação, pode estar se perguntando qual é o melhor.
Antes de podermos responder a isso, vamos recuar um pouco e falar sobre o que a história significa.

Um ponto de vista sobre isso é que o histórico de commits do seu repositório é um **registro do que realmente aconteceu.** É um documento histórico, valioso por si só, e não deve ser adulterado.

Machine Translated by Google

Desse ângulo, mudar o histórico de commits é quase uma blasfêmia; você está *mentindo* sobre o que realmente aconteceu. E se houvesse uma série confusa de commits de merge? Foi assim que aconteceu, e o repositório deve preservar isso para a posteridade.

O ponto de vista oposto é que o histórico de commits é a **história de como seu projeto foi feito.** Você não publicaria o primeiro rascunho de um livro, então por que mostrar seu trabalho confuso? Quando você está trabalhando em um projeto, pode precisar de um registro de todos os seus erros e caminhos sem saída, mas quando é hora de mostrar seu trabalho para o mundo, você pode querer contar uma história mais coerente de como chegar de A a B. As pessoas neste campo usam ferramentas como rebase e filter-branch para reescrever seus commits antes de serem mesclados no ramo principal. Eles usam ferramentas como rebase e filter-branch, para contar a história da maneira que for melhor para futuros leitores.

Agora, para a questão de se mesclar ou rebasear é melhor: espero que você veja que não é tão simples. Git é uma ferramenta poderosa e permite que você faça muitas coisas para e com seu histórico, mas cada equipe e cada projeto é diferente. Agora que você sabe como essas duas coisas funcionam, cabe a você decidir qual é a melhor para sua situação específica.

Você pode obter o melhor dos dois mundos: rebase as alterações locais antes de enviar para limpar seu trabalho, mas nunca rebase qualquer coisa que você tenha enviado para algum lugar.

Resumo

Cobrimos ramificações e mesclagens básicas no Git. Você deve se sentir confortável criando e alternando para novos branches, alternando entre branches e mesclando branches locais. Você também deve ser capaz de compartilhar suas ramificações enviando-as para um servidor compartilhado, trabalhando com outras pessoas em ramificações compartilhadas e rebaseando suas ramificações antes de serem compartilhadas. Em seguida, abordaremos o que você precisará para executar seu próprio servidor de hospedagem de repositório Git.

Machine Translated by Google

Git no servidor

Neste ponto, você deve ser capaz de realizar a maioria das tarefas diárias para as quais usará o Git. No entanto, para fazer qualquer colaboração no Git, você precisará ter um repositório Git remoto. Embora você possa tecnicamente enviar e extrair alterações de repositórios individuais, isso é desencorajado porque você pode facilmente confundir o que eles estão trabalhando se não for cuidadoso. Além disso, você deseja que seus colaboradores possam acessar o repositório mesmo que seu computador esteja offline — geralmente é útil ter um repositório comum mais confiável. Portanto, o método preferido para colaborar com alguém é configurar um repositório intermediário ao qual vocês dois tenham acesso e fazer push e pull dele.

Executar um servidor Git é bastante simples. Primeiro, você escolhe quais protocolos deseja que seu servidor suporte. A primeira seção deste capítulo cobrirá os protocolos disponíveis e os prós e contras de cada um. As próximas seções explicarão algumas configurações típicas usando esses protocolos e como fazer seu servidor rodar com eles. Por último, examinaremos algumas opções hospedadas, se você não se importar em hospedar seu código no servidor de outra pessoa e não quiser passar pelo incômodo de configurar e manter seu próprio servidor.

Se você não tem interesse em executar seu próprio servidor, pode pular para a última seção do capítulo para ver algumas opções para configurar uma conta hospedada e, em seguida, passar para o próximo capítulo, onde discutimos os vários prós e contras do trabalho. em um ambiente de controle de origem distribuído.

Um repositório remoto geralmente é um *repositório simples* — um repositório Git que não possui diretório de trabalho. Como o repositório é usado apenas como um ponto de colaboração, não há motivo para fazer check-out de um instantâneo no disco; são apenas os dados do Git. Em termos simples, um repositório simples é o conteúdo do diretório .git do seu projeto e nada mais.

Os protocolos

O Git pode usar quatro protocolos distintos para transferir dados: Local, HTTP, Secure Shell (SSH) e Git. Aqui discutiremos o que são e em quais circunstâncias básicas você gostaria (ou não) de usá-los.

Protocolo local

O mais básico é o *protocolo Local,* no qual o repositório remoto está em outro diretório no mesmo host. Isso geralmente é usado se todos em sua equipe tiverem acesso a um sistema de arquivos compartilhado, como um NFS mount, ou no caso menos provável de todos logarem no mesmo computador. O último não seria o ideal, porque todas as suas instâncias de repositório de código residiriam no mesmo computador, tornando muito mais provável uma perda catastrófica.

Se você tiver um sistema de arquivos montado compartilhado, poderá clonar, enviar e extrair de um repositório baseado em arquivo local. Para clonar um repositório como este ou adicionar um como remoto a um projeto existente, use o caminho para o repositório como URL. Por exemplo, para clonar um repositório local, você pode executar algo assim:

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

Machine Translated by Google

Ou você pode fazer isso:

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

O Git opera de maneira ligeiramente diferente se você especificar explicitamente file:// no início da URL. Se você apenas especificar o caminho, o Git tentará usar hardlinks ou copiar diretamente os arquivos necessários. Se você especificar file://, o Git acionará os processos que normalmente usa para transferir dados por uma rede, o que geralmente é muito menos eficiente. O principal motivo para especificar o prefixo file:// é se você deseja uma cópia limpa do repositório com referências ou objetos estranhos deixados de fora — geralmente após uma importação de outro VCS ou algo semelhante (consulte Git Internals para tarefas de manutenção). Usaremos o caminho normal aqui porque fazer isso é quase sempre mais rápido.

Para adicionar um repositório local a um projeto Git existente, você pode executar algo assim:

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

Em seguida, você pode enviar e receber desse controle remoto por meio de seu novo nome remoto local_proj como se estivesse fazendo isso em uma rede.

Os prós

As vantagens dos repositórios baseados em arquivos são que eles são simples e usam permissões de arquivos existentes e acesso à rede. Se você já possui um sistema de arquivos compartilhado ao qual toda a sua equipe tem acesso, configurar um repositório é muito fácil. Você coloca a cópia do repositório simples em algum lugar ao qual todos tenham acesso compartilhado e define as permissões de leitura/gravação como faria para qualquer outro diretório compartilhado.
Discutiremos como exportar uma cópia simples do repositório para essa finalidade em Obtendo o Git em um servidor.

Essa também é uma boa opção para obter rapidamente o trabalho do repositório de trabalho de outra pessoa. Se você e um colega de trabalho estiverem trabalhando no mesmo projeto e eles quiserem que você verifique algo, executar um comando como git pull /home/john/project geralmente é mais fácil do que enviar para um servidor remoto e você subseqüentemente buscar dele .

Os contras

As desvantagens desse método são que o acesso compartilhado geralmente é mais difícil de configurar e acessar de vários locais do que o acesso básico à rede. Se você quiser enviar do seu laptop quando estiver em casa, precisará montar o disco remoto, o que pode ser difícil e lento em comparação com o acesso baseado em rede.

É importante mencionar que esta não é necessariamente a opção mais rápida se você estiver usando algum tipo de montagem compartilhada. Um repositório local é rápido apenas se você tiver acesso rápido aos dados. Um repositório em NFS geralmente é mais lento que o repositório em SSH no mesmo servidor, permitindo que o Git execute discos locais em cada sistema.

Finalmente, este protocolo não protege o repositório contra danos acidentais. Todo usuário tem acesso shell completo ao diretório “remoto” e não há nada que os impeça de alterar ou remover arquivos internos do Git e corromper o repositório.

Machine Translated by Google

Os protocolos HTTP

O Git pode se comunicar por HTTP usando dois modos diferentes. Antes do Git 1.6.6, havia apenas uma maneira de fazer isso, que era muito simples e geralmente somente leitura. Na versão 1.6.6, foi introduzido um protocolo novo e mais inteligente que envolvia o Git sendo capaz de negociar de forma inteligente a transferência de dados de maneira semelhante à que ele faz por SSH. Nos últimos anos, este novo protocolo HTTP tornou-se muito popular, pois é mais simples para o usuário e mais inteligente na forma como se comunica. A versão mais recente costuma ser chamada de protocolo *Smart* HTTP e a mais antiga de *Dumb* HTTP. Abordaremos primeiro o protocolo Smart HTTP mais recente.

HTTP inteligente

O Smart HTTP opera de maneira muito semelhante aos protocolos SSH ou Git, mas é executado em portas HTTPS padrão e pode usar vários mecanismos de autenticação HTTP, o que significa que geralmente é mais fácil para o usuário do que algo como SSH, pois você pode usar coisas como autenticação de nome de usuário/senha em vez de ter para configurar as chaves SSH.

Provavelmente se tornou a maneira mais popular de usar o Git agora, pois pode ser configurado para servir anonimamente como o protocolo git:// e também pode ser enviado com autenticação e criptografia como o protocolo SSH. Em vez de configurar diferentes URLs para essas coisas, agora você pode usar um único URL para ambos. Se você tentar enviar e o repositório exigir autenticação (o que normalmente deveria), o servidor poderá solicitar um nome de usuário e senha. O mesmo vale para ler

acesso.

Na verdade, para serviços como o GitHub, a URL que você usa para visualizar o repositório online (por exemplo, https://github.com/schacon/simplegit) é a mesma URL que você pode usar para clonar e, se tiver acesso, enviar.

HTTP estúpido

Se o servidor não responder com um serviço inteligente Git HTTP, o cliente Git tentará recorrer ao protocolo *Dumb* HTTP mais simples. O protocolo Dumb espera que o repositório Git vazio seja servido como arquivos normais do servidor web. A beleza do Dumb HTTP é a simplicidade de configurá-lo.
Basicamente, tudo o que você precisa fazer é colocar um repositório Git simples sob a raiz do documento HTTP e configurar um gancho de pós-atualização específico e pronto (consulte Ganchos do Git). Nesse ponto, qualquer pessoa que possa acessar o servidor web no qual você colocou o repositório também pode clonar seu repositório. Para permitir o acesso de leitura ao seu repositório por HTTP, faça algo assim:

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

Isso é tudo. O gancho pós-atualização que vem com o Git por padrão executa o comando apropriado (git update-server-info) para fazer a busca e clonagem de HTTP funcionar corretamente. Este comando é executado quando você envia para este repositório (talvez por SSH); então, outras pessoas podem clonar através de algo como:

Machine Translated by Google

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

Neste caso particular, estamos usando o caminho /var/www/htdocs que é comum para configurações do Apache, mas você pode usar qualquer servidor web estático — basta colocar o repositório vazio em seu caminho. Os dados do Git são servidos como arquivos estáticos básicos (consulte o capítulo Git Internals para obter detalhes sobre exatamente como eles são servidos).

Geralmente, você escolheria executar um servidor Smart HTTP de leitura/gravação ou simplesmente ter os arquivos acessíveis como somente leitura da maneira burra. É raro executar uma combinação dos dois serviços.

Os prós

Vamos nos concentrar nos profissionais da versão Smart do protocolo HTTP.

A simplicidade de ter uma única URL para todos os tipos de acesso e ter o prompt do servidor apenas quando a autenticação é necessária torna as coisas muito fáceis para o usuário final. Ser capaz de autenticar com um nome de usuário e senha também é uma grande vantagem sobre o SSH, pois os usuários não precisam gerar chaves SSH localmente e fazer upload de sua chave pública para o servidor antes de poder interagir com ele. Para usuários menos sofisticados, ou usuários em sistemas onde o SSH é menos comum, esta é uma grande vantagem em usabilidade. Também é um protocolo muito rápido e eficiente, semelhante ao SSH.

Você também pode servir seus repositórios somente leitura por HTTPS, o que significa que você pode criptografar a transferência de conteúdo; ou você pode fazer com que os clientes usem certificados SSL assinados específicos.

Outra coisa boa é que HTTP e HTTPS são protocolos tão comumente usados que os firewalls corporativos costumam ser configurados para permitir o tráfego por meio de suas portas.

Os contras

O Git over HTTPS pode ser um pouco mais complicado de configurar em comparação com o SSH em alguns servidores. Fora isso, há muito pouca vantagem que outros protocolos têm sobre o Smart HTTP para servir o Git conteúdo.

Se você estiver usando HTTP para push autenticado, fornecer suas credenciais às vezes é mais complicado do que usar chaves por SSH. Existem, no entanto, várias ferramentas de cache de credenciais que você pode usar, incluindo o acesso às chaves no macOS e o Credential Manager no Windows, para tornar isso bastante indolor. Leia Credential Storage para ver como configurar o cache seguro de senha HTTP em seu sistema.

O protocolo SSH

Um protocolo de transporte comum para Git quando a auto-hospedagem é via SSH. Isso ocorre porque o acesso SSH aos servidores já está configurado na maioria dos lugares — e se não estiver, é fácil de fazer. O SSH também é um protocolo de rede autenticado e, por ser onipresente, geralmente é fácil de configurar e usar.

Para clonar um repositório Git por SSH, você pode especificar uma URL ssh:// como esta:

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

Machine Translated by Google

Ou você pode usar a sintaxe mais curta do tipo scp para o protocolo SSH:

```
$ git clone [usuário@]servidor:projeto.git
```

Em ambos os casos acima, se você não especificar o nome de usuário opcional, o Git assumirá o usuário com o qual você está conectado no momento.

Os prós

Os prós de usar o SSH são muitos. Primeiro, o SSH é relativamente fácil de configurar — os daemons SSH são comuns, muitos administradores de rede têm experiência com eles e muitas distribuições de SO são configuradas com eles ou possuem ferramentas para gerenciá-los. Em seguida, o acesso por SSH é seguro — toda a transferência de dados é criptografada e autenticada. Por fim, como os protocolos HTTPS, Git e Local, o SSH é eficiente, tornando os dados o mais compactos possível antes de transferi-los.

Os contras

O aspecto negativo do SSH é que ele não suporta acesso anônimo ao seu repositório Git. Se você estiver usando SSH, as pessoas *devem* ter acesso SSH à sua máquina, mesmo em uma capacidade somente leitura, o que não torna o SSH propício para projetos de código aberto para os quais as pessoas podem simplesmente querer clonar seu repositório para examiná-lo. Se você o estiver usando apenas em sua rede corporativa, o SSH pode ser o único protocolo com o qual você precisa lidar. Se você deseja permitir acesso anônimo somente leitura aos seus projetos e também deseja usar o SSH, precisará configurar o SSH para enviar, mas outra coisa para outros buscarem.

O Protocolo Git

Finalmente, temos o protocolo Git. Este é um daemon especial que vem empacotado com o Git; ele escuta em uma porta dedicada (9418) que fornece um serviço semelhante ao protocolo SSH, mas sem nenhuma autenticação. Para que um repositório seja servido pelo protocolo Git, você deve criar um arquivo git-daemon-export-ok — o daemon não servirá a um repositório sem esse arquivo — mas, fora isso, não há segurança . O repositório Git está disponível para todos clonarem ou não.

Isso significa que geralmente não há push neste protocolo. Você pode ativar o acesso por push, mas, devido à falta de autenticação, qualquer pessoa na Internet que encontrar a URL do seu projeto poderá enviar por push para esse projeto. Basta dizer que isso é raro.

Os prós

O protocolo Git geralmente é o protocolo de transferência de rede mais rápido disponível. Se você estiver atendendo a muito tráfego para um projeto público ou a um projeto muito grande que não requer autenticação do usuário para acesso de leitura, é provável que você queira configurar um daemon Git para atender ao seu projeto. Ele usa o mesmo mecanismo de transferência de dados do protocolo SSH, mas sem a sobrecarga de criptografia e autenticação.

Os contras

A desvantagem do protocolo Git é a falta de autenticação. Geralmente, é indesejável que o protocolo Git seja o único acesso ao seu projeto. Geralmente, você irá emparelhá-lo com acesso SSH ou HTTPS para

Machine Translated by Google

os poucos desenvolvedores que têm acesso push (gravação) e todos os outros usam git:// para acesso somente leitura. Também é provavelmente o protocolo mais difícil de configurar. Ele deve executar seu próprio daemon, que requer configuração xinetd ou systemd ou algo semelhante, o que nem sempre é um passeio no parque. Ele também requer acesso de firewall à porta 9418, que não é uma porta padrão que os firewalls corporativos sempre permitem. Atrás de grandes firewalls corporativos, essa porta obscura é comumente bloqueada.

Obtendo o Git em um servidor

Agora abordaremos a configuração de um serviço Git executando esses protocolos em seu próprio servidor.

ÿ

Aqui demonstraremos os comandos e as etapas necessárias para fazer instalações básicas e simplificadas em um servidor baseado em Linux, embora também seja possível executar esses serviços em servidores macOS ou Windows. Na verdade, a configuração de um servidor de produção em sua infraestrutura certamente acarretará diferenças nas medidas de segurança ou nas ferramentas do sistema operacional, mas esperamos que isso lhe dê uma ideia geral do que está envolvido.

Para configurar inicialmente qualquer servidor Git, você precisa exportar um repositório existente para um novo repositório simples — um repositório que não contém um diretório de trabalho. Isso geralmente é simples de fazer. Para clonar seu repositório para criar um novo repositório simples, execute o comando clone com a opção --bare . Por convenção, nomes de diretórios de repositório simples terminam com o sufixo .git, assim:

```
$ git clone --bare my_project my_project.git
Clonagem no repositório simples 'my_project.git'...
pronto.
```

Agora você deve ter uma cópia dos dados do diretório Git em seu diretório my_project.git .

Isso é aproximadamente equivalente a algo como:

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

Existem algumas pequenas diferenças no arquivo de configuração, mas, para o seu propósito, isso é quase a mesma coisa. Ele pega o repositório Git sozinho, sem um diretório de trabalho, e cria um diretório especificamente para ele.

Colocando o repositório vazio em um servidor

Agora que você tem uma cópia simples do seu repositório, tudo que você precisa fazer é colocá-lo em um servidor e configurar seus protocolos. Digamos que você configurou um servidor chamado git.example.com ao qual você tem acesso SSH e deseja armazenar todos os seus repositórios Git no diretório /srv/git . Supondo que /srv/git exista nesse servidor, você pode configurar seu novo repositório copiando seu repositório vazio
sobre:

Machine Translated by Google

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

Neste ponto, outros usuários que têm acesso de leitura baseado em SSH ao diretório /srv/git nesse servidor podem clonar seu repositório executando:

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

Se um usuário fizer SSH em um servidor e tiver acesso de gravação ao diretório /srv/git/my_project.git , ele também terá acesso push automaticamente.

O Git adicionará automaticamente permissões de gravação de grupo a um repositório corretamente se você executar o comando git init com a opção --shared . Observe que, ao executar este comando, você não destruirá nenhum commit, refs, etc. no processo.

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

Você vê como é fácil pegar um repositório Git, criar uma versão básica e colocá-la em um servidor ao qual você e seus colaboradores tenham acesso SSH. Agora você está pronto para colaborar no mesmo projeto.

É importante observar que isso é literalmente tudo o que você precisa fazer para executar um servidor Git útil ao qual várias pessoas tenham acesso - basta adicionar contas compatíveis com SSH em um servidor e colocar um repositório simples em algum lugar onde todos esses usuários tenham lido e gravado acesso a. Você está pronto para ir - nada mais é necessário.

Nas próximas seções, você verá como expandir para configurações mais sofisticadas. Esta discussão incluirá não ter que criar contas de usuário para cada usuário, adicionar acesso público de leitura a repositórios, configurar UIs da Web e muito mais. No entanto, lembre-se de que, para colaborar com algumas pessoas em um projeto privado, tudo o que você *precisa* é de um servidor SSH e um repositório simples.

Pequenas configurações

Se você é um pequeno grupo ou está apenas experimentando o Git em sua organização e tem apenas alguns desenvolvedores, as coisas podem ser simples para você. Um dos aspectos mais complicados da configuração de um servidor Git é o gerenciamento de usuários. Se você deseja que alguns repositórios sejam somente leitura para determinados usuários e leitura/gravação para outros, o acesso e as permissões podem ser um pouco mais difíceis de organizar.

Acesso SSH

Se você tiver um servidor ao qual todos os seus desenvolvedores já tenham acesso SSH, geralmente é mais fácil configurar seu primeiro repositório lá, porque você quase não precisa fazer nenhum trabalho (como abordamos na última seção). Se você deseja permissões de tipo de controle de acesso mais complexas em seus repositórios, pode manipulá-las com as permissões normais do sistema de arquivos do sistema operacional do seu servidor.

Machine Translated by Google

Se você deseja colocar seus repositórios em um servidor que não possui contas para todos em sua equipe para os quais deseja conceder acesso de gravação, você deve configurar o acesso SSH para eles. Assumimos que, se você tiver um servidor para fazer isso, já possui um servidor SSH instalado e é assim que está acessando o servidor.

Existem algumas maneiras de dar acesso a todos em sua equipe. A primeira é configurar contas para todos, o que é simples, mas pode ser complicado. Você pode não querer executar adduser (ou a possível alternativa useradd) e ter que definir senhas temporárias para cada novo usuário.

Um segundo método é criar uma única conta de usuário 'git' na máquina, pedir a cada usuário que tenha acesso de gravação para lhe enviar uma chave pública SSH e adicionar essa chave ao arquivo ~/.ssh/authorized_keys desse novo conta 'git'. Nesse ponto, todos poderão acessar essa máquina por meio da conta 'git'. Isso não afeta os dados do commit de forma alguma — o usuário SSH com o qual você se conecta não afeta os commits que você registrou.

Outra maneira de fazer isso é fazer com que seu servidor SSH seja autenticado a partir de um servidor LDAP ou alguma outra fonte de autenticação centralizada que você já tenha configurado. Contanto que cada usuário possa obter acesso shell na máquina, qualquer mecanismo de autenticação SSH que você possa imaginar deve funcionar.

Gerando sua chave pública SSH

Muitos servidores Git são autenticados usando chaves públicas SSH. Para fornecer uma chave pública, cada usuário em seu sistema deve gerar uma, se ainda não tiver uma. Esse processo é semelhante em todos os sistemas operacionais. Primeiro, você deve verificar se já não possui uma chave. Por padrão, as chaves SSH de um usuário são armazenadas no diretório ~/.ssh desse usuário . Você pode verificar facilmente se já possui uma chave acessando esse diretório e listando o conteúdo:

```
$ cd ~/.ssh $
ls
authorised_keys2 id_dsa config                    host_conhecido
id_dsa.pub
```

Você está procurando um par de arquivos nomeados como id_dsa ou id_rsa e um arquivo correspondente com uma extensão .pub . O arquivo .pub é sua chave pública e o outro arquivo é a chave privada correspondente. Se você não tiver esses arquivos (ou nem mesmo tiver um diretório .ssh), poderá criá-los executando um programa chamado ssh-keygen, que é fornecido com o pacote SSH em sistemas Linux/macOS e vem com Git para Windows:

```
$ ssh-keygen -o
Gerando par de chaves rsa pública/privada.
Digite o arquivo no qual salvar a chave (/home/schacon/.ssh/id_rsa): Diretório criado '/
home/schacon/.ssh'.
Insira a frase secreta (vazio para nenhuma frase secreta):
Digite a mesma frase secreta novamente: Sua identificação
foi salva em /home/schacon/.ssh/id_rsa.
Sua chave pública foi salva em /home/schacon/.ssh/id_rsa.pub.
A impressão digital da chave é:
```

Machine Translated by Google

```
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Primeiro, ele confirma onde você deseja salvar a chave (.ssh/id_rsa) e, em seguida, solicita duas vezes uma senha, que você pode deixar em branco se não quiser digitar uma senha ao usar a chave.
No entanto, se você usar uma senha, certifique-se de adicionar a opção -o ; ele salva a chave privada em um formato que é mais resistente à quebra de senha por força bruta do que o formato padrão. Você também pode usar a ferramenta ssh-agent para evitar ter que digitar a senha toda vez.

Agora, cada usuário que fizer isso deve enviar sua chave pública para você ou para quem estiver administrando o servidor Git (supondo que você esteja usando uma configuração de servidor SSH que requer chaves públicas). Tudo o que eles precisam fazer é copiar o conteúdo do arquivo .pub e enviá-lo por e-mail. As chaves públicas são mais ou menos assim:

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

Para um tutorial mais aprofundado sobre como criar uma chave SSH em vários sistemas operacionais, consulte o guia GitHub sobre chaves SSH em https://docs.github.com/en/github/authenticating-to-github/generating-a-new ssh-chave-e-adicionando-ao-agente-ssh.

Configurando o servidor

Vamos percorrer a configuração do acesso SSH no lado do servidor. Neste exemplo, você usará o método author_keys para autenticar seus usuários. Também presumimos que você esteja executando uma distribuição padrão do Linux, como o Ubuntu.

ÿ Muito do que é descrito aqui pode ser automatizado usando o ssh-copy-id comando, em vez de copiar e instalar manualmente as chaves públicas.

Primeiro, você cria uma conta de usuário git e um diretório .ssh para esse usuário.

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

Em seguida, você precisa adicionar algumas chaves públicas SSH do desenvolvedor ao arquivo author_keys para o usuário git . Vamos supor que você tenha algumas chaves públicas confiáveis e as tenha salvo em arquivos temporários. Novamente, as chaves públicas são mais ou menos assim:

```
$ cat /tmp/id_rsa.john.pub
```

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Basta anexá-los ao arquivo author_keys do usuário git em seu diretório .ssh :

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

Agora, você pode configurar um repositório vazio para eles executando git init com a opção --bare , que inicializa o repositório sem um diretório de trabalho:

```
$ cd /srv/git $
mkdir project.git $ cd
project.git $ git init --bare
Repositório Git vazio
inicializado em /srv/git/project.git/
```

Em seguida, John, Josie ou Jessica podem enviar a primeira versão de seu projeto para esse repositório adicionando-o como um controle remoto e enviando uma ramificação. Observe que alguém deve fazer shell na máquina e criar um repositório simples toda vez que você quiser adicionar um projeto. Vamos usar gitserver como o nome do host do servidor no qual você configurou seu usuário e repositório git . Se você o estiver executando internamente e configurar o DNS para o gitserver apontar para esse servidor, poderá usar os comandos praticamente como estão (supondo que myproject seja um projeto existente com arquivos):

```
# no computador de John
$ cd myproject $ git init $
git add . $ git commit -m
'Initial commit' $ git remote
add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

Neste ponto, os outros podem cloná-lo e enviar as alterações de volta com a mesma facilidade:

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

Machine Translated by Google

Com esse método, você pode obter rapidamente um servidor Git de leitura/gravação instalado e funcionando para um punhado de desenvolvedores.

Você deve observar que atualmente todos esses usuários também podem fazer login no servidor e obter um shell como o usuário git . Se você quiser restringir isso, terá que alterar o shell para outra coisa no arquivo /etc/passwd .

Você pode facilmente restringir a conta de usuário git apenas para atividades relacionadas ao Git com uma ferramenta de shell limitada chamada git-shell que vem com o Git. Se você definir isso como o shell de login da conta do usuário git , essa conta não poderá ter acesso shell normal ao seu servidor. Para usar isso, especifique git-shell em vez de bash ou csh para o shell de login dessa conta. Para fazer isso, você deve primeiro adicionar o caminho completo do comando git-shell para /etc/shells se ainda não estiver lá:

```
$ cat /etc/shells # veja se o git-shell já está lá. Se não... $ which git-shell # verifique se
o git-shell está instalado em seu sistema. $ sudo -e /etc/shells # e adicione o caminho
para git-shell do último comando
```

Agora você pode editar o shell para um usuário usando chsh <username> -s <shell>:

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

Agora, o usuário git ainda pode usar a conexão SSH para enviar e receber repositórios Git, mas não pode fazer shell na máquina. Se você tentar, verá uma rejeição de login como esta:

```
$ ssh git@gitserver
fatal: O shell git interativo não está ativado. dica: ~/git-
shell-commands deve existir e ter acesso de leitura e execução.
Conexão com o gitserver fechada.
```

Neste ponto, os usuários ainda podem usar o encaminhamento de porta SSH para acessar qualquer host que o servidor git possa alcançar. Se quiser evitar isso, você pode editar o arquivo author_keys e incluir as seguintes opções antes de cada chave que deseja restringir:

```
sem encaminhamento de porta, sem encaminhamento de X11, sem encaminhamento de agente, sem pty
```

O resultado deve ficar assim:

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

Machine Translated by Google

```
sem encaminhamento de porta, sem encaminhamento de X11, sem encaminhamento de agente, sem pty
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...
```

Agora, os comandos de rede do Git ainda funcionarão bem, mas os usuários não poderão obter um shell. Conforme a saída indica, você também pode configurar um diretório no diretório inicial do usuário git que personaliza um pouco o comando git-shell . Por exemplo, você pode restringir os comandos do Git que o servidor aceitará ou pode personalizar a mensagem que os usuários verão se tentarem o SSH dessa maneira. Execute git help shell para obter mais informações sobre como personalizar o shell.

Git DaemonGenericName

Em seguida, configuraremos um daemon servindo repositórios usando o protocolo “Git”. Essa é uma escolha comum para acesso rápido e não autenticado aos seus dados do Git. Lembre-se de que, como esse não é um serviço autenticado, tudo o que você servir por meio desse protocolo é público em sua rede.

Se você estiver executando isso em um servidor fora do seu firewall, ele deve ser usado apenas para projetos visíveis publicamente para o mundo. Se o servidor em que você está executando estiver dentro do seu firewall, você pode usá-lo para projetos em que um grande número de pessoas ou computadores (integração contínua ou servidores de compilação) tenham acesso somente leitura, quando você não quiser ter para adicionar uma chave SSH para cada um.

De qualquer forma, o protocolo Git é relativamente fácil de configurar. Basicamente, você precisa executar este comando de maneira daemonizada:

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

A opção --reuseaddr permite que o servidor reinicie sem esperar que as conexões antigas expirem, enquanto a opção --base-path permite que as pessoas clonem projetos sem especificar o caminho inteiro, e o caminho no final informa ao Git daemon onde procure repositórios para exportar. Se você estiver executando um firewall, também precisará fazer um furo na porta 9418 na caixa em que está configurando.

Você pode tornar esse processo um daemon de várias maneiras, dependendo do sistema operacional em execução.

Como o systemd é o sistema init mais comum entre as distribuições modernas do Linux, você pode usá-lo para essa finalidade. Simplesmente coloque um arquivo em /etc/systemd/system/git-daemon.service com este conteúdo:

```
[Unidade]
Description=Iniciar Git Daemon

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

Reiniciar=sempre
RestartSec=500ms

StandardOutput=syslog
```

Machine Translated by Google

```
StandardError=syslog
SyslogIdentifier=git-daemon

Usuário=git
Grupo=git

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

Você deve ter notado que o Git daemon é iniciado aqui com git como grupo e usuário. Modifique-o para atender às suas necessidades e verifique se o usuário fornecido existe no sistema. Além disso, verifique se o binário do Git está realmente localizado em /usr/bin/git e altere o caminho, se necessário.

Por fim, você executará systemctl enable git-daemon para iniciar automaticamente o serviço na inicialização e poderá iniciar e interromper o serviço com, respectivamente, systemctl start git-daemon e systemctl stop git demônio.

Em outros sistemas, você pode querer usar o xinetd, um script em seu sistema sysvinit ou qualquer outra coisa - contanto que você tenha esse comando daemonizado e observado de alguma forma.

Em seguida, você deve informar ao Git quais repositórios permitir o acesso baseado em servidor Git não autenticado. Você pode fazer isso em cada repositório criando um arquivo chamado git-daemon-export-ok.

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

A presença desse arquivo informa ao Git que não há problema em servir este projeto sem autenticação.

HTTP inteligente

Agora temos acesso autenticado via SSH e acesso não autenticado via git://, mas também existe um protocolo que pode fazer as duas coisas ao mesmo tempo. Configurar o Smart HTTP é basicamente habilitar um script CGI fornecido com o Git chamado git-http-backend no servidor. Este CGI lerá o caminho e os cabeçalhos enviados por um git fetch ou git push para uma URL HTTP e determinará se o cliente pode se comunicar por HTTP (o que é verdade para qualquer cliente desde a versão 1.6.6). Se o CGI perceber que o cliente é inteligente, ele se comunicará de maneira inteligente com ele; caso contrário, ele retornará ao comportamento burro (portanto, é compatível com versões anteriores para leituras com clientes mais antigos).

Vamos percorrer uma configuração muito básica. Vamos configurar isso com o Apache como o servidor CGI. Se você não tiver a configuração do Apache, poderá fazê-lo em uma caixa do Linux com algo assim:

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

Isso também habilita os módulos mod_cgi, mod_alias e mod_env , todos necessários para que funcione corretamente.

Machine Translated by Google

Você também precisará definir o grupo de usuários Unix dos diretórios /srv/git para www-data para que seu servidor da Web possa acessar os repositórios para leitura e gravação, porque a instância do Apache executando o script CGI será (por padrão) executando como esse usuário:

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

Em seguida, precisamos adicionar algumas coisas à configuração do Apache para executar o git-http-backend como o manipulador de qualquer coisa que entre no caminho /git do seu servidor web.

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

Se você deixar de fora a variável de ambiente GIT_HTTP_EXPORT_ALL , o Git servirá apenas para clientes não autenticados os repositórios com o arquivo git-daemon-export-ok neles, assim como o Git daemon fazia.

Por fim, você desejará dizer ao Apache para permitir solicitações ao git-http-backend e fazer com que as gravações sejam autenticadas de alguma forma, possivelmente com um bloco Auth como este:

```
<Arquivos "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$#)
Exigir usuário válido </
Files>
```

Isso exigirá que você crie um arquivo .htpasswd contendo as senhas de todos os usuários válidos. Aqui está um exemplo de como adicionar um usuário “schacon” ao arquivo:

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

Existem inúmeras maneiras de fazer com que o Apache autentique os usuários, você terá que escolher e implementar uma delas. Este é apenas o exemplo mais simples que poderíamos apresentar. Você também certamente desejará configurar isso por SSL para que todos esses dados sejam criptografados.

Não queremos ir muito longe nas especificações de configuração do Apache, já que você pode estar usando um servidor diferente ou ter necessidades de autenticação diferentes. A ideia é que o Git venha com um CGI chamado git-http-backend que, quando invocado, fará toda a negociação para enviar e receber dados por HTTP. Ele não implementa nenhuma autenticação em si, mas isso pode ser facilmente controlado na camada do servidor web que o invoca. Você pode fazer isso com praticamente qualquer servidor da Web compatível com CGI, portanto, escolha aquele que você conhece melhor.

Machine Translated by Google

ÿ Para obter mais informações sobre como configurar a autenticação no Apache, consulte o Apache docs aqui: https://httpd.apache.org/docs/current/howto/auth.html.

GitWeb

Agora que você tem acesso básico de leitura/gravação e somente leitura ao seu projeto, talvez queira configurar um visualizador simples baseado na Web. O Git vem com um script CGI chamado GitWeb que às vezes é usado para isso.

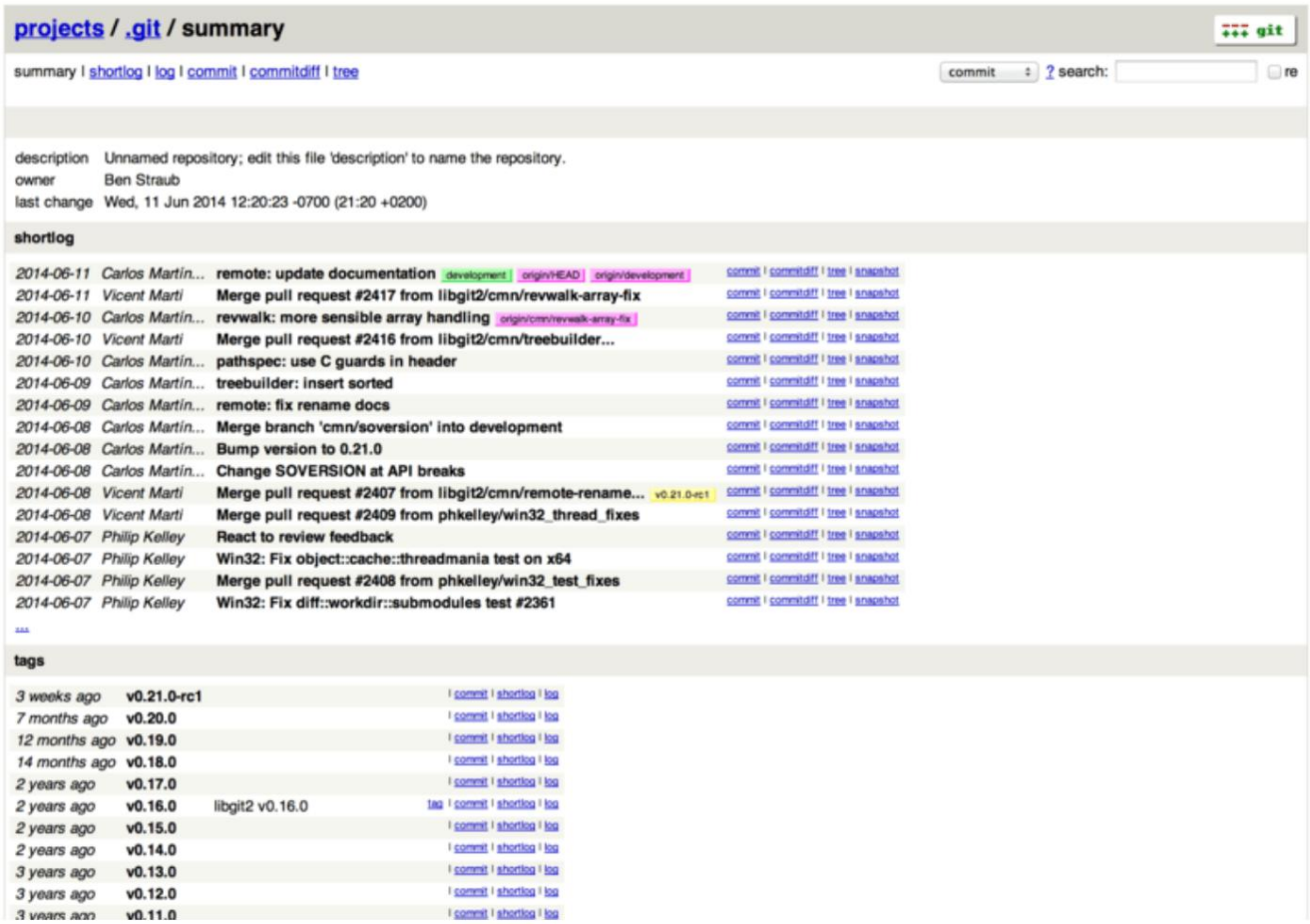


Figura 49. A interface do usuário baseada na web do GitWeb

Se você quiser verificar como seria o GitWeb para o seu projeto, o Git vem com um comando para iniciar uma instância temporária se você tiver um servidor da Web leve em seu sistema, como lighttpd ou webrick . Em máquinas Linux, o lighttpd geralmente é instalado, portanto, você pode executá-lo digitando git instaweb no diretório do projeto. Se você estiver executando um Mac, o Leopard vem pré-instalado com o Ruby, então o webrick pode ser sua melhor aposta. Para iniciar o instaweb com um manipulador não lighttpd, você pode executá-lo com a opção --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]
```

Isso inicia um servidor HTTPD na porta 1234 e, em seguida, inicia automaticamente um navegador da Web que é aberto nessa página. É muito fácil de sua parte. Quando terminar e quiser desligar o

Machine Translated by Google

server, você pode executar o mesmo comando com a opção --stop :

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

Se você deseja executar a interface da Web em um servidor o tempo todo para sua equipe ou para um projeto de código aberto que está hospedando, precisará configurar o script CGI para ser servido por seu servidor da Web normal. Algumas distribuições Linux têm um pacote gitweb que você pode instalar via apt ou dnf, então você pode querer tentar isso primeiro. Faremos a instalação manual do GitWeb muito rapidamente. Primeiro, você precisa obter o código-fonte do Git, que acompanha o GitWeb, e gerar o script CGI personalizado:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git $ cd git/ $ make GITWEB_PROJECTROOT="/srv/git" prefix=/usr gitweb

SUBDIR gitweb SUBDIR ../ make[2 ]: `GIT-VERSION-FILE' está atualizado.





GEN gitweb.cgi GEN static/gitweb.js $

sudo cp -Rf gitweb /var/www/
```

Observe que você precisa informar ao comando onde encontrar seus repositórios Git com a variável GITWEB_PROJECTROOT . Agora, você precisa fazer o Apache usar CGI para esse script, para o qual você pode adicionar um VirtualHost:

```
<VirtualHost *:80> ServerName gitserver

DocumentRoot /var/www/gitweb <Diretório /var/www/

gitweb> Opções +ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch

AllowOverride Todos os pedidos permitem, negam Permitir de todos

                    AddHandler cgi-script cgi DirectoryIndex gitweb.cgi < /Diretório> </VirtualHost>
```

Novamente, o GitWeb pode ser servido com qualquer servidor web compatível com CGI ou Perl; se você preferir usar outra coisa, não deve ser difícil de configurar. Neste ponto, você deve ser capaz de visitar http://gitserver/ para visualizar seus repositórios online.

GitLabGenericName

GitWeb é bastante simplista. Se você está procurando um servidor Git moderno e completo, existem várias soluções de código aberto que você pode instalar. Como o GitLab é um dos mais populares, abordaremos a instalação e o uso dele como exemplo. Isso é mais difícil que o GitWeb

Machine Translated by Google

opcional e exigirá mais manutenção, mas é uma opção com todos os recursos.

Instalação

O GitLab é um aplicativo da Web baseado em banco de dados, portanto, sua instalação é mais complicada do que alguns outros servidores Git. Felizmente, esse processo é bem documentado e suportado. O GitLab recomenda enfaticamente a instalação do GitLab em seu servidor por meio do pacote oficial Omnibus GitLab.

As outras opções de instalação são:

- Gráfico GitLab Helm, para uso com Kubernetes.
- Pacotes do GitLab dockerizados para uso com o Docker.
- Dos arquivos de origem.
- Provedor de nuvem como AWS, Google Cloud Platform, Azure, OpenShift e Digital Ocean.

Para obter mais informações, leia o leia-me do GitLab Community Edition (CE).

Administração

A interface de administração do GitLab é acessada pela web. Simplesmente aponte seu navegador para o nome do host ou endereço IP onde o GitLab está instalado e faça login como usuário administrador. O nome de usuário padrão é admin@local.host e a senha padrão é 5iveL!fe (que você deve alterar imediatamente). Depois de fazer login, clique no ícone “Área de administração” no menu no canto superior direito.

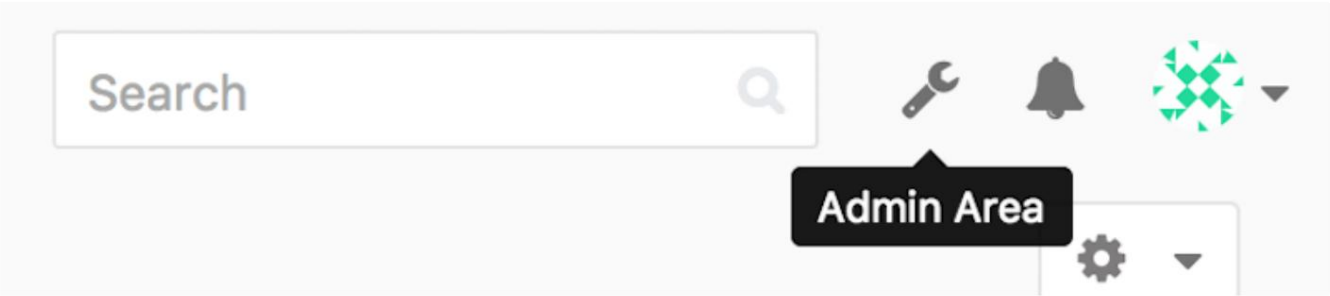


Figura 50. O item “Admin area” no menu GitLab

Usuários

Todos que usam seu servidor GitLab devem ter uma conta de usuário. As contas de usuário são bastante simples, elas contêm principalmente informações pessoais anexadas aos dados de login. Cada conta de usuário possui um **namespace,** que é um agrupamento lógico de projetos que pertencem a esse usuário. Se a usuária jane tivesse um projeto chamado projeto, a url desse projeto seria http://server/jane/project.

Machine Translated by Google

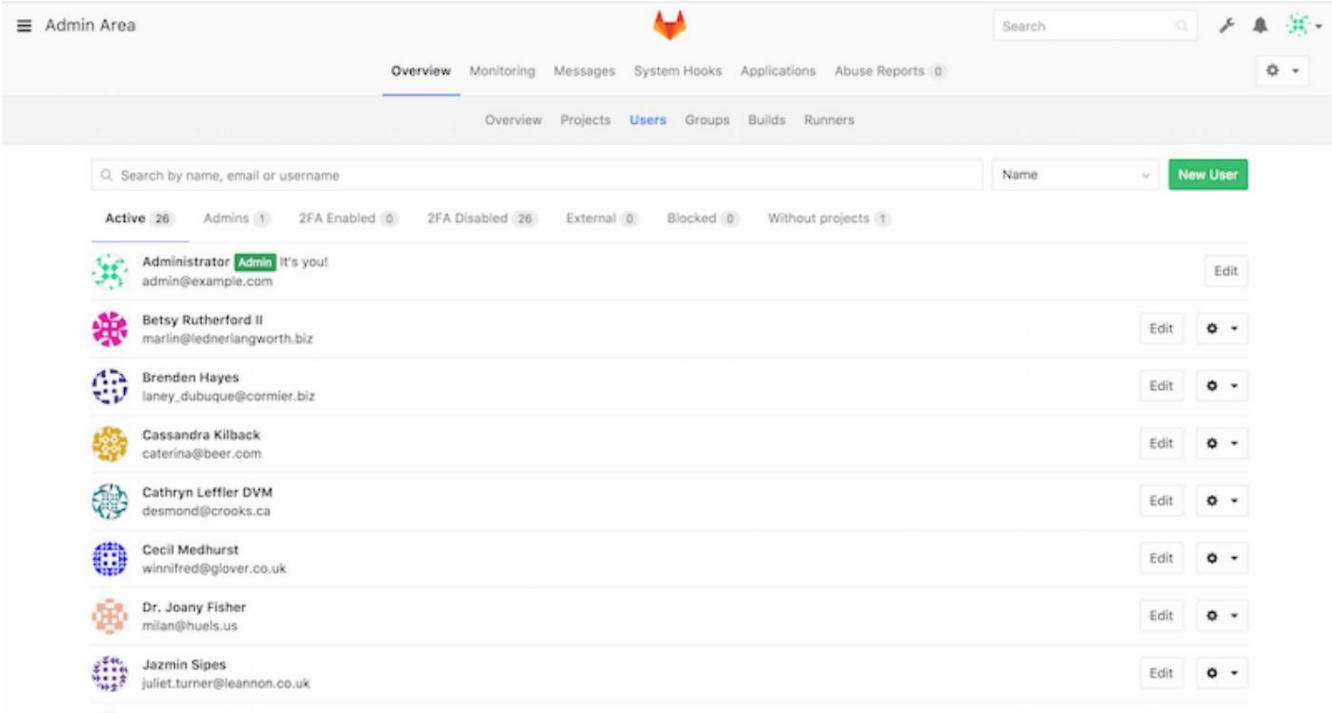


Figura 51. A tela de administração do usuário do GitLab

Você pode remover uma conta de usuário de duas maneiras: “Bloquear” um usuário impede que ele faça login na instância do GitLab, mas todos os dados sob o namespace desse usuário serão preservados e os commits assinados com o endereço de e-mail desse usuário ainda serão vinculados a o perfil deles.

“Destruir” um usuário, por outro lado, remove-o completamente do banco de dados e do sistema de arquivos. Todos os projetos e dados em seu namespace são removidos, e todos os grupos de sua propriedade também serão removidos. Esta é obviamente uma ação muito mais permanente e destrutiva, e você raramente precisará dela.

Grupos

Um grupo GitLab é uma coleção de projetos, juntamente com dados sobre como os usuários podem acessar esses projetos. Cada grupo tem um namespace de projeto (da mesma forma que os usuários), portanto, se o treinamento do grupo tiver materiais de projeto, sua url seria http://servidor/treinamento/materiais.

Machine Translated by Google

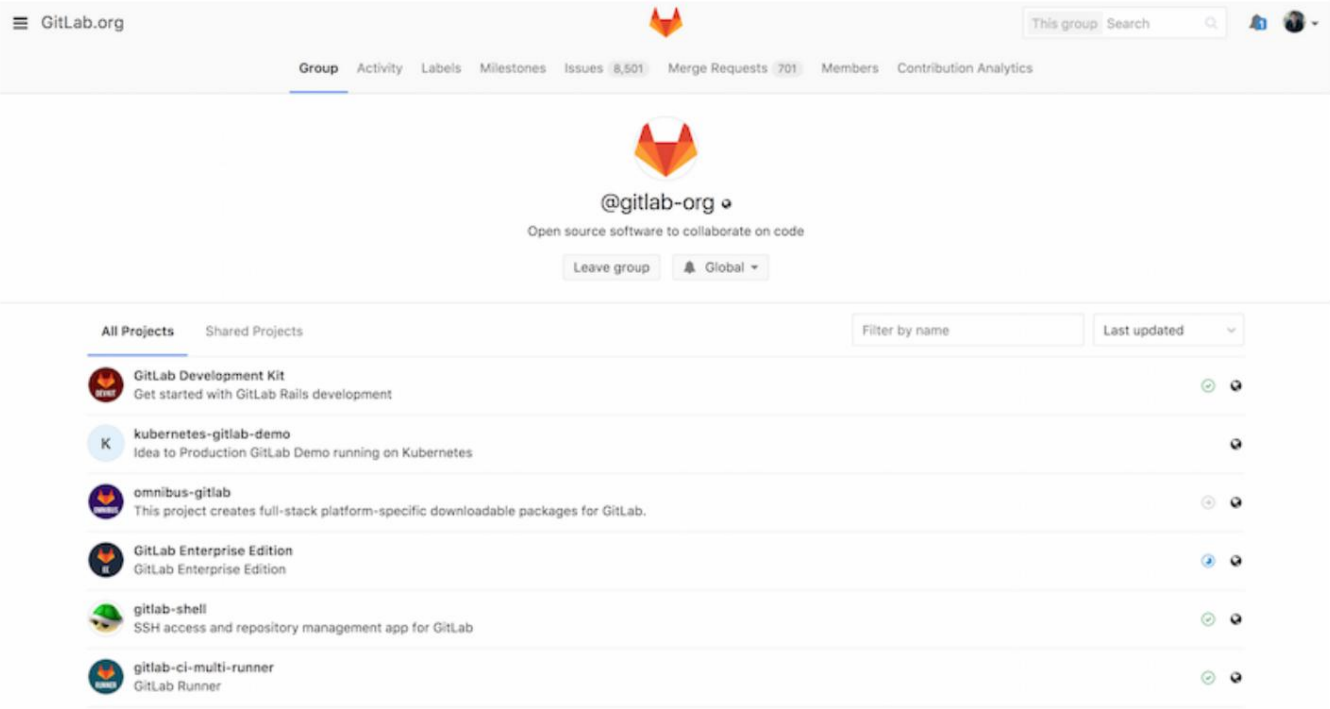


Figura 52. A tela de administração do grupo GitLab

Cada grupo está associado a um número de usuários, cada um com um nível de permissões para os projetos do grupo e para o próprio grupo. Eles variam de “Convidado” (somente problemas e bate-papo) a “Proprietário” (controle total do grupo, seus membros e seus projetos). Os tipos de permissões são muito numerosos para listar aqui, mas o GitLab tem um link útil na tela de administração.

Projetos

Um projeto GitLab corresponde aproximadamente a um único repositório Git. Cada projeto pertence a um único namespace, seja um usuário ou um grupo. Se o projeto pertencer a um usuário, o proprietário do projeto tem controle direto sobre quem tem acesso ao projeto; se o projeto pertencer a um grupo, as permissões de nível de usuário do grupo entrarão em vigor.

Cada projeto tem um nível de visibilidade, que controla quem tem acesso de leitura às páginas e repositório desse projeto. Se um projeto for *privado,* o proprietário do projeto deverá conceder acesso explicitamente a usuários específicos. Um projeto *interno* é visível para qualquer usuário conectado e um projeto *público* é visível para qualquer pessoa. Observe que isso controla o acesso git fetch , bem como o acesso à interface do usuário da web para esse projeto.

ganchos

O GitLab inclui suporte para ganchos, tanto no nível do projeto quanto do sistema. Para qualquer um deles, o servidor GitLab executará um HTTP POST com algum JSON descritivo sempre que ocorrerem eventos relevantes. Essa é uma ótima maneira de conectar seus repositórios Git e instância GitLab ao restante de sua automação de desenvolvimento, como servidores CI, salas de bate-papo ou ferramentas de implantação.

Uso básico

A primeira coisa que você deseja fazer com o GitLab é criar um novo projeto. Você pode fazer isso clicando no ícone “+” na barra de ferramentas. Você será solicitado a fornecer o nome do projeto, a qual namespace ele deve pertencer e qual deve ser seu nível de visibilidade. A maior parte do que você especifica aqui não é permanente e pode ser alterado posteriormente por meio da interface de configurações. Clique em “Criar Projeto” e pronto.

Machine Translated by Google

Depois que o projeto existir, você provavelmente desejará conectá-lo a um repositório Git local. Cada projeto pode ser acessado por HTTPS ou SSH, qualquer um dos quais pode ser usado para configurar um controle remoto do Git. As URLs ficam visíveis na parte superior da página inicial do projeto. Para um repositório local existente, este comando criará um remoto chamado gitlab para o local hospedado:

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

Se você não tiver uma cópia local do repositório, basta fazer isso:

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

A interface do usuário da Web fornece acesso a várias visualizações úteis do próprio repositório. A página inicial de cada projeto mostra a atividade recente e os links na parte superior o levarão a visualizações dos arquivos do projeto e do log de confirmação.

Trabalhando juntos

A maneira mais simples de trabalhar em conjunto em um projeto GitLab é dando a cada usuário acesso push direto ao repositório Git. Você pode adicionar um usuário a um projeto acessando a seção “Membros” das configurações desse projeto e associando o novo usuário a um nível de acesso (os diferentes níveis de acesso são discutidos um pouco em Grupos) . Ao dar a um usuário um nível de acesso de “Desenvolvedor” ou superior, esse usuário pode enviar commits e ramificações diretamente para o repositório.

Outra forma de colaboração mais desacoplada é usar solicitações de mesclagem. Esse recurso permite que qualquer usuário que veja um projeto contribua com ele de maneira controlada. Os usuários com acesso direto podem simplesmente criar uma ramificação, enviar commits para ela e abrir uma solicitação de mesclagem de sua ramificação de volta para a master ou qualquer outra ramificação. Os usuários que não têm permissões de push para um repositório podem “bifurcá-lo” para criar sua própria cópia, enviar commits para *sua* cópia e abrir uma solicitação de mesclagem de seu fork de volta ao projeto principal. Esse modelo permite que o proprietário tenha controle total sobre o que entra no repositório e quando, ao mesmo tempo em que permite contribuições de usuários não confiáveis.

Solicitações e problemas de mesclagem são as principais unidades de discussão de longa duração no GitLab. Cada solicitação de mesclagem permite uma discussão linha por linha da alteração proposta (que oferece suporte a um tipo leve de revisão de código), bem como um tópico geral de discussão geral. Ambos podem ser atribuídos a usuários ou organizados em marcos.

Esta seção é focada principalmente nos recursos relacionados ao Git do GitLab, mas como um projeto maduro, ele fornece muitos outros recursos para ajudar sua equipe a trabalhar em conjunto, como wikis de projeto e ferramentas de manutenção do sistema. Um benefício do GitLab é que, depois que o servidor estiver configurado e funcionando, você raramente precisará ajustar um arquivo de configuração ou acessar o servidor via SSH; a maior parte da administração e do uso geral pode ser feita por meio da interface do navegador.

Opções hospedadas por terceiros

Se não quiser passar por todo o trabalho envolvido na configuração de seu próprio servidor Git, você tem várias opções para hospedar seus projetos Git em um site de hospedagem dedicado externo. Fazer isso oferece um

Machine Translated by Google

uma série de vantagens: um site de hospedagem geralmente é rápido de configurar e fácil de iniciar projetos, e nenhuma manutenção ou monitoramento do servidor está envolvido. Mesmo que você configure e execute seu próprio servidor internamente, você ainda pode querer usar um site de hospedagem público para seu código-fonte aberto – geralmente é mais fácil para a comunidade de código-fonte aberto encontrar e ajudá-lo.

Hoje em dia, você tem um grande número de opções de hospedagem para escolher, cada uma com diferentes vantagens e desvantagens. Para ver uma lista atualizada, confira a página GitHosting no wiki principal do Git em https://git.wiki.kernel.org/index.php/GitHosting.

Abordaremos o uso do GitHub em detalhes no GitHub, pois é o maior host Git existente e você pode precisar interagir com projetos hospedados nele de qualquer maneira, mas há dezenas de outros para escolher, caso você não queira configurar seu próprio servidor Git.

Resumo

Você tem várias opções para colocar um repositório Git remoto em funcionamento para poder colaborar com outras pessoas ou compartilhar seu trabalho.

Executar seu próprio servidor oferece muito controle e permite que você execute o servidor dentro de seu próprio firewall, mas esse servidor geralmente requer uma quantidade razoável de seu tempo para configurar e manter. Se você colocar seus dados em um servidor hospedado, é fácil configurar e manter; no entanto, você deve ser capaz de manter seu código nos servidores de outra pessoa, e algumas organizações não permitem isso.

Deve ser bastante simples determinar qual solução ou combinação de soluções é apropriada para você e sua organização.

Machine Translated by Google

Git distribuído

Agora que você tem um repositório Git remoto configurado como um ponto focal para todos os desenvolvedores compartilharem seu código e está familiarizado com os comandos básicos do Git em um fluxo de trabalho local, verá como utilizar alguns dos fluxos de trabalho distribuídos que o Git oferece a você.

Neste capítulo, você verá como trabalhar com o Git em um ambiente distribuído como colaborador e integrador. Ou seja, você aprenderá como contribuir com código com sucesso para um projeto e torná-lo o mais fácil possível para você e para o mantenedor do projeto, e também como manter um projeto com sucesso com a contribuição de vários desenvolvedores.

Fluxos de Trabalho Distribuídos

Em contraste com os Sistemas Centralizados de Controle de Versão (CVCSs), a natureza distribuída do Git permite que você seja muito mais flexível em como os desenvolvedores colaboram em projetos. Em sistemas centralizados, cada desenvolvedor é um nó trabalhando mais ou menos igualmente com um hub central. No Git, no entanto, todo desenvolvedor é potencialmente um nó e um hub; ou seja, todo desenvolvedor pode contribuir com código para outros repositórios e manter um repositório público no qual outros possam basear seu trabalho e para o qual possam contribuir. Isso apresenta uma vasta gama de possibilidades de fluxo de trabalho para seu projeto e/ou sua equipe, então abordaremos alguns paradigmas comuns que tiram proveito dessa flexibilidade. Examinaremos os pontos fortes e possíveis pontos fracos de cada projeto; você pode escolher um único para usar ou pode misturar e combinar recursos de cada um.

Fluxo de trabalho centralizado

Em sistemas centralizados, geralmente há um único modelo de colaboração — o fluxo de trabalho centralizado. Um hub central, ou *repositório,* pode aceitar código e todos sincronizam seu trabalho com ele. Vários desenvolvedores são nós - consumidores desse hub - e sincronizam com esse local centralizado.

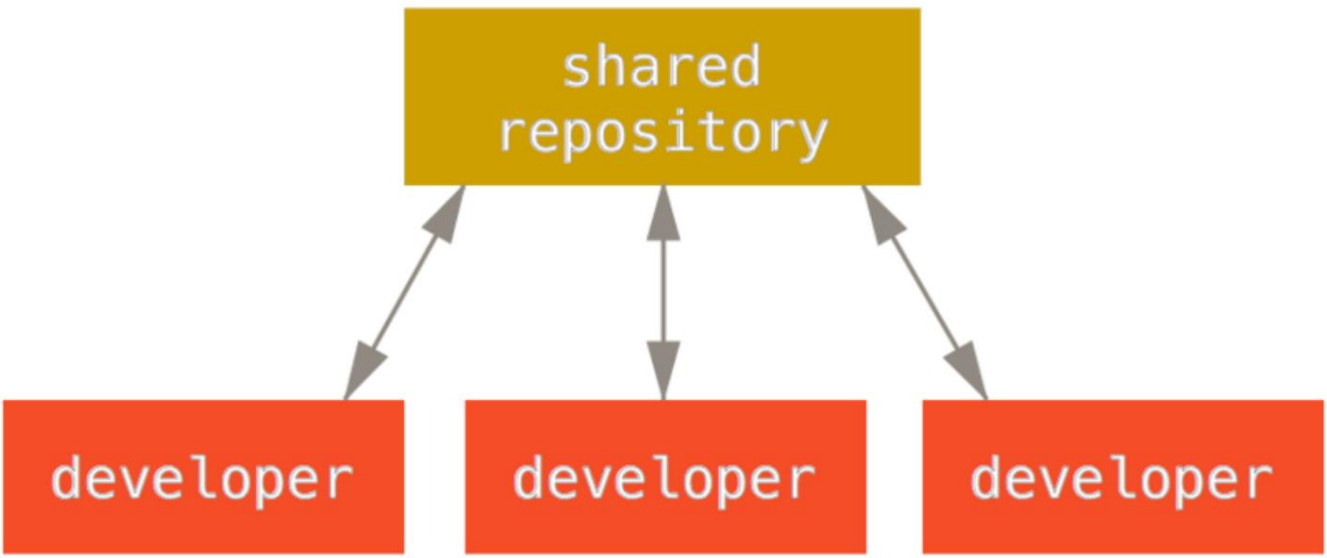


Figura 53. Fluxo de trabalho centralizado

Isso significa que, se dois desenvolvedores clonarem do hub e ambos fizerem alterações, o primeiro desenvolvedor a enviar suas alterações de volta poderá fazê-lo sem problemas. O segundo desenvolvedor deve se fundir no

Machine Translated by Google

o trabalho do primeiro antes de enviar as alterações, para não sobrescrever as alterações do primeiro desenvolvedor. Este conceito é tão verdadeiro no Git quanto no Subversion (ou qualquer CVCS), e este modelo funciona perfeitamente bem no Git.

Se você já está confortável com um fluxo de trabalho centralizado em sua empresa ou equipe, pode facilmente continuar usando esse fluxo de trabalho com o Git. Basta configurar um único repositório e dar a todos em sua equipe acesso push; O Git não permitirá que os usuários sobrescrevam uns aos outros.

Digamos que John e Jessica comecem a trabalhar ao mesmo tempo. John termina sua alteração e a envia para o servidor. Em seguida, Jessica tenta enviar suas alterações, mas o servidor as rejeita. Ela é informada de que está tentando fazer alterações não rápidas e que não poderá fazê-lo até que faça buscas e mesclagens. Esse fluxo de trabalho é atraente para muitas pessoas porque é um paradigma com o qual muitos estão familiarizados e confortáveis.

Isso também não se limita a equipes pequenas. Com o modelo de ramificação do Git, é possível que centenas de desenvolvedores trabalhem com sucesso em um único projeto por meio de dezenas de ramificações simultaneamente.

Fluxo de trabalho do gerenciador de integração

Como o Git permite que você tenha vários repositórios remotos, é possível ter um fluxo de trabalho em que cada desenvolvedor tenha acesso de gravação ao seu próprio repositório público e acesso de leitura aos demais.
Esse cenário geralmente inclui um repositório canônico que representa o projeto “oficial”. Para contribuir com esse projeto, você cria seu próprio clone público do projeto e envia suas alterações para ele. Em seguida, você pode enviar uma solicitação ao mantenedor do projeto principal para obter suas alterações. O mantenedor pode então adicionar seu repositório como remoto, testar suas alterações localmente, mesclá-las em sua ramificação e enviar de volta para seu repositório. O processo funciona da seguinte maneira (consulte Fluxo de trabalho do gerenciador de integração):

1. O mantenedor do projeto envia para seu repositório público.
2. Um colaborador clona esse repositório e faz alterações.
3. O colaborador envia para sua própria cópia pública.
4. O colaborador envia um e-mail ao mantenedor solicitando que ele faça o pull das alterações.
5. O mantenedor adiciona o repositório do contribuidor como remoto e mescla localmente.
6. O mantenedor envia alterações mescladas para o repositório principal.

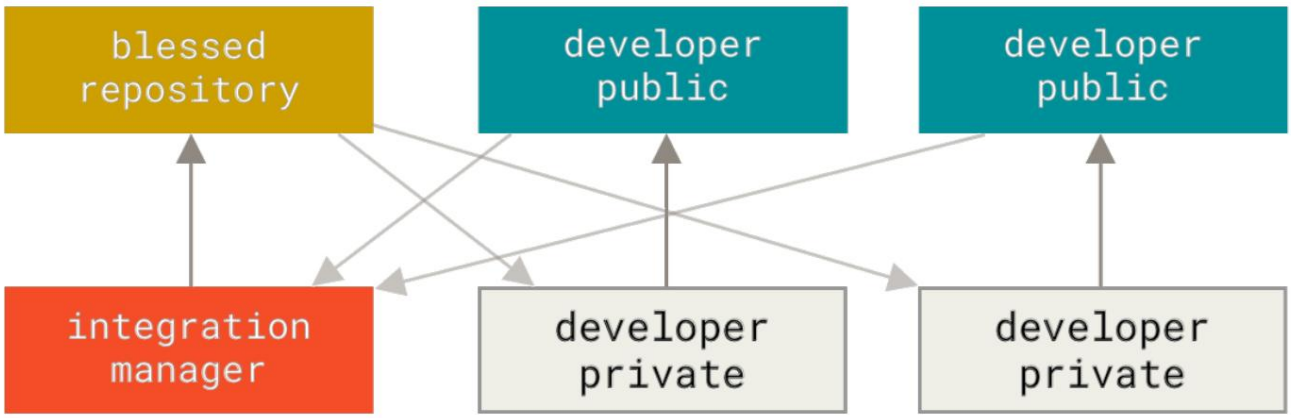


Figura 54. Fluxo de trabalho do gerenciador de integração

Machine Translated by Google

Este é um fluxo de trabalho muito comum com ferramentas baseadas em hub como GitHub ou GitLab, onde é fácil bifurcar um projeto e enviar suas alterações para o fork para que todos vejam. Uma das principais vantagens dessa abordagem é que você pode continuar trabalhando e o mantenedor do repositório principal pode obter suas alterações a qualquer momento. Os contribuidores não precisam esperar que o projeto incorpore suas alterações — cada parte pode trabalhar em seu próprio ritmo.

Fluxo de trabalho do ditador e dos tenentes

Esta é uma variante de um fluxo de trabalho de vários repositórios. Geralmente é usado por grandes projetos com centenas de colaboradores; um exemplo famoso é o kernel do Linux. Vários gerenciadores de integração são responsáveis por certas partes do repositório; eles são chamados *de tenentes.* Todos os tenentes têm um gerente de integração conhecido como ditador benevolente. O ditador benevolente faz push de seu diretório para um repositório de referência do qual todos os colaboradores precisam fazer pull. O processo funciona assim (consulte Fluxo de trabalho do ditador benevolente):

1. Os desenvolvedores regulares trabalham em seu branch de tópico e rebaseiam seu trabalho no topo do master. O branch master é aquele do repositório de referência para o qual o ditador empurra.
2. Lieutenants mesclam as ramificações de tópicos dos desenvolvedores em suas ramificações principais .
3. O ditador funde os ramos mestres dos tenentes no ramo mestre do ditador .
4. Por fim, o ditador envia essa ramificação mestre para o repositório de referência para que os outros desenvolvedores possam fazer o rebase nele.

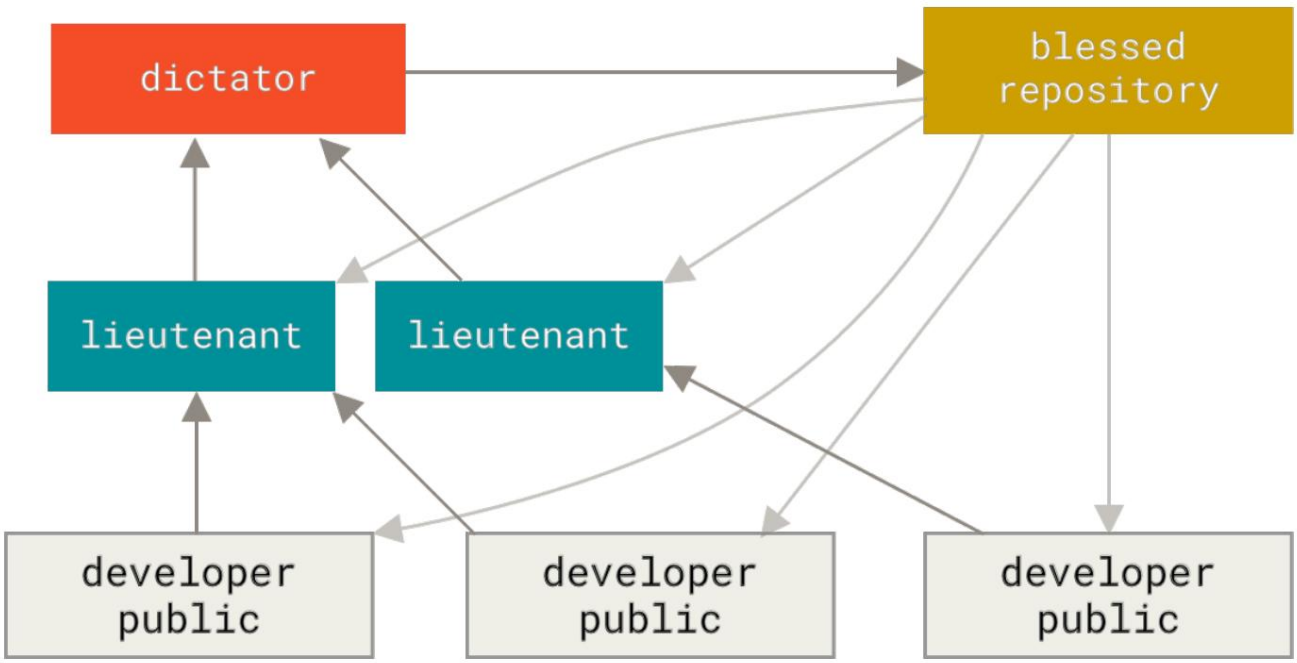


Figura 55. Fluxo de trabalho do ditador benevolente

Esse tipo de fluxo de trabalho não é comum, mas pode ser útil em projetos muito grandes ou em ambientes altamente hierárquicos. Ele permite que o líder do projeto (o ditador) delegue grande parte do trabalho e colete grandes subconjuntos de código em vários pontos antes de integrá-los.

Padrões para gerenciar ramificações de código-fonte

ÿ Martin Fowler fez um guia "Padrões para gerenciar ramificações de código-fonte".

Machine Translated by Google

> Este guia cobre todos os fluxos de trabalho comuns do Git e explica como/quando usá-los. Há também uma seção comparando frequências de integração altas e baixas.
>
> https://martinfowler.com/articles/branching-patterns.html

Resumo dos Fluxos de Trabalho

Estes são alguns fluxos de trabalho comumente usados que são possíveis com um sistema distribuído como o Git, mas você pode ver que muitas variações são possíveis para se adequar ao seu fluxo de trabalho específico do mundo real. Agora que você pode (espero) determinar qual combinação de fluxo de trabalho pode funcionar para você, abordaremos alguns exemplos mais específicos de como realizar as principais funções que compõem os diferentes fluxos. Na próxima seção, você aprenderá sobre alguns padrões comuns para contribuir com um projeto.

Contribuindo para um Projeto

A principal dificuldade em descrever como contribuir para um projeto são as inúmeras variações de como fazer isso. Como o Git é muito flexível, as pessoas podem e trabalham juntas de várias maneiras, e é problemático descrever como você deve contribuir — cada projeto é um pouco diferente. Algumas das variáveis envolvidas são contagem de colaboradores ativos, fluxo de trabalho escolhido, seu acesso de confirmação e, possivelmente, o método de contribuição externa.

A primeira variável é a contagem de contribuidores ativos — quantos usuários estão contribuindo ativamente com código para este projeto e com que frequência? Em muitos casos, você terá dois ou três desenvolvedores com alguns commits por dia, ou possivelmente menos para projetos um tanto inativos. Para empresas ou projetos maiores, o número de desenvolvedores pode estar na casa dos milhares, com centenas ou milhares de commits chegando todos os dias. Isso é importante porque, com mais e mais desenvolvedores, você enfrenta mais problemas para garantir que seu código seja aplicado de forma limpa ou possa ser facilmente mesclado. As alterações que você enviar podem se tornar obsoletas ou seriamente danificadas pelo trabalho mesclado enquanto você estava trabalhando ou enquanto suas alterações aguardavam para serem aprovadas ou aplicadas. Como você pode manter seu código consistentemente atualizado e seus commits válidos?

A próxima variável é o fluxo de trabalho em uso para o projeto. É centralizado, com cada desenvolvedor tendo igual acesso de gravação à linha de código principal? O projeto tem um mantenedor ou gerente de integração que verifica todos os patches? Todos os patches são revisados por pares e aprovados? Você está envolvido nesse processo? Existe um sistema de tenentes e você precisa enviar seu trabalho a eles primeiro?

A próxima variável é o seu acesso de confirmação. O fluxo de trabalho necessário para contribuir com um projeto é muito diferente se você tiver acesso de gravação ao projeto e se não tiver. Se você não tem acesso de gravação, como o projeto prefere aceitar trabalhos contribuídos? Ele ainda tem uma política? Com quanto trabalho você está contribuindo por vez? Com que frequência você contribui?

Todas essas questões podem afetar como você contribui efetivamente para um projeto e quais fluxos de trabalho são preferidos ou disponíveis para você. Abordaremos aspectos de cada um deles em uma série de casos de uso, indo do simples ao mais complexo; você deve ser capaz de construir os fluxos de trabalho específicos necessários na prática a partir desses exemplos.

Machine Translated by Google

Diretrizes de confirmação

Antes de começarmos a olhar para os casos de uso específicos, aqui está uma nota rápida sobre mensagens de commit. Ter uma boa diretriz para criar commits e cumpri-la facilita muito o trabalho com o Git e a colaboração com outras pessoas. O projeto Git fornece um documento que apresenta várias boas dicas para criar commits a partir dos quais enviar patches — você pode lê-lo no código-fonte do Git no arquivo Documentation/SubmittedPatches .

Primeiro, seus envios não devem conter nenhum erro de espaço em branco. O Git fornece uma maneira fácil de verificar isso — antes de confirmar, execute git diff --check, que identifica possíveis erros de espaço em branco e os lista para você.

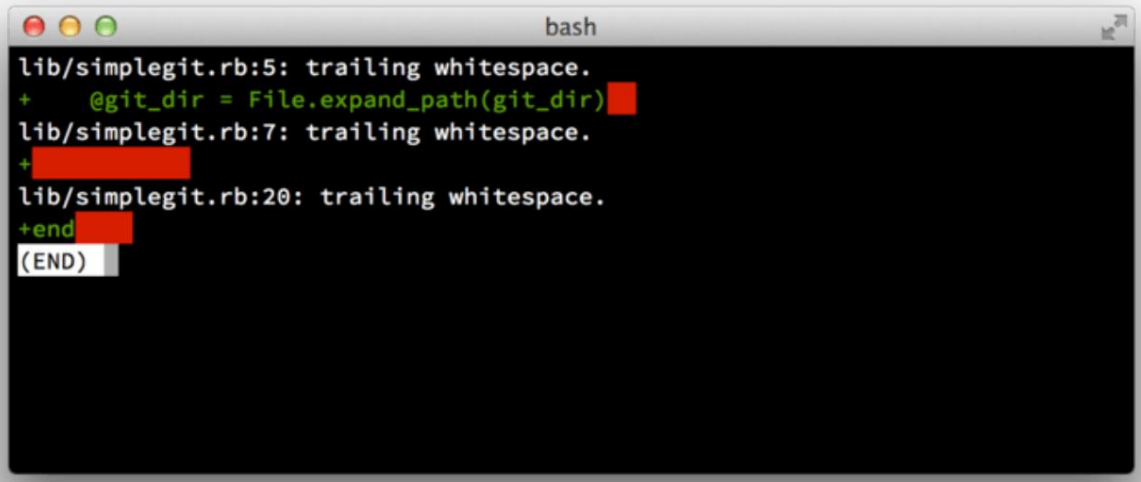


Figura 56. Saída de git diff --check

Se você executar esse comando antes de confirmar, poderá saber se está prestes a cometer problemas de espaço em branco que podem incomodar outros desenvolvedores.

Em seguida, tente fazer com que cada commit seja um changeset logicamente separado. Se puder, tente fazer com que suas alterações sejam digeríveis — não codifique durante um fim de semana inteiro em cinco problemas diferentes e, em seguida, envie-os todos como um grande commit na segunda-feira. Mesmo se você não se comprometer durante o fim de semana, use a área de preparação na segunda-feira para dividir seu trabalho em pelo menos um compromisso por edição, com uma mensagem útil por compromisso. Se algumas das alterações modificarem o mesmo arquivo, tente usar git add --patch para testar parcialmente os arquivos (abordados em detalhes em Teste interativo). O instantâneo do projeto na ponta da ramificação é idêntico, quer você faça um commit ou cinco, contanto que todas as alterações sejam adicionadas em algum ponto, então tente facilitar as coisas para seus colegas desenvolvedores quando eles tiverem que revisar suas alterações.

Essa abordagem também torna mais fácil remover ou reverter um dos conjuntos de alterações, caso seja necessário posteriormente. Reescrever o histórico descreve vários truques úteis do Git para reescrever o histórico e preparar arquivos de forma interativa — use essas ferramentas para ajudar a criar um histórico limpo e compreensível antes de enviar o trabalho para outra pessoa.

A última coisa a ter em mente é a mensagem de confirmação. Adquirir o hábito de criar mensagens de commit de qualidade facilita muito o uso e a colaboração com o Git. Como regra geral, seu

Machine Translated by Google

as mensagens devem começar com uma única linha com no máximo 50 caracteres e que descreva o changeset de forma concisa, seguida por uma linha em branco, seguida por uma explicação mais detalhada. O projeto Git requer que a explicação mais detalhada inclua sua motivação para a mudança e contraste sua implementação com o comportamento anterior — essa é uma boa diretriz a seguir. Escreva sua mensagem de commit no imperativo: "Fix bug" e não "Fixed bug" ou "Fixes bug". Aqui está um modelo que você pode seguir, que adaptamos levemente de um originalmente escrito por Tim Pope:

```
Resumo curto (50 caracteres ou menos) em letras maiúsculas

Texto explicativo mais detalhado, se necessário. Envolva-o para cerca de 72 caracteres
ou mais. Em alguns contextos, a primeira linha é tratada como o assunto de um e-mail e o
restante do texto como o corpo. A linha em branco que separa o resumo do corpo é crítica (a
menos que você omita o corpo inteiramente); ferramentas como rebase irão confundi-lo se você
executar os dois juntos.


Escreva sua mensagem de commit no imperativo: "Fix bug" e não "Fixed bug" ou "Fixes bug". Essa
convenção corresponde às mensagens de confirmação geradas por comandos como git merge e git
revert.

Outros parágrafos vêm após as linhas em branco.

- Pontos de bala também estão bem

- Normalmente, um hífen ou asterisco é usado para o marcador, seguido por um único espaço,
com linhas em branco no meio, mas as convenções variam aqui

- Use um recuo deslocado
```

Se todas as suas mensagens de commit seguirem esse modelo, as coisas serão muito mais fáceis para você e para os desenvolvedores com quem você colabora. O projeto Git tem mensagens de commit bem formatadas — tente executar git log --no-merges lá para ver como é um histórico de commit de projeto bem formatado.

ÿ

Faça o que dizemos, não o que fazemos.

Por uma questão de brevidade, muitos dos exemplos neste livro não têm mensagens de commit bem formatadas como esta; em vez disso, simplesmente usamos a opção -m para git commit.

Em suma, faça o que dizemos, não o que fazemos.

Equipe Pequena Privada

A configuração mais simples que você provavelmente encontrará é um projeto privado com um ou dois outros desenvolvedores. “Privado”, neste contexto, significa código fechado – não acessível ao mundo exterior. Você e os outros desenvolvedores têm acesso push ao repositório.

Nesse ambiente, você pode seguir um fluxo de trabalho semelhante ao que faria ao usar o Subversion ou outro sistema centralizado. Você ainda obtém as vantagens de coisas como off-line

Machine Translated by Google

confirmação e ramificação e mesclagem muito mais simples, mas o fluxo de trabalho pode ser muito semelhante; a principal diferença é que as mesclagens acontecem no lado do cliente e não no servidor no momento do commit. Vamos ver como pode ser quando dois desenvolvedores começam a trabalhar juntos com um repositório compartilhado. O primeiro desenvolvedor, John, clona o repositório, faz uma alteração e confirma localmente. As mensagens de protocolo foram substituídas por … nestes exemplos para encurtá-las um pouco.

```
# Máquina do João
$ git clone john@githost:simplegit.git
Clonagem em 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb $
git commit -am 'Remover valor padrão
inválido' [mestre 738ee87] Remover valor padrão
inválido 1 arquivo alterado, 1 inserção (+), 1 exclusão (-)
```

A segunda desenvolvedora, Jessica, faz a mesma coisa — clona o repositório e confirma uma alteração:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/ $
vim TODO $ git
commit -am 'Adicionar tarefa de
redefinição' [master fbff5bc] Adicionar
tarefa de redefinição 1 arquivos alterados, 1 inserções (+), 0 exclusões (-)
```

Agora, Jessica envia seu trabalho para o servidor, que funciona muito bem:

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

A última linha da saída acima mostra uma mensagem de retorno útil da operação push. O formato básico é <oldref>..<newref> fromref ÿ toref, onde oldref significa a referência antiga, newref significa a nova referência, fromref é o nome da referência local que está sendo enviada e toref é o nome da referência remota que está sendo Atualizada. Você verá uma saída semelhante a esta abaixo nas discussões, portanto, ter uma ideia básica do significado ajudará na compreensão dos vários estados dos repositórios.
Mais detalhes estão disponíveis na documentação do git-push.

Continuando com este exemplo, logo depois, John faz algumas alterações, as envia para seu repositório local e tenta enviá-las para o mesmo servidor:

Machine Translated by Google

```
# John's Machine
$ git push origin master
Para john@githost:simplegit.git !
[rejeitado] erro master -> master (não avanço rápido): falha ao enviar
algumas referências para 'john@githost:simplegit.git'
```

Nesse caso, o push de John falha devido ao push anterior de Jéssica de *suas* alterações. Isso é especialmente importante para entender se você está acostumado com o Subversion, porque você notará que os dois desenvolvedores não editaram o mesmo arquivo. Embora o Subversion faça essa mesclagem automaticamente no servidor se arquivos diferentes forem editados, com o Git, você deve *primeiro* mesclar os commits localmente. Em outras palavras, John deve primeiro buscar as alterações upstream de Jessica e mesclá-las em seu repositório local antes de poder enviar por push.

Como primeiro passo, John busca o trabalho de Jessica (isso só *busca* o trabalho upstream de Jessica, ainda não o mescla com o trabalho de John):

```
$ git buscar origem
...
De john@githost:simplegit +
049d078...fbff5bc mestre          -> origem/mestre
```

Neste ponto, o repositório local de John se parece com isto:

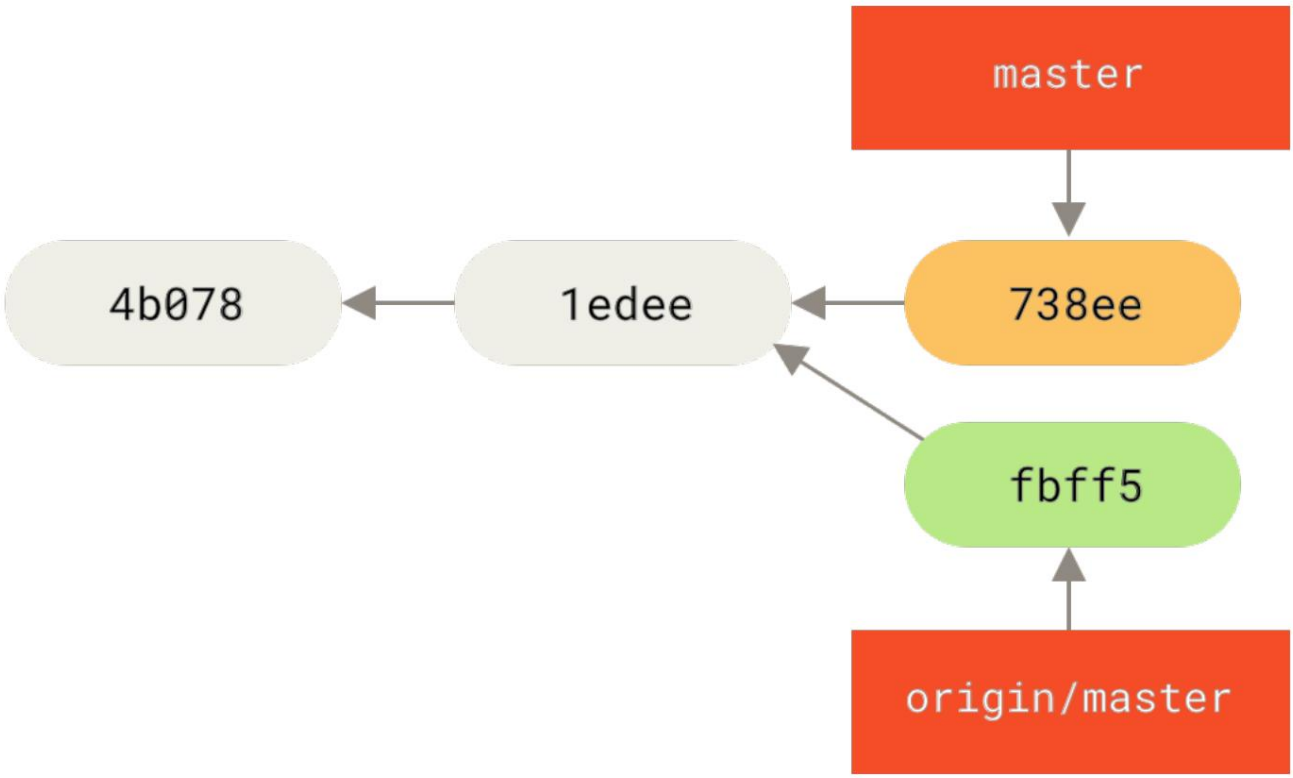


Figura 57. A história divergente de John

Agora John pode mesclar o trabalho de Jessica que ele buscou em seu próprio trabalho local:

Machine Translated by Google

```
$ git merge origin/master
Merge feita pela estratégia 'recursiva'.
TODO | 1 + 1
arquivos alterados, 1 inserções (+), 0 exclusões (-)
```

Contanto que a mesclagem local ocorra sem problemas, o histórico atualizado de John ficará assim:

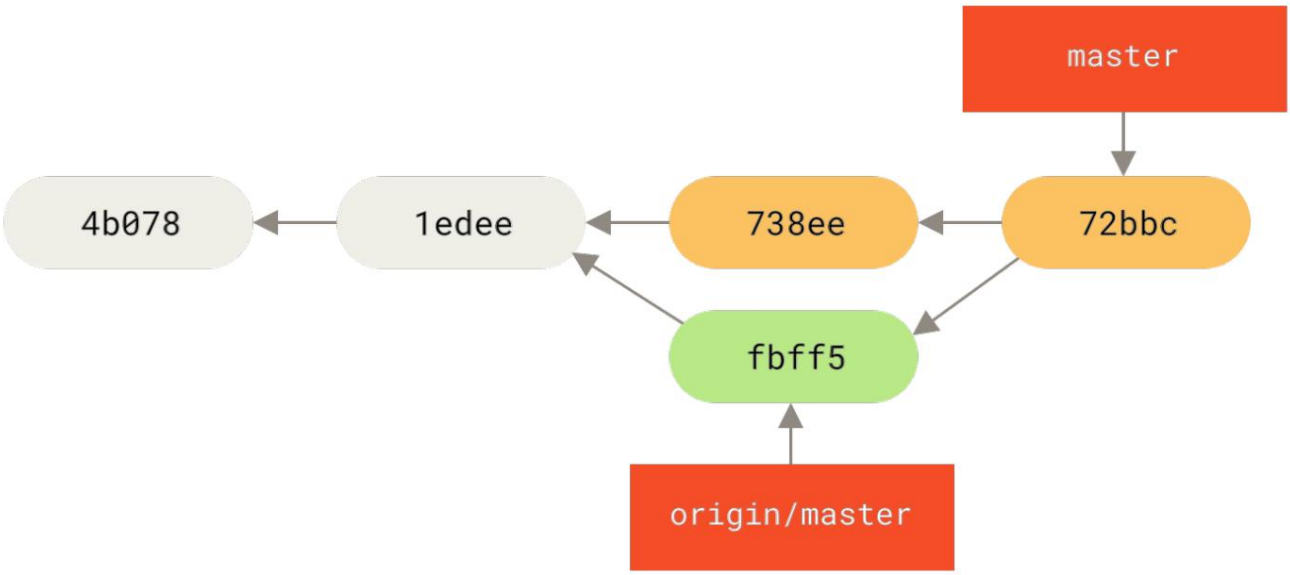


Figura 58. Repositório de John após mesclar origin/master

Nesse ponto, John pode querer testar esse novo código para garantir que nenhum dos trabalhos de Jessica afete o dele e, desde que tudo pareça bem, ele pode finalmente enviar o novo trabalho mesclado para o servidor:

```
$ git push mestre de origem
...
Para john@githost:simplegit.git
fbff5bc..72bbc59 master -> master
```

No final, o histórico de commits de John ficará assim:

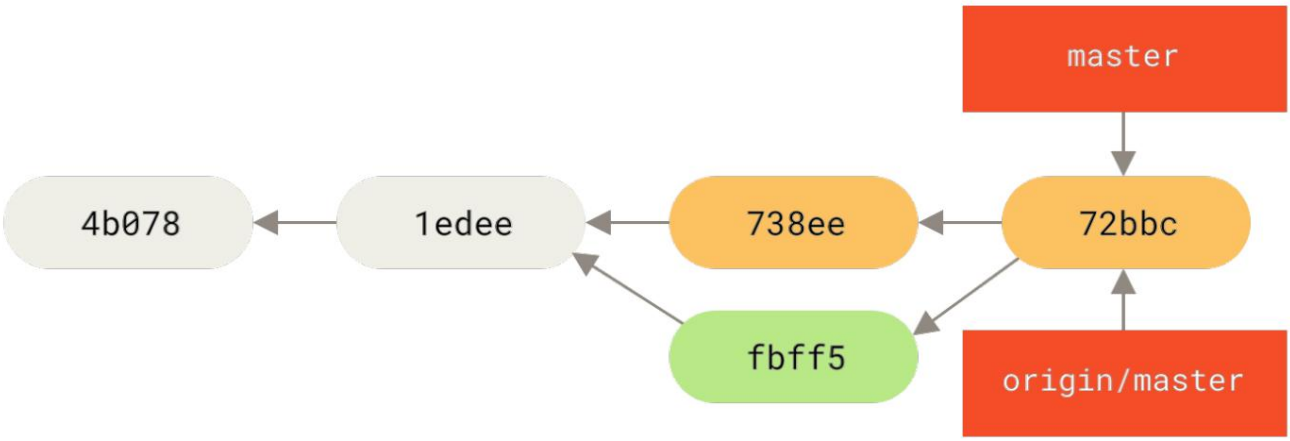


Figura 59. Histórico de John depois de enviar para o servidor de origem

Machine Translated by Google

Nesse ínterim, Jessica criou um novo branch de tópico chamado issue54 e fez três commits nesse branch. Ela ainda não buscou as alterações de John, então seu histórico de commits é assim:

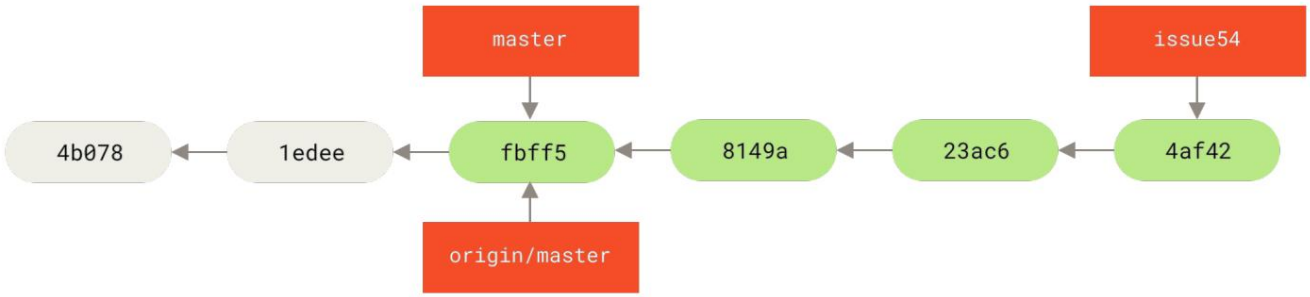


Figura 60. Ramificação de tópicos de Jessica

De repente, Jessica descobre que John enviou algum novo trabalho para o servidor e ela quer dar uma olhada nele, para que ela possa buscar todo o novo conteúdo do servidor que ela ainda não possui com:

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

Isso puxa para baixo o trabalho que John impulsionou nesse meio tempo. A história de Jessica agora se parece com isso:

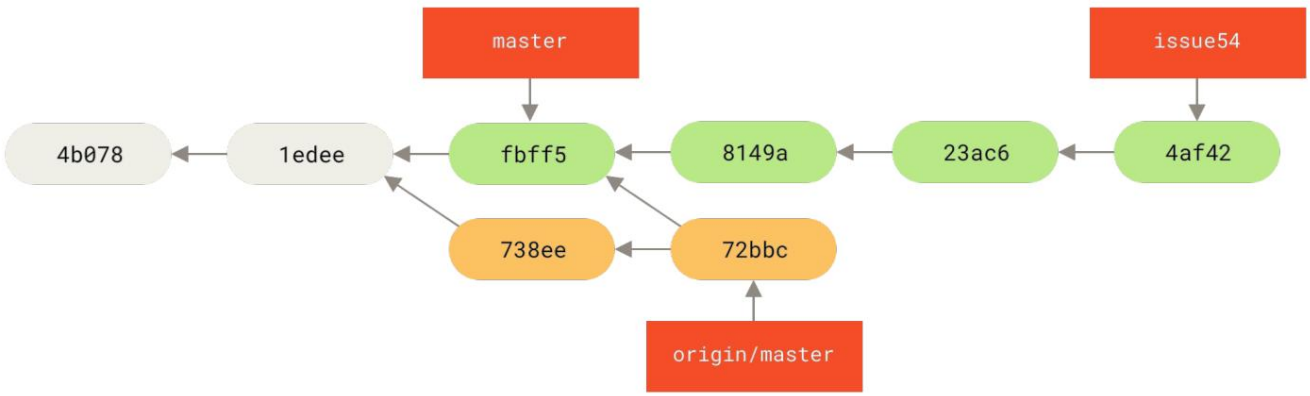


Figura 61. Histórico de Jessica após buscar as alterações de John

Jéssica acha que sua ramificação de tópico está pronta, mas ela quer saber que parte do trabalho buscado de John ela precisa mesclar em seu trabalho para que ela possa enviar. Ela executa git log para descobrir:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Autor: John Smith <jsmith@example.com> Data:
Sex 29 de maio 16:01:27 2009 -0700

Remover valor padrão inválido
```

A sintaxe issue54..origin/master é um filtro de log que pede ao Git para exibir apenas os commits que estão na última ramificação (neste caso origin/master) que não estão na primeira ramificação (neste caso

Machine Translated by Google

problema54). Analisaremos essa sintaxe em detalhes em Intervalos de confirmação.

A partir da saída acima, podemos ver que há um único commit que John fez que Jessica não fundiu em seu trabalho local. Se ela mesclar origin/master, esse é o único commit que modificará seu trabalho local.

Agora, Jessica pode mesclar seu trabalho de tópico em seu branch master , mesclar o trabalho de John (origin/master) em seu branch master e, em seguida, enviar de volta para o servidor novamente.

Primeiro (tendo confirmado todo o trabalho em seu branch de tópico issue54), Jessica volta para seu branch master em preparação para integrar todo esse trabalho:

```
$ git checkout master
Alterado para branch 'master'
Sua ramificação está atrás de 'origin/master' por 2 commits e pode ser avançada.
```

Jessica pode mesclar origin/master ou issue54 primeiro — ambos são upstream, então a ordem não importa. O instantâneo final deve ser idêntico, independentemente da ordem que ela escolher; apenas a história será diferente. Ela escolhe mesclar a ramificação issue54 primeiro:

```
$ git merge issue54
Atualizando fbff5bc..4af4298
Avanço rápido README |
lib/simplegit.rb | 2 arquivos1 +
alterados, 6 inserções (+),6++++-
1 exclusões (-)
```

Não ocorrem problemas; como você pode ver, foi uma mesclagem simples de avanço rápido. Jessica agora conclui o processo de mesclagem local mesclando o trabalho buscado anteriormente de John que está no branch origin/master :

```
$ git merge origin/master
Mesclagem automática lib/
simplegit.rb Mesclagem feita pela estratégia
'recursiva'. lib/simplegit.rb2 | 1 -arquivos
alterados, 1 inserções (+), 1 exclusões (-)
```

Tudo se funde de forma limpa, e a história de Jessica agora se parece com isso:

Machine Translated by Google

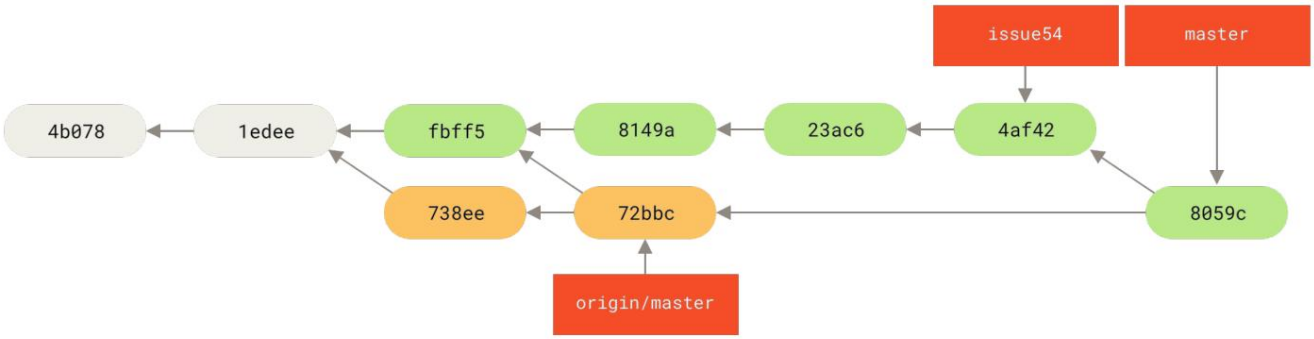


Figura 62. Histórico de Jessica após mesclar as alterações de John

Agora origin/master pode ser acessado a partir do branch master de Jessica , então ela deve ser capaz de enviar com sucesso (supondo que John não tenha enviado ainda mais alterações nesse meio tempo):

```
$ git push mestre de origem
...
Para jessica@githost:simplegit.git
72bbc59..8059c15 master -> master
```

Cada desenvolvedor fez commit algumas vezes e fundiu o trabalho um do outro com sucesso.

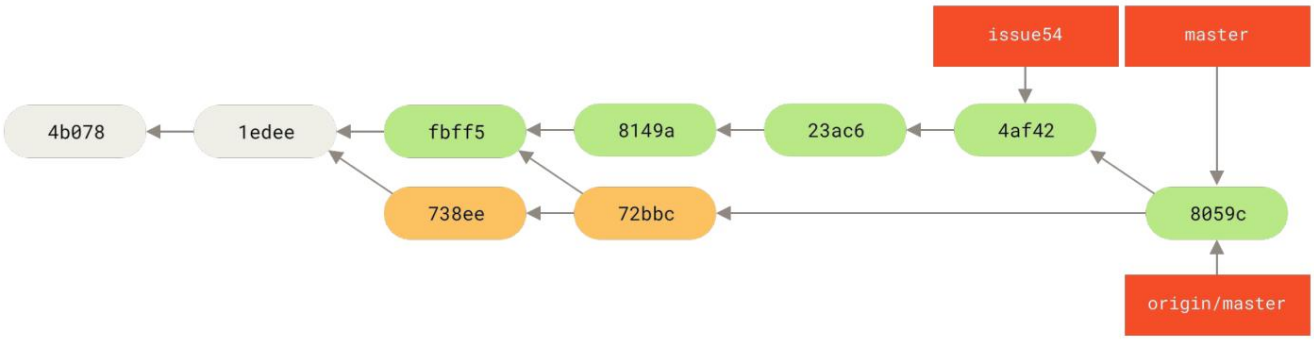


Figura 63. Histórico de Jessica após enviar todas as alterações de volta ao servidor

Esse é um dos fluxos de trabalho mais simples. Você trabalha por um tempo (geralmente em um branch de tópico) e mescla esse trabalho em seu branch master quando estiver pronto para ser integrado. Quando você deseja compartilhar esse trabalho, você busca e mescla seu mestre de origem/mestre se ele foi alterado e, finalmente, envia para a ramificação mestre no servidor. A sequência geral é mais ou menos assim:

Machine Translated by Google

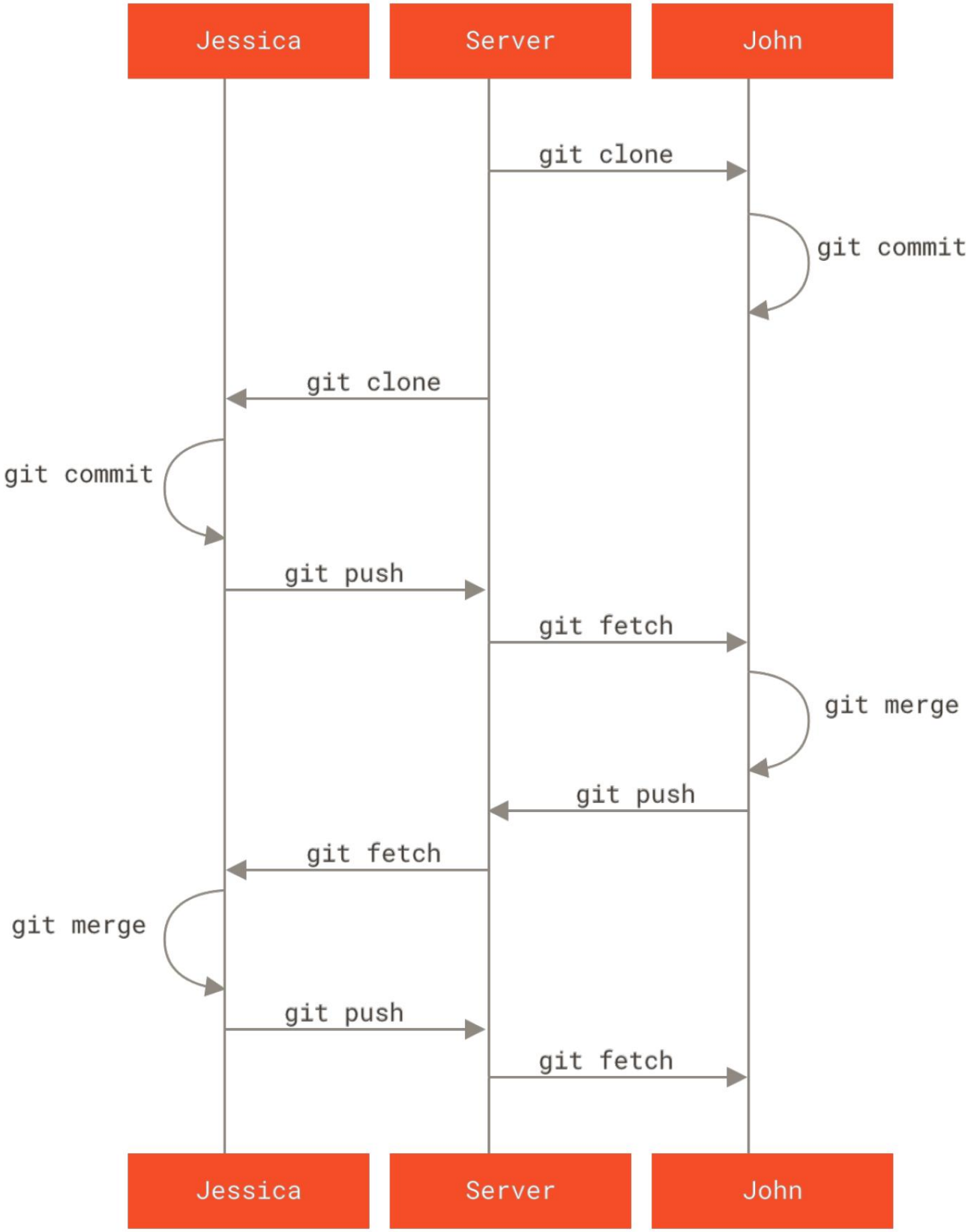


Figura 64. Sequência geral de eventos para um fluxo de trabalho Git simples de vários desenvolvedores

Equipe Gerenciada Privada

Neste próximo cenário, você verá as funções de colaborador em um grupo privado maior. Você aprenderá como trabalhar em um ambiente onde pequenos grupos colaboram em recursos, após o que essas contribuições baseadas em equipe são integradas por outra parte.

Machine Translated by Google

Digamos que John e Jessica estejam trabalhando juntos em um recurso (chame-o de “recursoA”), enquanto Jessica e uma terceira desenvolvedora, Josie, estejam trabalhando em um segundo (digamos, “recursoB”). Nesse caso, a empresa está usando um tipo de fluxo de trabalho do gerenciador de integração em que o trabalho dos grupos individuais é integrado apenas por determinados engenheiros, e a ramificação principal do repositório principal pode ser atualizada apenas por esses engenheiros. Nesse cenário, todo o trabalho é feito em ramificações baseadas em equipe e posteriormente reunido pelos integradores.

Vamos acompanhar o fluxo de trabalho de Jessica enquanto ela trabalha em seus dois recursos, colaborando em paralelo com dois desenvolvedores diferentes neste ambiente. Supondo que ela já tenha seu repositório clonado, ela decide trabalhar primeiro no featureA . Ela cria uma nova ramificação para o recurso e faz algum trabalho isso aí:

```
# Jessica's Machine
$ git checkout -b featureA
Mudou para uma nova ramificação
'featureA' $ vim lib/simplegit.rb $ git
commit -am 'Adicionar limite à função de
log' [featureA 3300904] Adicionar limite à função
de log 1 arquivos alterados, 1 inserções(+), 1 exclusões(-)
```

Nesse ponto, ela precisa compartilhar seu trabalho com John, então ela envia seus commits do branch featureA para o servidor. Jéssica não tem acesso push ao branch master — apenas os integradores têm — então ela tem que fazer push para outro branch para colaborar com John:

```
$ git push -u origem featureA
...
Para jessica@githost:simplegit.git *
[novo ramo]                recursoA -> recursoA
```

Jessica envia um e-mail para John para dizer a ele que ela enviou algum trabalho para uma ramificação chamada featureA e ele pode vê-lo agora. Enquanto espera pelo feedback de John, Jessica decide começar a trabalhar no recurso B com Josie. Para começar, ela inicia um novo branch de recursos, baseando-se no branch master do servidor:

```
# Jessica's Machine
$ git fetch origin $ git
checkout -b featureB origin/master
Mudou para uma nova ramificação 'featureB'
```

Agora, Jessica faz alguns commits no branch featureB :

```
$ vim lib/simplegit.rb $
git commit -am 'Tornar a função ls-tree recursiva' [recursoB
e5b0fdc] Tornar a função ls-tree recursiva 1 arquivo
alterado, 1 inserção(+), 1 exclusão(-) $ vim lib/simplegit .rb
```

Machine Translated by Google

```
$ git commit -am 'Adicionar ls-
 files' [featureB 8512791] Adicionar
ls-files 1 arquivo alterado, 5 inserções(+), 0 exclusões(-)
```

O repositório de Jessica agora se parece com isso:

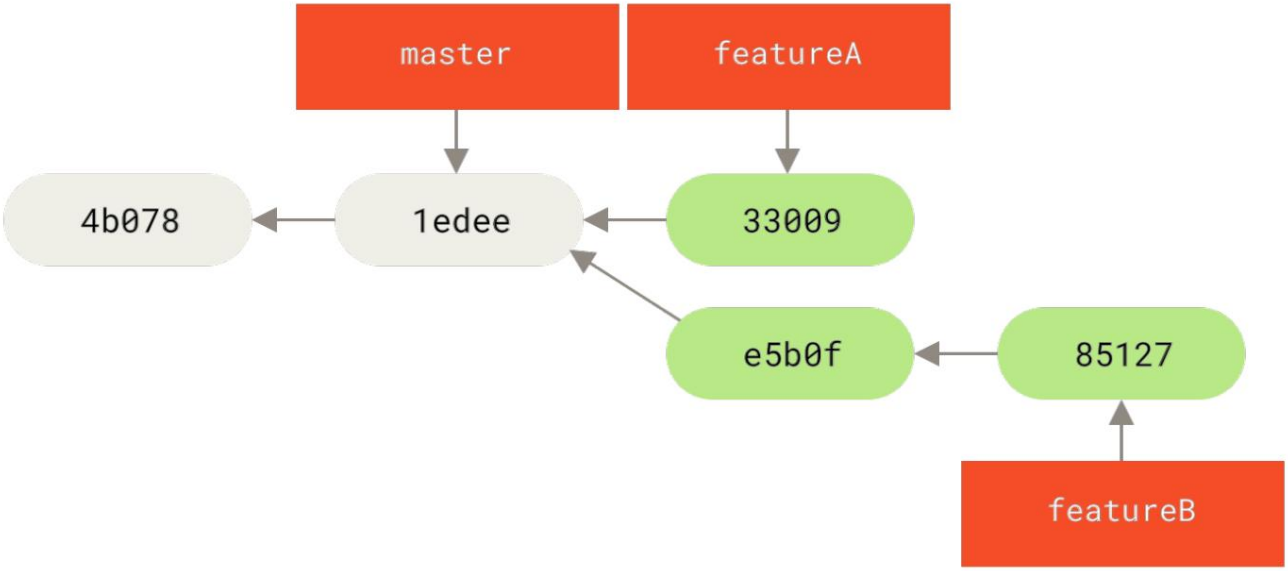


Figura 65. Histórico de confirmação inicial de Jessica

Ela está pronta para enviar seu trabalho, mas recebe um e-mail de Josie informando que uma ramificação com algum trabalho inicial "featureB" já foi enviada para o servidor como a ramificação featureBee . Jessica precisa mesclar essas alterações com as dela antes de enviar seu trabalho para o servidor. Jessica primeiro busca as mudanças de Josie com git fetch:

```
$ git buscar origem
 ...
De jessica@githost:simplegit *
[novo ramo] featureBee -> origem/
                          featureBee
```

Supondo que Jessica ainda esteja em seu branch featureB verificado , ela agora pode mesclar o trabalho de Josie nesse branch com git merge:

```
$ git merge origem/featureBee
Mesclagem automática lib/
simplegit.rb Mesclagem feita pela estratégia
'recursiva'. lib/simplegit.rb | 4 +++ 1
alterados, 4 inserções(+), 0 exclusões(-)
```

Nesse ponto, Jéssica deseja enviar todo esse trabalho "featureB" mesclado de volta ao servidor, mas ela não deseja simplesmente enviar por push sua própria ramificação featureB . Em vez disso, como Josie já iniciou uma ramificação upstream featureBee , Jessica deseja enviar para *essa* ramificação, o que ela faz com:

```
$ git push -u origem featureB:featureBee
```

Machine Translated by Google

```
...
Para jessica@githost:simplegit.git
fba9af8..cd685d1 featureB -> featureBee
```

Isso é chamado de *refspec.* Veja The Refspec para uma discussão mais detalhada sobre as refspecs do Git e as diferentes coisas que você pode fazer com elas. Observe também o sinalizador -u ; é a abreviação de --set-upstream, que configura as ramificações para facilitar o push e pull posteriormente.

De repente, Jessica recebe um e-mail de John, que diz a ela que fez algumas alterações no branch featureA no qual eles estão colaborando e pede a Jessica para dar uma olhada nelas. Novamente, Jessica executa um simples git fetch para buscar *todo* o novo conteúdo do servidor, incluindo (é claro) o trabalho mais recente de John:

```
$ git buscar origem
...
De jessica@githost:simplegit
3300904..aad881d featureA -> origin/featureA
```

Jessica pode exibir o log do novo trabalho de John comparando o conteúdo do branch featureA recém-obtido com sua cópia local do mesmo branch:

```
$ git log featureA..origin/featureA commit
aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Autor: John Smith <jsmith@example.com>
Data: sexta-feira, 29 de maio 19:57:33 2009 -0700

Aumente a saída de log para 30 de 25
```

Se Jessica gostar do que vê, ela pode mesclar o novo trabalho de John em sua ramificação local featureA com:

```
$ git checkout featureA
Mudou para branch 'featureA' $
git merge origin/featureA
Atualizando 3300904..aad881d
Avanço rápido lib/simplegit.rb | 1
arquivos alterados, 9 inserções 10++++++++-
(+), 1 exclusões (-)
```

Por fim, Jessica pode querer fazer algumas pequenas alterações em todo o conteúdo mesclado, para que ela fique livre para fazer essas alterações, confirmá-las em sua ramificação featureA local e enviar o resultado final de volta ao servidor:

```
$ git commit -am 'Adicionar pequeno ajuste ao conteúdo
mesclado' [featureA 774b3ed] Adicionar pequeno ajuste ao
conteúdo mesclado 1 arquivo alterado, 1 inserção(+), 1
exclusão(-) $ git push
```

Machine Translated by Google

```
...
Para jessica@githost:simplegit.git
3300904..774b3ed featureA -> featureA
```

O histórico de commits de Jessica agora se parece com isto:

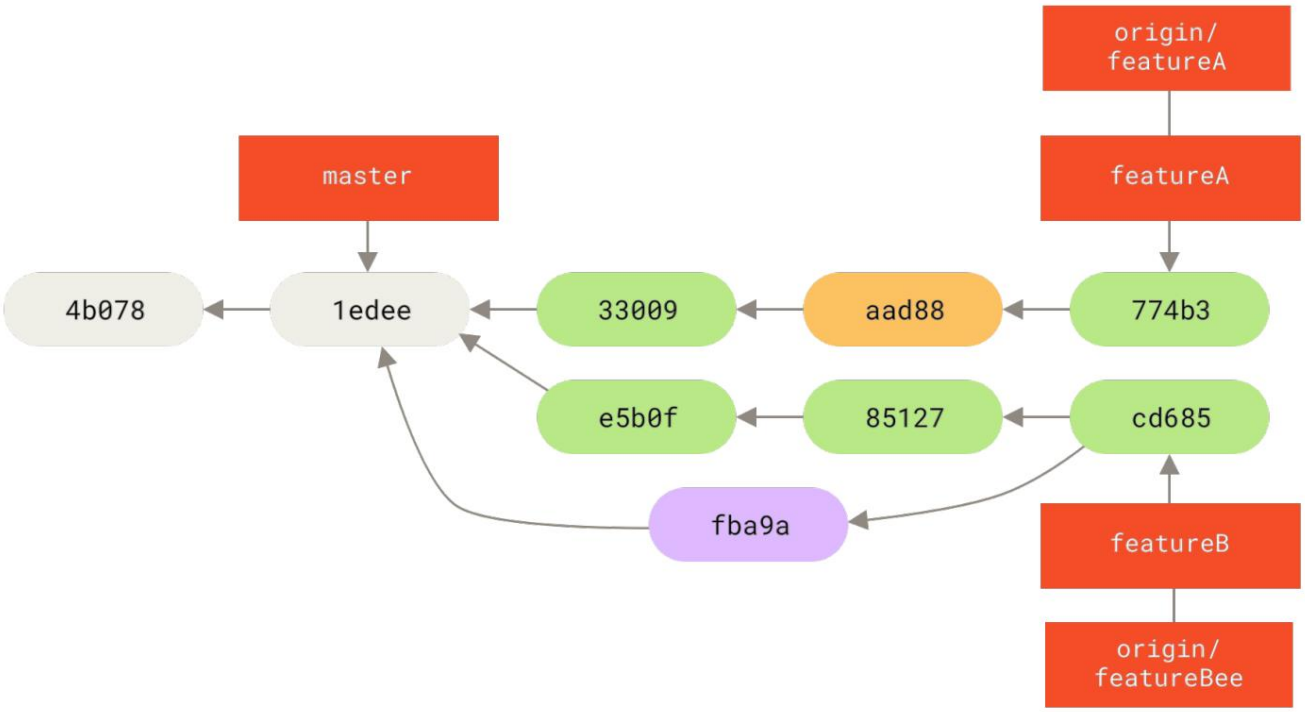


Figura 66. Histórico de Jessica após o commit em um feature branch

Em algum momento, Jessica, Josie e John informam aos integradores que as ramificações featureA e featureBee no servidor estão prontas para integração na linha principal. Depois que os integradores mesclarem essas ramificações na linha principal, uma busca trará o novo commit de mesclagem, fazendo com que o histórico fique assim:

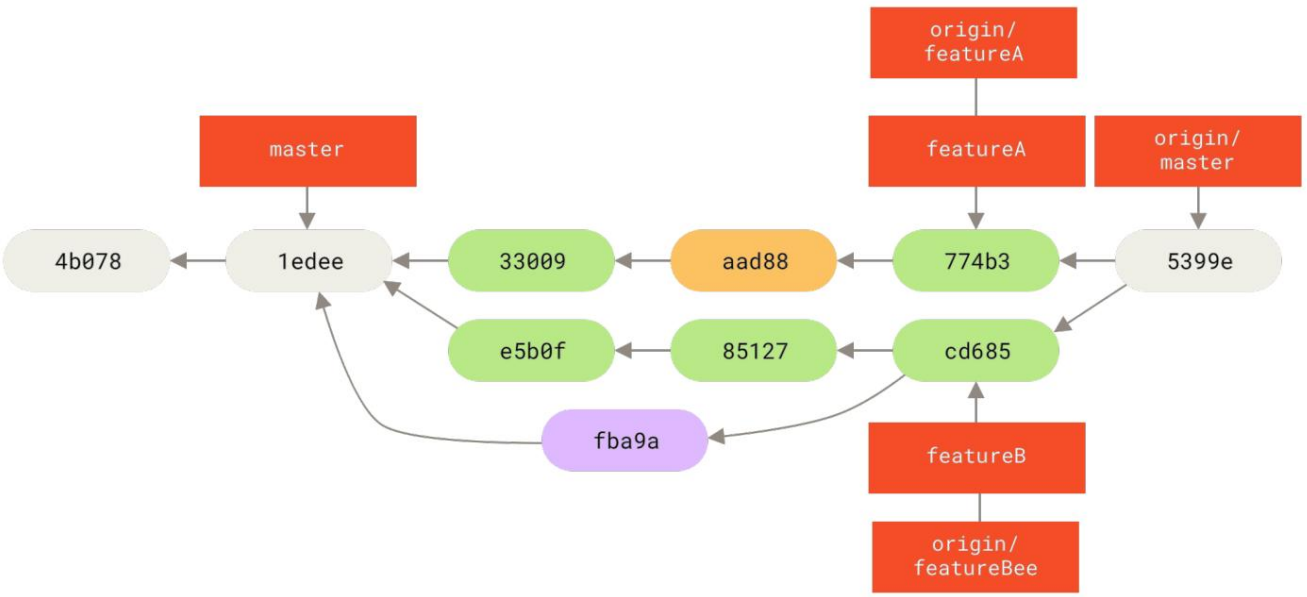


Figura 67. A história de Jessica após a fusão de ambas as ramificações de tópico

Machine Translated by Google

Muitos grupos mudam para o Git por causa dessa capacidade de ter várias equipes trabalhando em paralelo, mesclando as diferentes linhas de trabalho no final do processo. A capacidade de subgrupos menores de uma equipe de colaborar por meio de ramificações remotas sem necessariamente envolver ou impedir toda a equipe é um grande benefício do Git. A sequência do fluxo de trabalho que você viu aqui é mais ou menos assim:

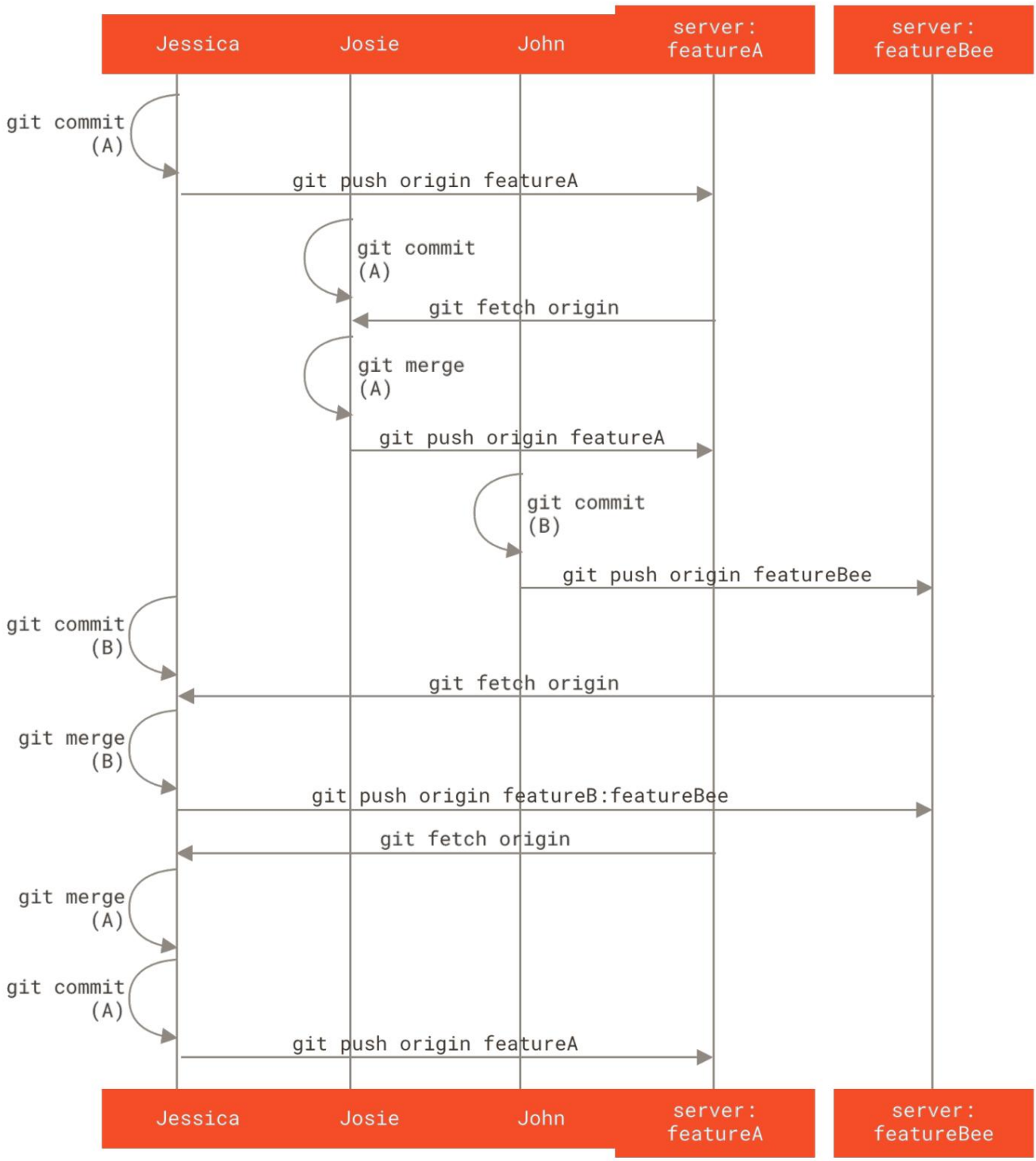


Figura 68. Sequência básica deste fluxo de trabalho de equipe gerenciada

Projeto público bifurcado

Contribuir para projetos públicos é um pouco diferente. Como você não tem permissão para atualizar ramificações diretamente no projeto, você precisa enviar o trabalho aos mantenedores de outra maneira. Este primeiro exemplo descreve a contribuição por bifurcação em hosts Git que oferecem suporte à bifurcação fácil. Muitos

Machine Translated by Google

sites de hospedagem suportam isso (incluindo GitHub, BitBucket, repo.or.cz e outros), e muitos mantenedores de projetos esperam esse estilo de contribuição. A próxima seção lida com projetos que preferem aceitar correções enviadas por e-mail.

Primeiro, você provavelmente desejará clonar o repositório principal, criar uma ramificação de tópico para o patch ou série de patches com o qual planeja contribuir e fazer seu trabalho lá. A sequência é basicamente assim:

```
$ git clone <url>
$ cd project $ git
checkout -b featureA ...
work ... $ git commit ...
work ... $ git commit
```

ÿ

Você pode querer usar rebase -i para compactar seu trabalho em um único commit, ou reorganizar o trabalho nos commits para tornar o patch mais fácil para o mantenedor rever — veja Reescrevendo o Histórico para mais informações sobre rebase interativo.

Quando seu trabalho de branch terminar e você estiver pronto para contribuir com ele de volta para os mantenedores, vá para a página original do projeto e clique no botão “Fork”, criando seu próprio fork gravável do projeto.
Você então precisa adicionar este URL de repositório como um novo controle remoto de seu repositório local; neste exemplo, vamos chamá-lo de myfork:

```
$ git remote add myfork <url>
```

Em seguida, você precisa enviar seu novo trabalho para este repositório. É mais fácil enviar o branch de tópico no qual você está trabalhando para seu repositório bifurcado, em vez de mesclar esse trabalho em seu branch principal e enviá-lo por push. O motivo é que, se seu trabalho não for aceito ou for escolhido a dedo, você não precisa rebobinar sua ramificação mestre (a operação de seleção a dedo do Git é abordada com mais detalhes em Fluxos de trabalho de rebasing e seleção a dedo). Se os mantenedores fundirem, rebasearem ou selecionarem seu trabalho, você eventualmente o recuperará puxando de seu repositório de qualquer maneira.

De qualquer forma, você pode empurrar seu trabalho com:

```
$ git push -u myfork featureA
```

Depois que seu trabalho for enviado para a bifurcação do repositório, você precisará notificar os mantenedores do projeto original de que possui um trabalho que gostaria que eles mesclassem. Isso geralmente é chamado de *pull request,* e você normalmente gera tal solicitação por meio do site — o GitHub tem seu próprio mecanismo de “Pull Request” que veremos no GitHub — ou você pode executar o comando git request-pull e enviar por e-mail a saída subsequente para o mantenedor do projeto manualmente.

O comando git request-pull leva o branch base para o qual você deseja que seu branch de tópico seja puxado e o URL do repositório Git do qual você deseja que eles extraiam e produz um resumo de todas as alterações que você está pedindo para serem puxadas. Por exemplo, se Jéssica deseja enviar uma solicitação pull para John e

Machine Translated by Google

ela fez dois commits no branch do tópico que ela acabou de enviar, ela pode rodar isso:

```
$ git request-pull origin/master myfork
As seguintes alterações desde o commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jéssica Smith (1):
          Criar nova função

estão disponíveis no repositório git em:

git://githost/simplegit.git featureA

Jéssica Smith (2):
        Adicionar limite à função de log
        Aumente a saída de log para 30 de 25

lib/simplegit.rb | 1 arquivos      10++++++++-
alterados, 9 inserções (+), 1 exclusões (-)
```

Essa saída pode ser enviada para o mantenedor — informa de onde o trabalho foi ramificado, resume os commits e identifica de onde o novo trabalho deve ser puxado.

Em um projeto do qual você não é o mantenedor, geralmente é mais fácil ter uma ramificação como master sempre rastreando origin/master e fazer seu trabalho em ramificações de tópico que você pode descartar facilmente se forem rejeitadas. Ter temas de trabalho isolados em ramificações de tópico também facilita o rebase do seu trabalho se a ponta do repositório principal tiver sido movida nesse meio tempo e seus commits não forem mais aplicados corretamente. Por exemplo, se você deseja enviar um segundo tópico de trabalho para o projeto, não continue trabalhando no branch do tópico que você acabou de enviar — recomece a partir do branch master do repositório principal :

```
$ git checkout -b featureB origin/master ... trabalho ... $
git commit $ git push myfork featureB $ git request-pull
origin/master myfork ... e-mail gerado solicitação pull
para o mantenedor ... $ git fetch origin
```

Agora, cada um de seus tópicos está contido em um silo — semelhante a uma fila de patches — que você pode reescrever, rebasear e modificar sem que os tópicos interfiram ou dependam uns dos outros, assim:

Machine Translated by Google

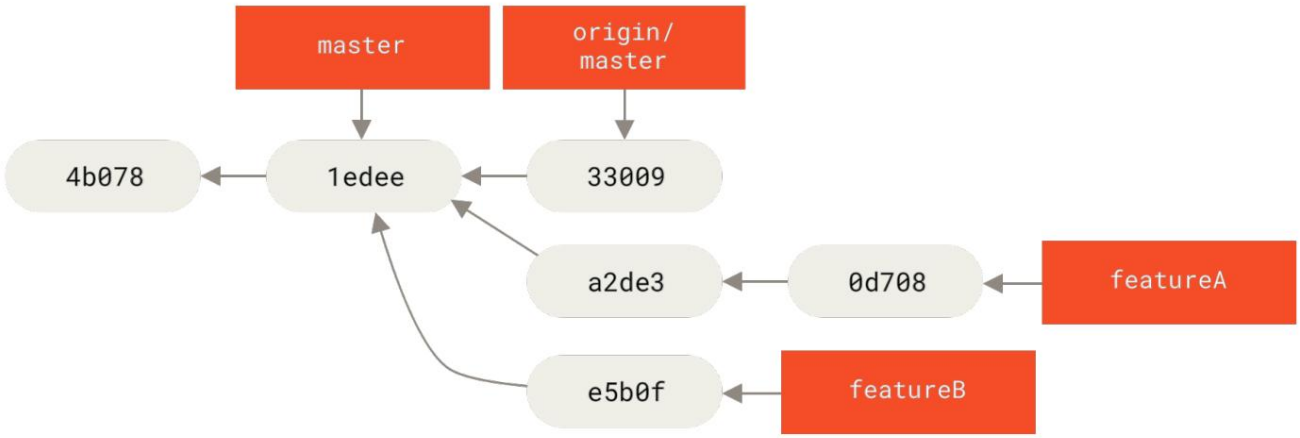


Figura 69. Histórico de confirmação inicial com o trabalho featureB

Digamos que o mantenedor do projeto tenha obtido vários outros patches e tentado sua primeira ramificação, mas ela não mescla mais corretamente. Nesse caso, você pode tentar fazer o rebase dessa ramificação em cima da origem/mestre, resolver os conflitos para o mantenedor e reenviar suas alterações:

```
$ git checkout featureA
$ git rebase origin/master $
git push -f myfork featureA
```

Isso reescreve seu histórico para agora se parecer com Commit history após featureA work.

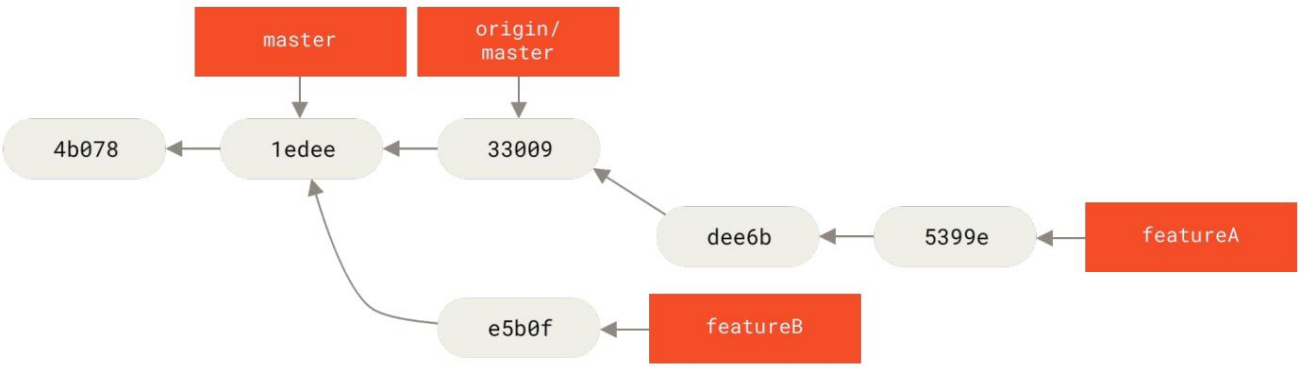


Figura 70. Histórico de confirmação após o trabalho do featureA

Como você rebaseou o branch, é necessário especificar o -f para o comando push para poder substituir o branch featureA no servidor por um commit que não seja descendente dele. Uma alternativa seria enviar esse novo trabalho para uma ramificação diferente no servidor (talvez chamada de featureAv2).

Vejamos mais um cenário possível: o mantenedor olhou para o trabalho em sua segunda ramificação e gostou do conceito, mas gostaria que você mudasse um detalhe de implementação. Você também aproveitará esta oportunidade para mover o trabalho a ser baseado no branch master atual do projeto . Você inicia uma nova ramificação com base na ramificação original/mestre atual , esmaga as alterações do recurso B , resolve quaisquer conflitos, faz a alteração de implementação e, em seguida, envia-a como uma nova ramificação:

Machine Translated by Google

```
$ git checkout -b featureBv2 origin/master $
git merge --squash featureB ... alterar
implementação ... $ git commit $ git push
myfork featureBv2
```

A opção --squash pega todo o trabalho na ramificação mesclada e a compacta em um conjunto de alterações, produzindo o estado do repositório como se uma mesclagem real tivesse acontecido, sem realmente fazer uma confirmação de mesclagem. Isso significa que seu commit futuro terá apenas um pai e permite que você introduza todas as alterações de outro branch e faça mais alterações antes de gravar o novo commit.
Além disso, a opção --no-commit pode ser útil para atrasar a confirmação da mesclagem no caso do processo de mesclagem padrão.

Neste ponto, você pode notificar o mantenedor de que você fez as alterações solicitadas e que ele pode encontrar essas alterações em sua ramificação featureBv2 .

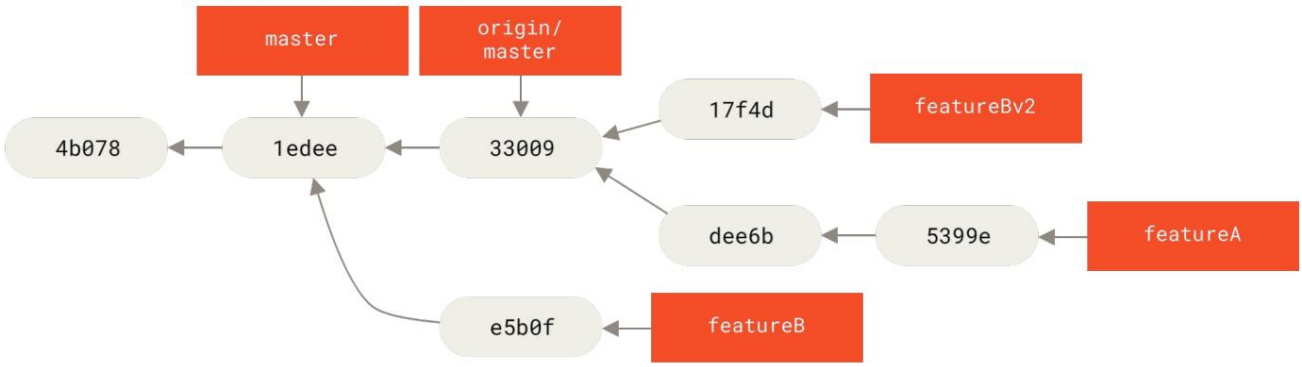


Figura 71. Histórico de confirmação após o trabalho do featureBv2

Projeto público por e-mail

Muitos projetos têm procedimentos estabelecidos para aceitação de patches — você precisará verificar as regras específicas de cada projeto, pois elas serão diferentes. Como existem vários projetos maiores e mais antigos que aceitam patches por meio de uma lista de discussão de desenvolvedores, examinaremos um exemplo disso agora.

O fluxo de trabalho é semelhante ao caso de uso anterior — você cria ramificações de tópicos para cada série de patches em que trabalha. A diferença é como você os envia ao projeto. Em vez de bifurcar o projeto e enviar para sua própria versão gravável, você gera versões de e-mail de cada série de commits e as envia por e-mail para a lista de discussão do desenvolvedor:

```
$ git checkout -b tópicoA ...
trabalho ...
$ git
commit ...
trabalho ... $ git commit
```

Agora você tem dois commits que deseja enviar para a lista de discussão. Você usa o git format-patch para gerar os arquivos no formato mbox que você pode enviar por e-mail para a lista — ele transforma cada commit em um

Machine Translated by Google

mensagem de e-mail com a primeira linha da mensagem de commit como assunto e o restante da mensagem mais o patch que o commit introduz como corpo. O bom disso é que aplicar um patch de um e-mail gerado com format-patch preserva todas as informações do commit corretamente.

```
$ git format-patch -M origin/master 0001-
add-limit-to-log-function.patch 0002-increase-
log-output-to-30-from-25.patch
```

O comando format-patch imprime os nomes dos arquivos de patch que ele cria. A opção -M diz ao Git para procurar renomeações. Os arquivos ficam assim:

```
$ cat 0001-add-limit-to-log-function.patch De
330090432754092d704da8e76ca5c05c198e71a8 Seg 17 de setembro 00:00:00
2001 De: Jessica Smith <jessica@example.com> Data: Dom, 6 de abril de 2008
10:17:23 - 0700 Subject: [PATCH 1/2] Adicionar limite à função de registro


 Limite a funcionalidade de registro aos primeiros 20

 ---
lib/simplegit.rb | 2 +- 1 arquivos
alterados, 1 inserções(+), 1 exclusões(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb índice
 76f47bc..f9815f1 100644
 --- a/lib/simplegit.rb +++ b/
lib/simplegit.rb @@ -14,7
+14,7 @@ class SimpleGit end


def log(árvore = 'mestre')
-       command("git log #{treeish}")
+       command("git log -n 20 #{treeish}") end


def ls_tree(treeish = 'mestre')
 --
 2.1.0
```

Você também pode editar esses arquivos de patch para adicionar mais informações para a lista de e-mail que você não deseja que apareça na mensagem de confirmação. Se você adicionar texto entre a linha --- e o início do patch (a linha diff --git), os desenvolvedores poderão lê-lo, mas esse conteúdo será ignorado pelo processo de patching.

Para enviar por e-mail para uma lista de discussão, você pode colar o arquivo em seu programa de e-mail ou enviá-lo por meio de um programa de linha de comando. Colar o texto geralmente causa problemas de formatação, especialmente com clientes “mais inteligentes” que não preservam as novas linhas e outros espaços em branco adequadamente. Felizmente, o Git fornece uma ferramenta para ajudá-lo a enviar patches formatados corretamente via IMAP, o que pode ser mais fácil para você. Demonstraremos como enviar um patch via Gmail, que por acaso é o agente de e-mail que conhecemos melhor;

Machine Translated by Google

você pode ler instruções detalhadas para vários programas de e-mail no final do arquivo Documentation/SubmittedPatches mencionado acima no código-fonte do Git.

Primeiro, você precisa configurar a seção imap em seu arquivo ~/.gitconfig . Você pode definir cada valor separadamente com uma série de comandos git config ou pode adicioná-los manualmente, mas no final seu arquivo de configuração deve se parecer com isto:

```
[imap]
pasta = "[Gmail]/Rascunhos"
host = imaps://imap.gmail.com user
= user@gmail.com pass =
YX]8g76G_2^sFbd port = 993
sslverify = false
```

Se o seu servidor IMAP não usar SSL, as duas últimas linhas provavelmente não serão necessárias e o valor do host será imap:// em vez de imaps://. Quando isso estiver configurado, você pode usar git imap-send para colocar a série de patches na pasta Rascunhos do servidor IMAP especificado:

```
$ cat *.patch |git imap-send
Resolvendo imap.gmail.com... ok
Conectando a [74.125.142.109]:993... ok
Fazendo login... enviando 2 mensagens 100%
(2/2) concluído
```

Neste ponto, você deve ser capaz de ir para sua pasta Rascunhos, alterar o campo Para para a lista de discussão para a qual você está enviando o patch, possivelmente CC do mantenedor ou pessoa responsável por essa seção e enviá-lo.

Você também pode enviar os patches por meio de um servidor SMTP. Como antes, você pode definir cada valor separadamente com uma série de comandos git config ou pode adicioná-los manualmente na seção sendemail em seu arquivo ~/.gitconfig :

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

Feito isso, você pode usar git send-email para enviar seus patches:

```
$ git send-email *.patch
0001-add-limit-to-log-function.patch 0002-
increase-log-output-to-30-from-25.patch De quem
os e-mails devem parecer ser? [Jessica Smith <jessica@example.com>]
```

Machine Translated by Google

```
Os e-mails serão enviados de: Jessica Smith <jessica@example.com> Para quem os e-
mails devem ser enviados? jessica@example.com Message-ID a ser usado como In-
Reply-To para o primeiro e-mail? y
```

Em seguida, o Git mostra um monte de informações de log parecidas com esta para cada patch que você está enviando:

```
(mbox) Adicionando cc: Jessica Smith <jessica@example.com> de \line 'De: Jessica
Smith <jessica@example.com>'
OK. Log diz:
Sendmail: /usr/sbin/sendmail -i jessica@example.com De: Jessica Smith
<jessica@example.com> Para: jessica@example.com Subject: [PATCH
1/2] Adicionar limite à data da função de log : Sáb, 30 de maio de 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> Referências: <y>



Resultado: OK
```

ÿ Para obter ajuda sobre como configurar seu sistema e e-mail, mais dicas e truques e uma sandbox para enviar um patch de avaliação por e-mail, acesse git-send-email.io.

Resumo

Nesta seção, abordamos vários fluxos de trabalho e falamos sobre as diferenças entre trabalhar como parte de uma equipe pequena em projetos de código fechado e contribuir para um grande projeto público. Você sabe verificar se há erros de espaço em branco antes de confirmar e pode escrever uma ótima mensagem de confirmação. Você aprendeu como formatar patches e enviá-los por e-mail para uma lista de discussão de desenvolvedores. Lidar com fusões também foi abordado no contexto dos diferentes fluxos de trabalho. Agora você está bem preparado para colaborar em qualquer projeto.

A seguir, você verá como trabalhar o outro lado da moeda: manter um projeto Git. Você aprenderá como ser um ditador benevolente ou gerente de integração.

Mantendo um Projeto

Além de saber como contribuir efetivamente para um projeto, você provavelmente precisará saber como manter um. Isso pode consistir em aceitar e aplicar patches gerados por meio de patch de formato e enviados por e-mail para você ou integrar alterações em ramificações remotas para repositórios que você adicionou como remotos ao seu projeto. Quer você mantenha um repositório canônico ou queira ajudar verificando ou aprovando patches, você precisa saber como aceitar o trabalho de uma maneira que seja mais clara para outros contribuidores e sustentável para você a longo prazo.

Machine Translated by Google

Trabalhando em Ramos de Tópicos

Quando você está pensando em integrar um novo trabalho, geralmente é uma boa ideia experimentá-lo em uma *ramificação de tópico* — uma ramificação temporária feita especificamente para testar esse novo trabalho. Dessa forma, é fácil ajustar um patch individualmente e deixá-lo se não estiver funcionando até que você tenha tempo de voltar a ele. Se você criar um nome de ramificação simples com base no tema do trabalho que vai tentar, como ruby_client ou algo descritivo semelhante, poderá lembrá-lo facilmente se tiver que abandoná-lo por um tempo e voltar mais tarde. O mantenedor do projeto Git tende a nomear esses branches também — como sc/ruby_client, onde sc é a abreviação da pessoa que contribuiu com o trabalho. Como você deve se lembrar, você pode criar o branch baseado em seu branch master assim:

```
$ git branch sc/ruby_client master
```

Ou, se você também quiser mudar para ele imediatamente, pode usar a opção checkout -b :

```
$ git checkout -b sc/ruby_client mestre
```

Agora você está pronto para adicionar o trabalho de contribuição que recebeu neste branch de tópico e determinar se deseja mesclá-lo em seus branches de longo prazo.

Aplicando patches de e-mail

Se você receber um patch por e-mail que precisa integrar ao seu projeto, será necessário aplicar o patch em sua ramificação de tópico para avaliá-lo. Há duas maneiras de aplicar um patch enviado por e-mail: com git apply ou com git am.

Aplicando um Patch com apply

Se você recebeu o patch de alguém que o gerou com git diff ou alguma variação do comando Unix diff (o que não é recomendado; veja a próxima seção), você pode aplicá-lo com o comando git apply . Assumindo que você salvou o patch em /tmp/patch-ruby-client.patch, você pode aplicar o patch assim:

```
$ git apply /tmp/patch-ruby-client.patch
```

Isso modifica os arquivos em seu diretório de trabalho. É quase idêntico a executar um comando patch -p1 para aplicar o patch, embora seja mais paranóico e aceite menos correspondências difusas do que patch. Ele também lida com acréscimos, exclusões e renomeações de arquivos, se eles forem descritos no formato git diff , o que o patch não fará. Finalmente, git apply é um modelo “aplicar tudo ou abortar tudo” onde tudo é aplicado ou nada é, enquanto patch pode aplicar parcialmente patchfiles, deixando seu diretório de trabalho em um estado estranho. git apply é, em geral, muito mais conservador do que patch. Ele não criará um commit para você — depois de executá-lo, você deve testar e confirmar as alterações introduzidas manualmente.

Você também pode usar git apply para ver se um patch se aplica corretamente antes de tentar aplicá-lo — você pode executar git apply --check com o patch:

Machine Translated by Google

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch error: patch falhou:
ticgit.gemspec:1 erro: ticgit.gemspec: patch não se aplica
```

Se não houver saída, o patch deve ser aplicado corretamente. Este comando também sai com um status diferente de zero se a verificação falhar, portanto, você pode usá-lo em scripts, se desejar.

Aplicando um patch com am

Se o colaborador for um usuário do Git e for bom o suficiente para usar o comando format-patch para gerar seu patch, seu trabalho será mais fácil porque o patch contém informações do autor e uma mensagem de commit para você. Se puder, incentive seus colaboradores a usar format-patch em vez de diff para gerar patches para você. Você só deve usar o git apply para patches legados e coisas assim.

Para aplicar um patch gerado pelo format-patch, você usa git am (o comando é denominado am porque é usado para "aplicar uma série de patches de uma caixa de correio"). Tecnicamente, o git am foi criado para ler um arquivo mbox, que é um formato de texto simples e simples para armazenar uma ou mais mensagens de e-mail em um arquivo de texto. Parece algo assim:

```
De 330090432754092d704da8e76ca5c05c198e71a8 Seg 17 de setembro 00:00:00 2001
De: Jessica Smith <jessica@example.com> Data: Dom, 6 de abril de 2008 10:17:23 -0700
Assunto: [PATCH 1/2] Adicionar limite à função de registro


Limite a funcionalidade de registro aos primeiros 20
```

Este é o início da saída do comando git format-patch que você viu na seção anterior; também representa um formato de e-mail mbox válido. Se alguém lhe enviou o patch por e-mail usando o git send-email e você baixou isso em um formato mbox, então você pode apontar o git am para esse arquivo mbox e ele começará a aplicar todos os patches que encontrar. Se você executar um cliente de e-mail que pode salvar vários e-mails no formato mbox, poderá salvar toda a série de patches em um arquivo e, em seguida, usar o git am para aplicá-los um de cada vez.

No entanto, se alguém carregou um arquivo de patch gerado via git format-patch para um sistema de tickets ou algo semelhante, você pode salvar o arquivo localmente e passar esse arquivo salvo em seu disco para git am para aplicá-lo:

```
$ git am 0001-limit-log-function.patch Aplicando:
Adicionar limite à função de log
```

Você pode ver que ele aplicou de forma limpa e criou automaticamente o novo commit para você. As informações do autor são obtidas dos cabeçalhos De e Data do e-mail , e a mensagem do commit é obtida do Assunto e do corpo (antes do patch) do e-mail. Por exemplo, se este patch foi aplicado a partir do exemplo mbox acima, o commit gerado seria mais ou menos assim:

Machine Translated by Google

```
$ git log --pretty=fuller -1 commit
6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Autor: Jessica Smith <jessica@example.com> Data do
autor: Dom 6 de abril 10:17:23 2008 -0700 Commit: Scott
Chacon <schacon@gmail.com> CommitDate: Qui 9 de
abril 09:19:06 2009 -0700


Adicionar limite à função de log

Limite a funcionalidade de registro aos primeiros 20
```

As informações do Commit indicam a pessoa que aplicou o patch e a hora em que foi aplicado. As informações do autor são o indivíduo que criou originalmente o patch e quando ele foi originalmente criado.

Mas é possível que o patch não seja aplicado corretamente. Talvez seu branch principal tenha divergido muito do branch a partir do qual o patch foi criado, ou o patch dependa de outro patch que você ainda não aplicou. Nesse caso, o processo git am falhará e perguntará o que você deseja fazer:

```
$ git am 0001-veja-se-isso-ajuda-a-gem.patch
Aplicando: Veja se isso ajuda a gem erro: patch falhou:
ticgit.gemspec:1 erro: ticgit.gemspec: patch não se
aplica Patch falhou em 0001 .

Depois de resolver esse problema, execute "git am --resolved".
Se você preferir pular este patch, execute "git am --skip".
Para restaurar a ramificação original e interromper o patch, execute "git am --abort".
```

Este comando coloca marcadores de conflito em todos os arquivos com os quais ele tem problemas, muito parecido com uma operação de mesclagem ou rebase em conflito. Você resolve esse problema da mesma maneira - edite o arquivo para resolver o conflito, teste o novo arquivo e execute git am --resolved para continuar no próximo patch:

```
$ (conserte o
arquivo) $ git add
ticgit.gemspec $ git am --
resolved Aplicando: Veja se isso ajuda a gem
```

Se você quiser que o Git tente resolver o conflito de forma um pouco mais inteligente, você pode passar uma opção -3 para ele, o que faz com que o Git tente uma mesclagem de três vias. Esta opção não está ativada por padrão porque não funciona se o commit no qual o patch diz que foi baseado não estiver em seu repositório. Se você tiver esse commit — se o patch foi baseado em um commit público — então a opção -3 é geralmente muito mais inteligente ao aplicar um patch conflitante:

```
$ git am -3 0001-veja-se-isso-ajuda-a-gem.patch Aplicando:
Veja se isso ajuda o erro da gem: falha no patch:
ticgit.gemspec:1
```

Machine Translated by Google

```
erro: ticgit.gemspec: patch não se aplica Usando
informações de índice para reconstruir uma árvore base...
Voltando ao patch base e 3-way merge...
Sem alterações -- Patch já aplicado.
```

Nesse caso, sem a opção -3 , o patch seria considerado um conflito. Como a opção -3 foi usada, o patch foi aplicado de forma limpa.

Se você estiver aplicando vários patches de uma mbox, também poderá executar o comando am no modo interativo, que para a cada patch encontrado e pergunta se você deseja aplicá-lo:

```
$ git am -3 -i mbox
Commit Body é:
--------------------------
Veja se isso ajuda a gema
--------------------------
Aplicar? [y]es/[n]o/[e]dit/[v]view patch/[a]ccept all
```

Isso é bom se você tiver vários patches salvos, porque você pode ver o patch primeiro se não lembrar o que é, ou não aplicar o patch se já o tiver feito.

Quando todos os patches para o seu tópico forem aplicados e confirmados em sua ramificação, você poderá escolher se e como integrá-los em uma ramificação de execução mais longa.

Check-out de filiais remotas

Se sua contribuição veio de um usuário do Git que configurou seu próprio repositório, enviou várias alterações para ele e, em seguida, enviou a URL para o repositório e o nome da ramificação remota em que as alterações estão, você pode adicioná-las como um remoto e fazer fusões localmente.

Por exemplo, se Jessica enviar um e-mail para você dizendo que ela tem um ótimo novo recurso no branch ruby-client de seu repositório, você pode testá-lo adicionando o controle remoto e verificando esse branch localmente:

```
$ git remote add jessica git://github.com/jessica/myproject.git $ git
buscar jessica $ git checkout -b rubyclient jessica/ruby-client
```

Se ela enviar um e-mail para você novamente mais tarde com outra ramificação contendo outro ótimo recurso, você poderá buscar e finalizar a compra diretamente porque já possui a configuração remota.

Isso é mais útil se você estiver trabalhando com uma pessoa de forma consistente. Se alguém tiver apenas um único patch para contribuir de vez em quando, aceitá-lo por e-mail pode consumir menos tempo do que exigir que todos executem seu próprio servidor e ter que adicionar e remover continuamente controles remotos para obter alguns patches. Também é improvável que você queira ter centenas de controles remotos, cada um para alguém que contribui com apenas um patch ou dois. No entanto, scripts e serviços hospedados podem facilitar isso — depende muito de como você desenvolve e de como seus colaboradores se desenvolvem.

Machine Translated by Google

A outra vantagem dessa abordagem é que você também obtém o histórico dos commits. Embora você possa ter problemas legítimos de mesclagem, você sabe em que parte do seu histórico o trabalho deles se baseia; uma mesclagem de três vias adequada é o padrão, em vez de ter que fornecer um -3 e esperar que o patch tenha sido gerado a partir de um commit público ao qual você tem acesso.

Se você não está trabalhando com uma pessoa de forma consistente, mas ainda deseja extrair dela dessa maneira, pode fornecer a URL do repositório remoto para o comando git pull . Isso faz uma extração única e não salva a URL como uma referência remota:

```
$ git pull https://github.com/onetimeguy/project From
https://github.com/onetimeguy/project * branch HEAD ->
FETCH_HEAD Mesclar feito pela estratégia 'recursiva'.
```

Determinando o que é introduzido

Agora você tem uma ramificação de tópico que contém o trabalho contribuído. Neste ponto, você pode determinar o que gostaria de fazer com ele. Esta seção revisita alguns comandos para que você possa ver como usá-los para revisar exatamente o que introduzirá se mesclar isso em sua ramificação principal.

Muitas vezes é útil obter uma revisão de todos os commits que estão neste branch, mas que não estão em seu branch master . Você pode excluir commits no branch master adicionando a opção --not antes do nome do branch. Isso faz a mesma coisa que o formato master..contrib que usamos anteriormente. Por exemplo, se o seu colaborador enviar dois patches e você criar uma ramificação chamada contrib e aplicar esses patches lá, poderá executar isto:

```
$ git log contrib --not master commit
5b6235bd297351589efc4d73316f0a68d484f118 Autor:
Scott Chacon <schacon@gmail.com> Data: Sex, 24 de
outubro, 09:53:59 2008 -0700

Veja se isso ajuda a gema

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Autor: Scott Chacon <schacon@gmail.com> Data: Seg
22 de outubro 19:38:36 2008 -0700

Atualize o gemspec para funcionar melhor
```

Para ver quais alterações cada commit introduz, lembre-se de que você pode passar a opção -p para o git log e ele anexará o diff introduzido a cada commit.

Para ver uma comparação completa do que aconteceria se você mesclasse esse branch de tópico com outro branch, talvez seja necessário usar um truque estranho para obter os resultados corretos. Você pode pensar em executar isto:

```
$ git diff mestre
```

Machine Translated by Google

Este comando fornece uma diferença, mas pode ser enganoso. Se seu branch master avançou desde que você criou o branch de tópico a partir dele, você obterá resultados aparentemente estranhos. Isso acontece porque o Git compara diretamente os snapshots do último commit do branch do tópico em que você está e o snapshot do último commit do branch master . Por exemplo, se você adicionou uma linha em um arquivo na ramificação master , uma comparação direta dos instantâneos parecerá que a ramificação do tópico removerá essa linha.

Se master for um ancestral direto de sua ramificação de tópico, isso não será um problema; mas se as duas histórias divergirem, o diff parecerá que você está adicionando todas as coisas novas em seu branch de tópico e removendo tudo que é exclusivo do branch master .

O que você realmente deseja ver são as alterações adicionadas ao branch de tópico — o trabalho que você introduzirá se mesclar esse branch com master. Você faz isso fazendo com que o Git compare o último commit em seu branch de tópico com o primeiro ancestral comum que ele tem com o branch master .

Tecnicamente, você pode fazer isso descobrindo explicitamente o ancestral comum e, em seguida, executando seu diff nele:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

ou, mais concisamente:

```
$ git diff $(git merge-base contrib master)
```

No entanto, nenhum deles é particularmente conveniente, então o Git fornece outra abreviação para fazer a mesma coisa: a sintaxe de ponto triplo. No contexto do comando git diff , você pode colocar três pontos após outro branch para fazer uma diferença entre o último commit do branch em que você está e seu ancestral comum com outro branch:

```
$ git diff master...contrib
```

Este comando mostra apenas o trabalho que sua ramificação de tópico atual introduziu desde seu ancestral comum com mestre. Essa é uma sintaxe muito útil para lembrar.

Integrando trabalho de contribuição

Quando todo o trabalho em sua ramificação de tópico estiver pronto para ser integrado a uma ramificação mais principal, a questão é como fazê-lo. Além disso, qual fluxo de trabalho geral você deseja usar para manter seu projeto? Você tem várias opções, por isso abordaremos algumas delas.

Mesclando Fluxos de Trabalho

Um fluxo de trabalho básico é simplesmente mesclar todo esse trabalho diretamente em sua ramificação principal . Nesse cenário, você tem uma ramificação mestre que contém código basicamente estável. Quando você tem trabalho em um

Machine Translated by Google

ramificação de tópico que você acha que concluiu ou trabalho que outra pessoa contribuiu e você verificou, você a mescla em sua ramificação mestre, exclui a ramificação de tópico recém-mesclada e repita.

Por exemplo, se tivermos um repositório com trabalho em duas ramificações chamadas ruby_client e php_client que se parece com History com várias ramificações de tópico, e mesclarmos ruby_client seguido por php_client, seu histórico ficará parecido com After a topic branch merge.

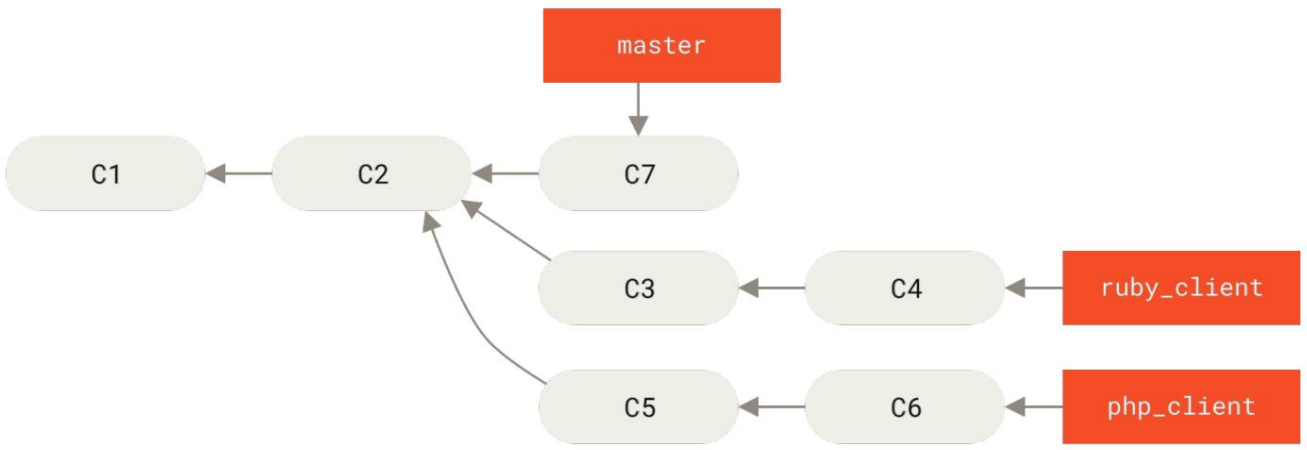


Figura 72. Histórico com várias ramificações de tópicos

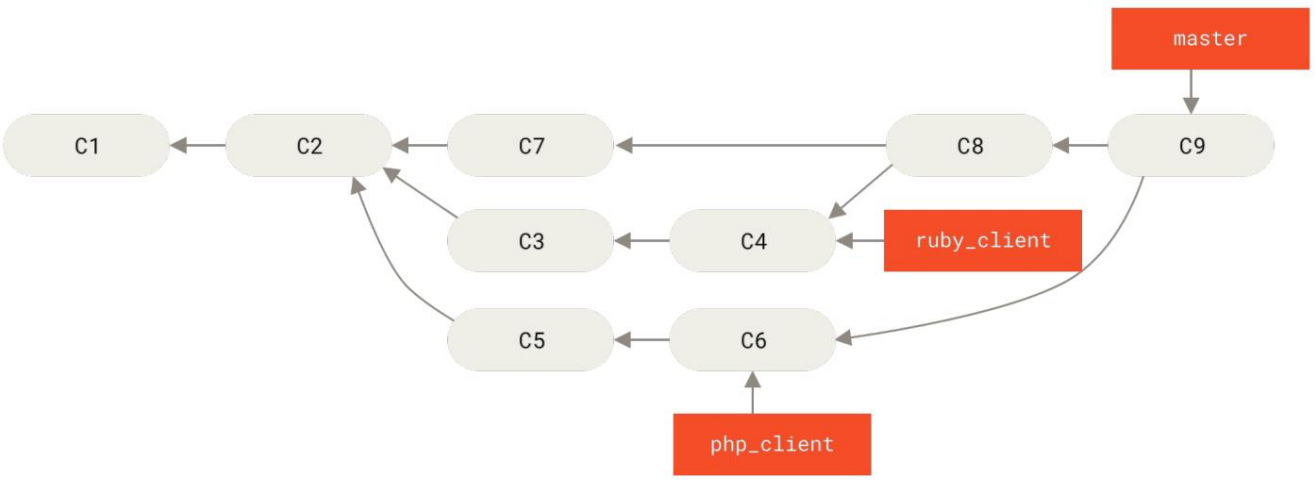


Figura 73. Após uma mesclagem de ramificação de tópico

Esse é provavelmente o fluxo de trabalho mais simples, mas pode ser problemático se você estiver lidando com projetos maiores ou mais estáveis, nos quais deseja ter muito cuidado com o que apresenta.

Se você tiver um projeto mais importante, talvez queira usar um ciclo de mesclagem de duas fases. Neste cenário, você tem duas ramificações de execução longa, master e develop, nas quais você determina que a master é atualizada somente quando uma versão muito estável é cortada e todo o novo código é integrado na ramificação de desenvolvimento . Você envia regularmente essas duas ramificações para o repositório público. Cada vez que você tem uma nova ramificação de tópico para mesclar (antes de uma mesclagem de ramificação de tópico), você a mescla em desenvolvimento (após uma mesclagem de ramificação de tópico); então, quando você marca uma versão, você avança rapidamente para onde quer que a ramificação de desenvolvimento agora estável esteja (após uma versão do projeto).

Machine Translated by Google

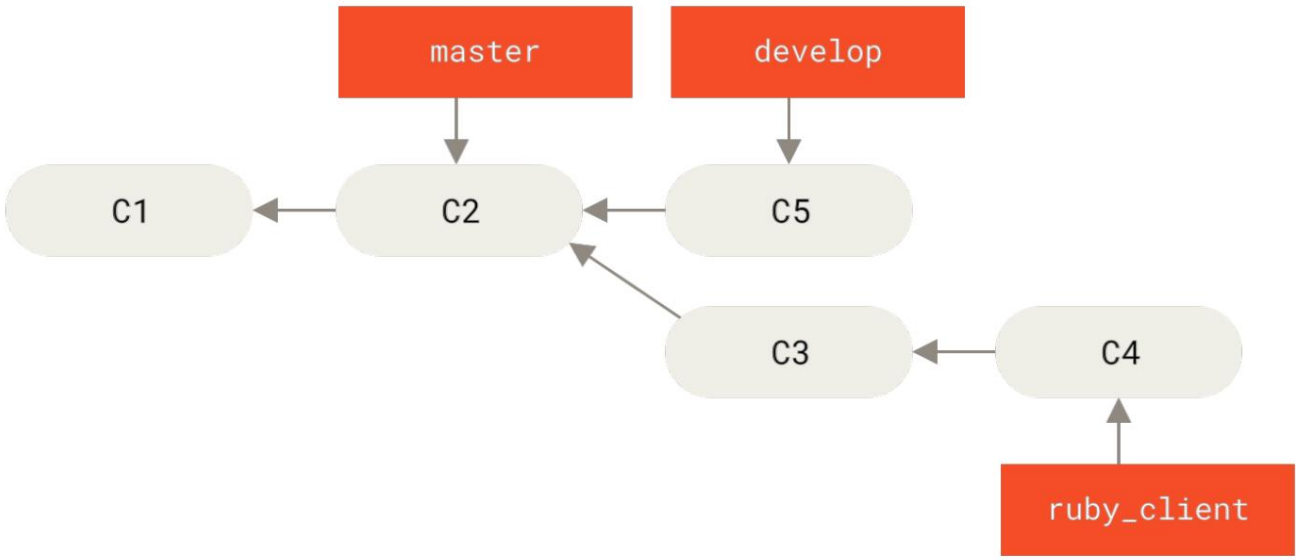


Figura 74. Antes de uma mesclagem de ramificação de tópico

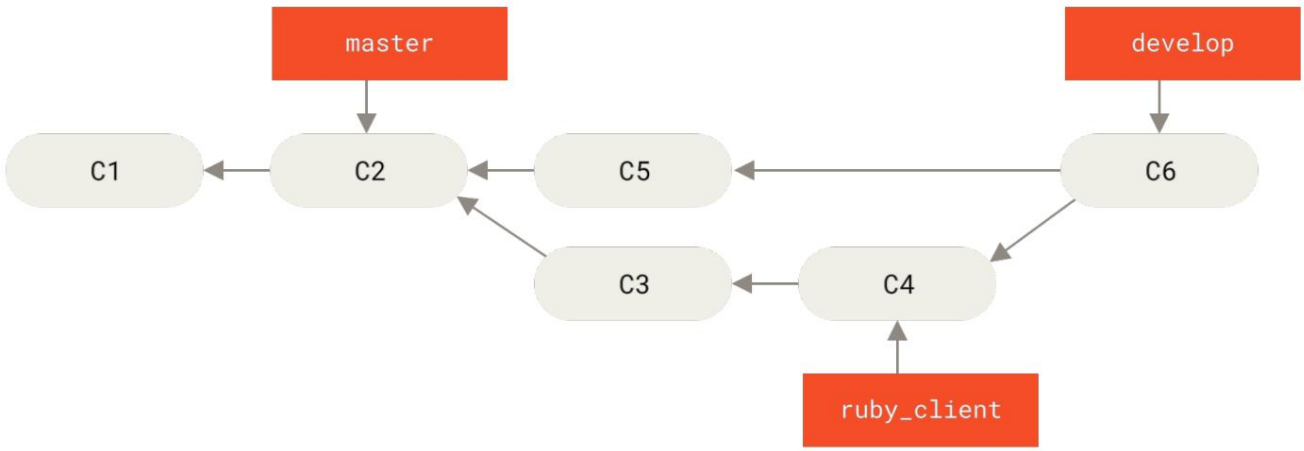


Figura 75. Após uma mesclagem de ramificação de tópico

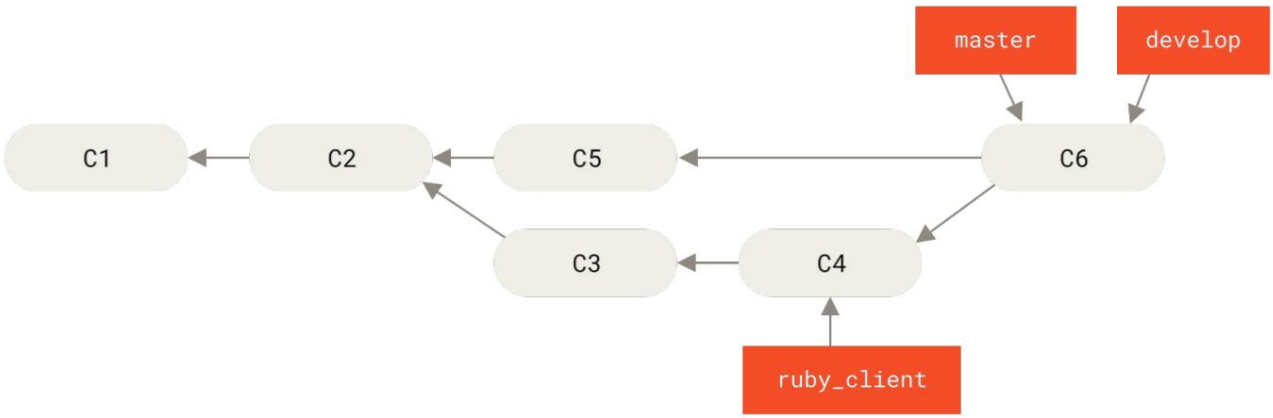


Figura 76. Após o lançamento de um projeto

Dessa forma, quando as pessoas clonarem o repositório do seu projeto, elas podem verificar o master para compilar a versão estável mais recente e manter-se atualizada facilmente, ou podem verificar o develop, que é o conteúdo mais avançado. Você também pode estender esse conceito tendo uma ramificação de integração onde todo o trabalho é mesclado. Então, quando a base de código nesse ramo estiver estável e passar nos testes, você o mesclará em um ramo de desenvolvimento ; e quando isso se mostrou estável por um tempo, você jejua

Machine Translated by Google

encaminhe seu branch master .

Grandes Fluxos de Trabalho de Mesclagem

O projeto Git tem quatro ramificações de execução longa: master, next e seen (anteriormente 'pu' — atualizações propostas) para novos trabalhos e maint para backports de manutenção. Quando um novo trabalho é introduzido por contribuidores, ele é coletado em ramificações de tópico no repositório do mantenedor de maneira semelhante à que descrevemos (consulte Gerenciando uma série complexa de ramificações de tópico contribuídas paralelamente). Nesse ponto, os tópicos são avaliados para determinar se estão seguros e prontos para consumo ou se precisam de mais trabalho. Se estiverem seguros, eles serão mesclados no próximo, e essa ramificação será empurrada para que todos possam experimentar os tópicos integrados juntos.

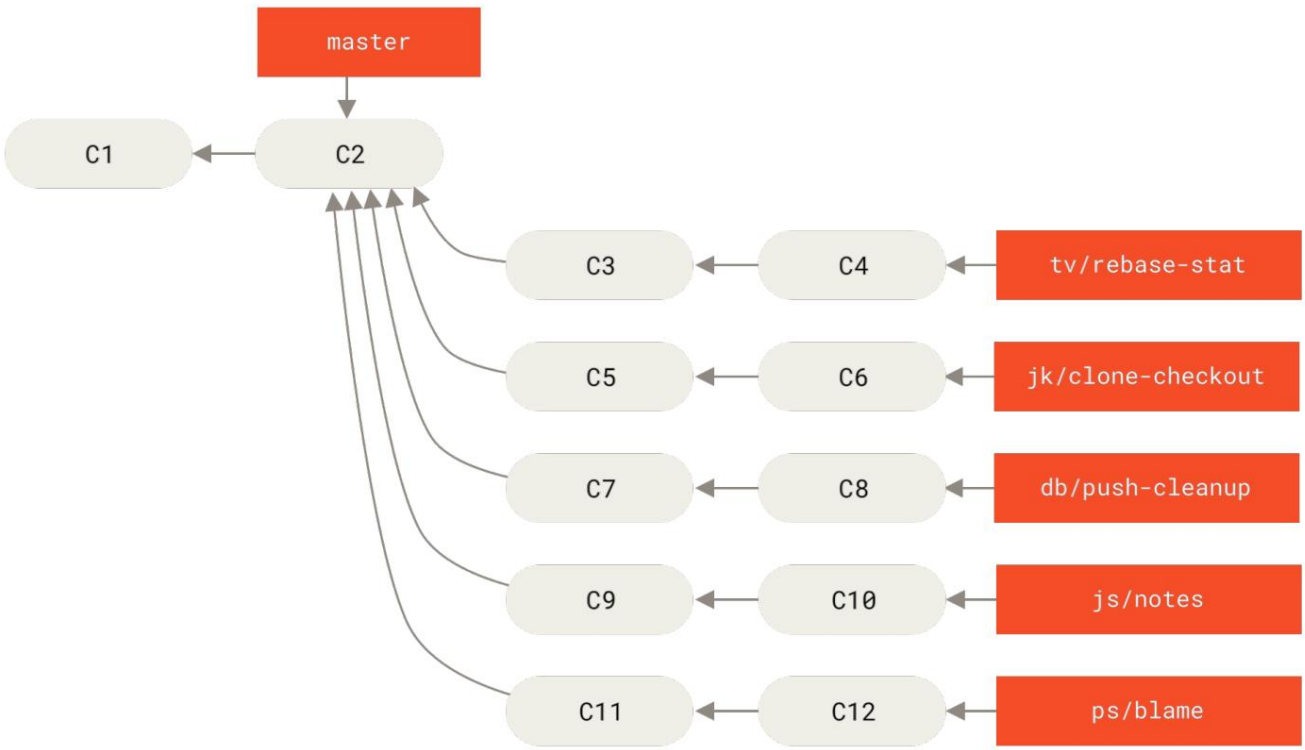


Figura 77. Gerenciando uma série complexa de ramificações paralelas de tópicos contribuídos

Se os tópicos ainda precisarem de trabalho, eles serão mesclados em vistos . Quando é determinado que estão totalmente estáveis, os tópicos são mesclados novamente no mestre. Os galhos próximos e vistos são reconstruídos a partir do mestre. Isso significa que o mestre quase sempre avança, o próximo é rebaseado ocasionalmente e o visto é rebaseado ainda mais frequentemente:

Machine Translated by Google

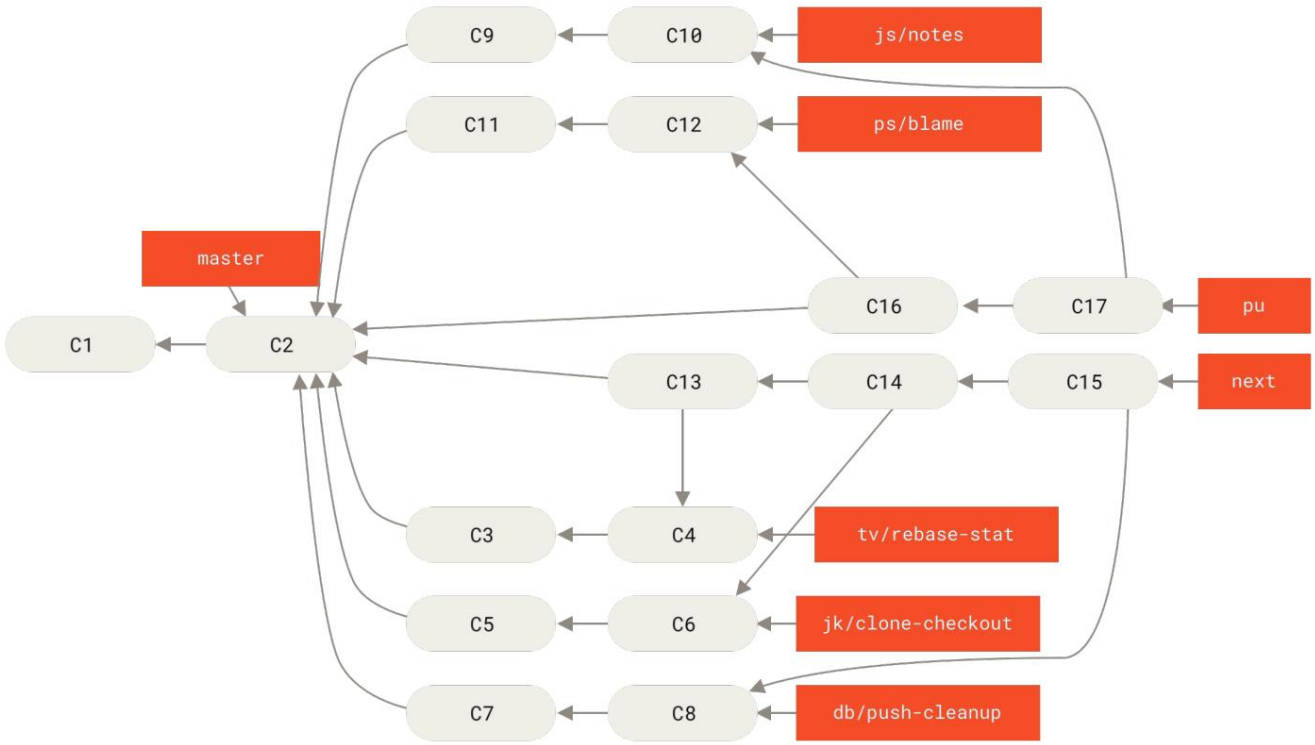


Figura 78. Mesclando ramificações de tópicos contribuídos em ramificações de integração de longo prazo

Quando um branch de tópico finalmente foi mesclado no master, ele é removido do repositório. O projeto Git também tem um branch maint que é derivado da última versão para fornecer patches de backport caso uma versão de manutenção seja necessária. Assim, ao clonar o repositório Git, você tem quatro branches que você pode conferir para avaliar o projeto em diferentes estágios de desenvolvimento, dependendo do quão vanguardista você quer ser ou como quer contribuir; e o mantenedor tem um fluxo de trabalho estruturado para ajudá-lo a avaliar novas contribuições. O fluxo de trabalho do projeto Git é especializado. Para entender isso claramente, você pode verificar o guia do Git Maintainer.

Fluxos de trabalho de rebase e seleção seletiva

Outros mantenedores preferem rebasear ou escolher a dedo o trabalho contribuído no topo de seu branch master , em vez de mesclá-lo, para manter um histórico principalmente linear. Quando você trabalha em uma ramificação de tópico e determina que deseja integrá-la, você move para essa ramificação e executa o comando rebase para reconstruir as alterações na parte superior de sua ramificação principal atual (ou desenvolver e assim por diante) . Se isso funcionar bem, você pode avançar sua ramificação principal e obterá um histórico de projeto linear.

A outra maneira de mover o trabalho introduzido de um ramo para outro é selecioná-lo a dedo. Uma escolha de cereja no Git é como um rebase para um único commit. Ele pega o patch que foi introduzido em um commit e tenta reaplicá-lo no branch em que você está. Isso é útil se você tiver vários commits em um branch de tópico e quiser integrar apenas um deles, ou se tiver apenas um commit em um branch de tópico e preferir selecioná-lo a dedo em vez de executar o rebase. Por exemplo, suponha que você tenha um projeto semelhante a este:

Machine Translated by Google

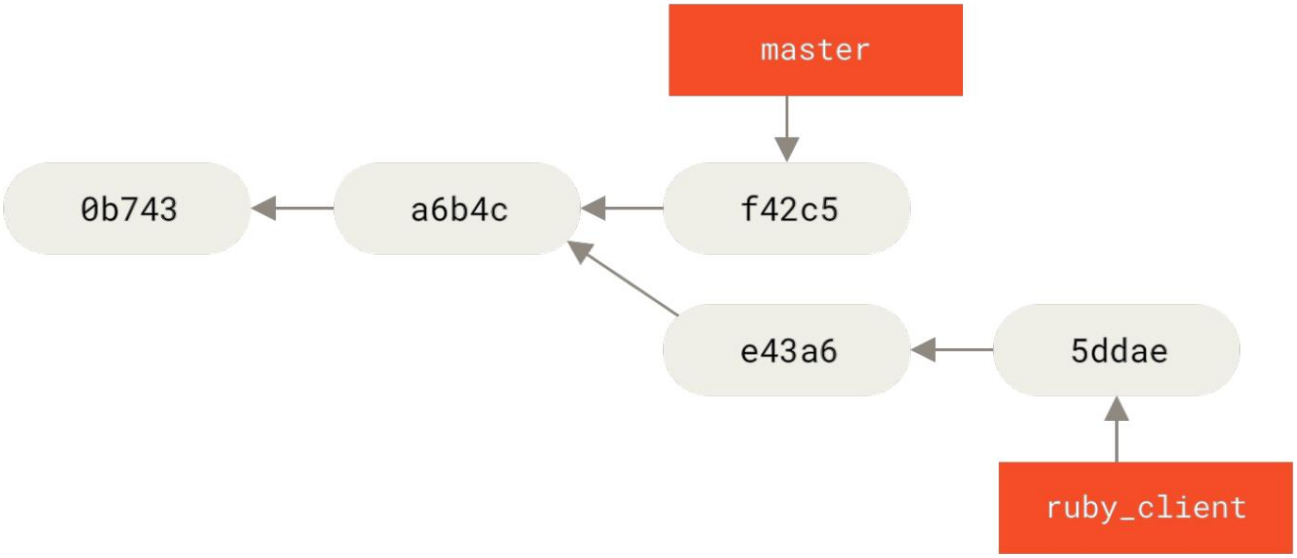


Figura 79. Exemplo de histórico antes de um cherry-pick

Se você deseja puxar o commit e43a6 para sua ramificação master , você pode executar:

```
$ git cherry-pick e43a6
Terminado um cherry-pick.
[mestre]: criado a0a41a9: "Mensagem mais amigável ao bloquear o índice falha." 3
arquivos alterados, 17 inserções (+), 3 exclusões (-)
```

Isso extrai a mesma alteração introduzida em e43a6, mas você obtém um novo valor de commit SHA-1, porque a data aplicada é diferente. Agora seu histórico está assim:

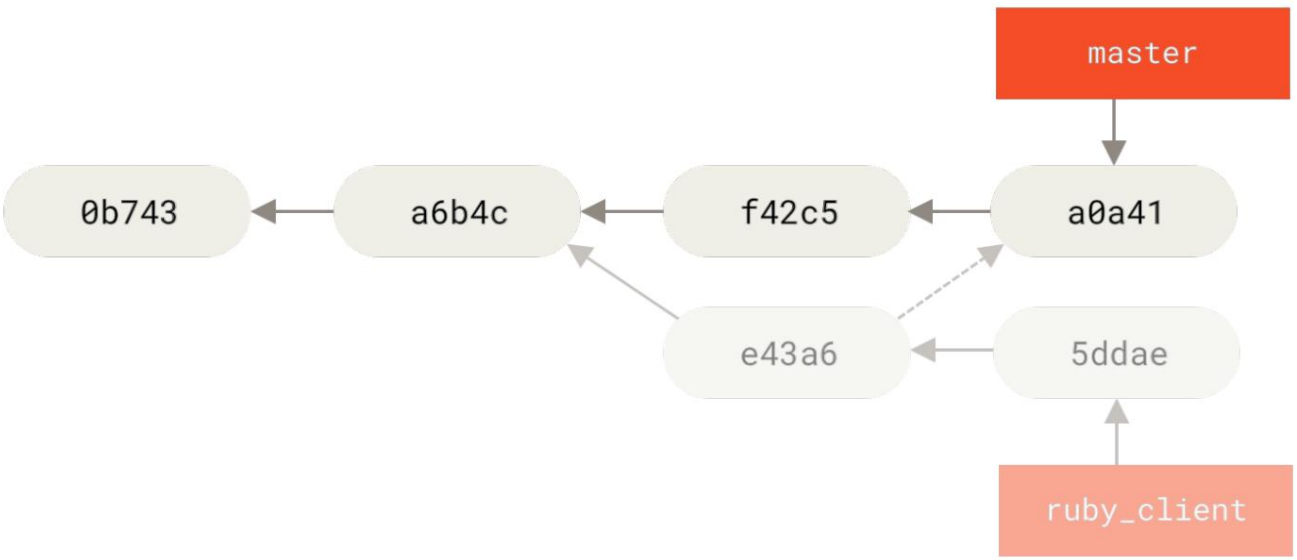


Figura 80. Histórico após selecionar um commit em um branch de tópico

Agora você pode remover sua ramificação de tópico e descartar os commits que não deseja obter.

Machine Translated by Google

Voo

Se você estiver fazendo muitos merges e rebases, ou estiver mantendo um branch de tópico de longa duração, o Git tem um recurso chamado “rerere” que pode ajudar.

Rerere significa “reutilizar resolução gravada” — é uma forma de abreviar a resolução manual de conflitos. Quando o rerere estiver ativado, o Git manterá um conjunto de pré e pós-imagens de mesclagens bem-sucedidas e, se perceber que há um conflito que se parece exatamente com um que você já corrigiu, ele usará a correção da última vez , sem incomodá-lo com isso.

Esse recurso vem em duas partes: uma definição de configuração e um comando. A definição de configuração é rerere.enabled e é útil o suficiente para colocar em sua configuração global:

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

Agora, sempre que você fizer uma mesclagem que resolva conflitos, a resolução ficará registrada no cache caso você precise no futuro.

Se precisar, você pode interagir com o cache rerere usando o comando git rerere . Quando é invocado sozinho, o Git verifica seu banco de dados de resoluções e tenta encontrar uma correspondência com quaisquer conflitos de mesclagem atuais e resolvê-los (embora isso seja feito automaticamente se rerere.enabled for definido como true). Também existem subcomandos para ver o que será gravado, para apagar uma resolução específica do cache e para limpar todo o cache. Abordaremos rerere com mais detalhes em Rerere.

Marcando seus lançamentos

Quando você decidir cortar um lançamento, provavelmente desejará atribuir uma tag para poder recriar esse lançamento a qualquer momento no futuro. Você pode criar uma nova tag conforme discutido em Git Basics. Se você decidir assinar a tag como o mantenedor, a marcação pode ser algo como isto:

```
$ git tag -s v1.5 -m 'minha tag 1.5 assinada'
Você precisa de uma senha para desbloquear a chave secreta
do usuário: "Scott Chacon <schacon@gmail.com>" Chave DSA
de 1024 bits, ID F721C45A, criada em 2009-02-09
```

Se você assinar suas tags, poderá ter o problema de distribuir a chave PGP pública usada para assinar suas tags. O mantenedor do projeto Git resolveu esse problema incluindo sua chave pública como um blob no repositório e adicionando uma tag que aponta diretamente para esse conteúdo. Para fazer isso, você pode descobrir qual chave deseja executando gpg --list-keys:

```
$ gpg --list-keys /
Users/schacon/.gnupg/pubring.gpg
---------------------------------

pub 1024D/F721C45A 2009-02-09 [expira: 2010-02-09] uid
Scott Chacon <schacon@gmail.com> sub 2048g/45D03282
2009-02-09 [expira: 2010-02-09]
```

Machine Translated by Google

Em seguida, você pode importar a chave diretamente para o banco de dados Git, exportando-a e canalizando-a por meio do git hash-object, que grava um novo blob com esse conteúdo no Git e devolve o SHA-1 do blob:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Agora que você tem o conteúdo de sua chave no Git, pode criar uma tag que aponte diretamente para ela especificando o novo valor SHA-1 que o comando hash-object forneceu a você:

```
$ git tag -a mantenedor-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Se você executar git push --tags, a tag Maintainer-pgp-pub será compartilhada com todos. Se alguém quiser verificar uma tag, pode importar diretamente sua chave PGP puxando o blob diretamente do banco de dados e importando-o para o GPG:

```
$ git show mantenedor-pgp-pub | gpg --import
```

Eles podem usar essa chave para verificar todas as suas tags assinadas. Além disso, se você incluir instruções na mensagem da tag, executar git show <tag> permitirá que você forneça ao usuário final instruções mais específicas sobre a verificação da tag.

Gerando um número de compilação

Como o Git não tem números monotonicamente crescentes como 'v123' ou o equivalente para ir com cada commit, se você quiser ter um nome legível por humanos para ir com um commit, você pode executar git describe nesse commit. Em resposta, o Git gera uma string que consiste no nome da tag mais recente anterior ao commit, seguido pelo número de commits desde aquele tag, seguido finalmente por um valor SHA-1 parcial do commit que está sendo descrito (prefixado com a letra "g" significa Git):

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

Dessa forma, você pode exportar um instantâneo ou criar e nomeá-lo como algo compreensível para as pessoas. Na verdade, se você criar o Git a partir do código-fonte clonado do repositório Git, git --version fornecerá algo parecido com isto. Se você está descrevendo um commit que você marcou diretamente, ele fornece apenas o nome da tag.

Por padrão, o comando git describe requer tags anotadas (tags criadas com o sinalizador -a ou -s); se você também quiser aproveitar as vantagens das tags leves (não anotadas), adicione a opção --tags ao comando. Você também pode usar essa string como alvo de um comando git checkout ou git show , embora ela dependa do valor SHA-1 abreviado no final, portanto, pode não ser válida para sempre. Por exemplo, o kernel do Linux saltou recentemente de 8 para 10 caracteres para garantir a exclusividade do objeto SHA-1, portanto, nomes de saída de descrição git mais antigos foram invalidados.

Machine Translated by Google

Preparando um lançamento

Agora você deseja liberar uma compilação. Uma das coisas que você vai querer fazer é criar um arquivo do último instantâneo do seu código para aquelas pobres almas que não usam o Git. O comando para fazer isso é git archive:

```
$ mestre do arquivo git --prefix='projeto/' | gzip > `git describe master`.tar.gz $ ls *.tar.gz v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Se alguém abrir esse tarball, obterá o instantâneo mais recente do seu projeto em um diretório de projeto .
Você também pode criar um arquivo zip da mesma maneira, mas passando a opção --format=zip para git archive:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Agora você tem um bom tarball e um arquivo zip do lançamento do seu projeto que pode ser carregado no seu site ou enviado por e-mail para as pessoas.

O Shortlog

É hora de enviar um e-mail para sua lista de discussão de pessoas que desejam saber o que está acontecendo em seu projeto. Uma boa maneira de obter rapidamente uma espécie de changelog do que foi adicionado ao seu projeto desde o seu último lançamento ou e-mail é usar o comando git shortlog . Ele resume todos os commits no intervalo que você fornece; por exemplo, o seguinte fornece um resumo de todos os commits desde seu último lançamento, se seu último lançamento foi nomeado v1.0.1:

```
$ git shortlog --no-merges master --não v1.0.1 Chris Wanstrath (6): Adicionar suporte para tags anotadas ao Grit::Tag Adicionar suporte

a tags anotadas com referências empacotadas.



                    Adicionar Grit::Commit#to_patch Atualizar versão e History.txt Remover

                    `puts` perdidos Fazer ls_tree ignorar nils



Tom Preston-Werner (4): corrigir datas no histórico método de

                    versão dinâmica Atualização de versão para 1.0.2 Gemspec

                    regenerado para a versão 1.0.2
```

Você obtém um resumo limpo de todos os commits desde v1.0.1, agrupados por autor, que você pode enviar por e-mail para sua lista.

Machine Translated by Google

Resumo

Você deve se sentir bastante à vontade para contribuir com um projeto no Git, bem como manter seu próprio projeto ou integrar as contribuições de outros usuários. Parabéns por ser um desenvolvedor Git eficaz! No próximo capítulo, você aprenderá como usar o maior e mais popular serviço de hospedagem Git, o GitHub.

Machine Translated by Google

GitHub

GitHub é o maior host único para repositórios Git e é o ponto central de colaboração para milhões de desenvolvedores e projetos. Uma grande porcentagem de todos os repositórios Git são hospedados no GitHub, e muitos projetos de código aberto o usam para hospedagem Git, rastreamento de problemas, revisão de código e outras coisas.
Portanto, embora não seja uma parte direta do projeto de código aberto do Git, há uma boa chance de você querer ou precisar interagir com o GitHub em algum momento ao usar o Git profissionalmente.

Este capítulo é sobre como usar o GitHub de forma eficaz. Abordaremos a inscrição e o gerenciamento de uma conta, a criação e o uso de repositórios Git, fluxos de trabalho comuns para contribuir com projetos e aceitar contribuições para o seu, a interface programática do GitHub e muitas pequenas dicas para facilitar sua vida em geral.

Se você não estiver interessado em usar o GitHub para hospedar seus próprios projetos ou para colaborar com outros projetos hospedados no GitHub, você pode pular com segurança para o Git Tools.

ÿ

Mudança de Interfaces

É importante observar que, como muitos sites ativos, os elementos da interface do usuário nessas capturas de tela podem mudar com o tempo. Esperamos que a ideia geral do que estamos tentando realizar aqui ainda esteja lá, mas se você quiser versões mais atualizadas dessas telas, as versões on-line deste livro podem ter capturas de tela mais recentes.

Configuração e configuração da conta

A primeira coisa que você precisa fazer é configurar uma conta de usuário gratuita. Basta visitar https://github.com, escolha um nome de usuário que ainda não tenha sido usado, forneça um endereço de e-mail e uma senha e clique no grande botão verde “Sign up for GitHub”.

Machine Translated by Google

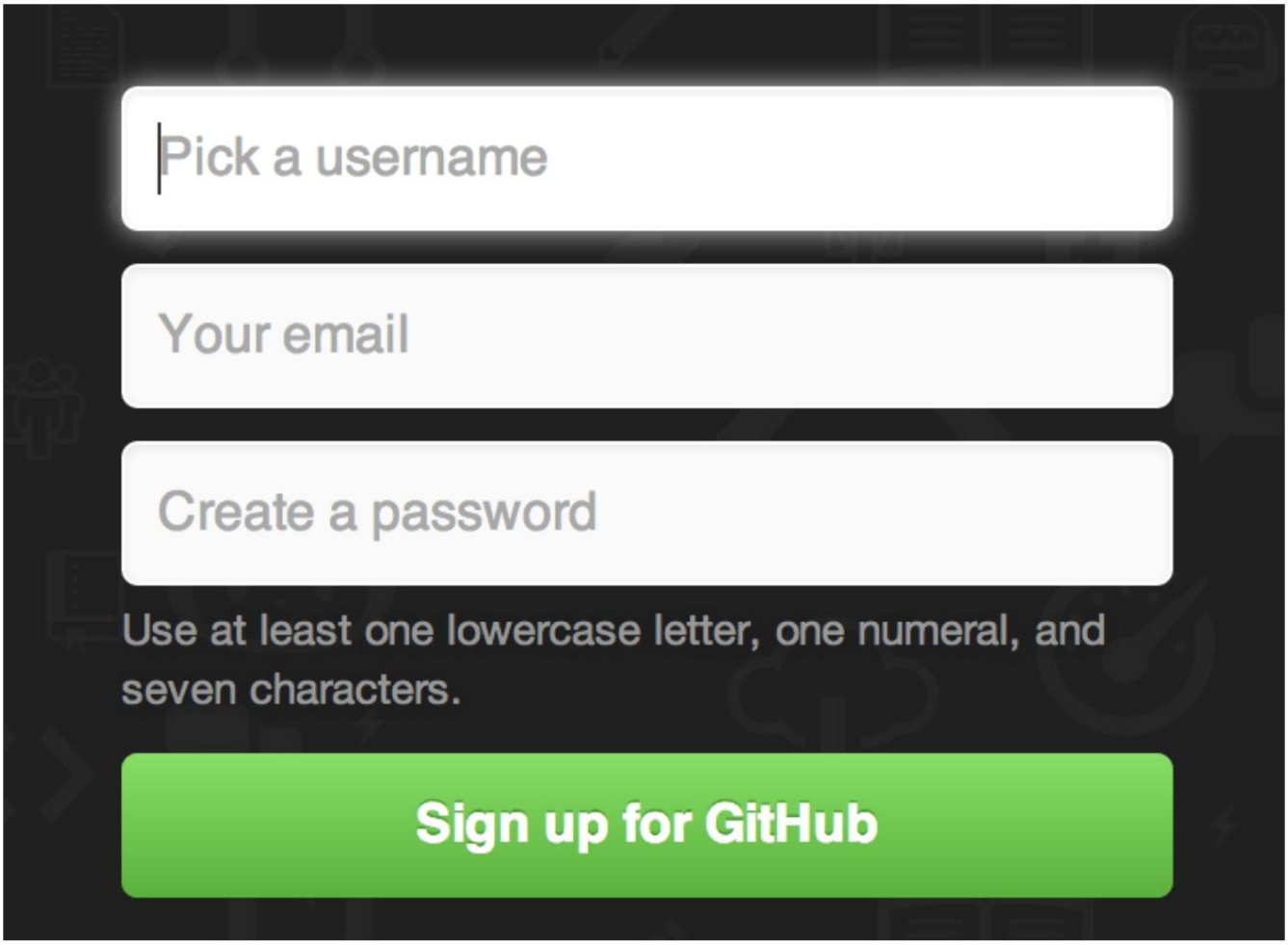


Figura 81. O formulário de inscrição do GitHub

A próxima coisa que você verá é a página de preços dos planos atualizados, mas é seguro ignorá-la por enquanto. O GitHub enviará um e-mail para verificar o endereço fornecido. Vá em frente e faça isso; é muito importante (como veremos mais adiante).

ÿ

O GitHub fornece quase todas as suas funcionalidades com contas gratuitas, exceto alguns recursos avançados.

Os planos pagos do GitHub incluem ferramentas e recursos avançados, bem como limites maiores para serviços gratuitos, mas não os abordaremos neste livro. Para obter mais informações sobre os planos disponíveis e sua comparação, visite https://github.com/pricing .

Clicar no logotipo da Octocat no canto superior esquerdo da tela o levará à página do painel. Agora você está pronto para usar o GitHub.

Acesso SSH

A partir de agora, você pode se conectar totalmente aos repositórios Git usando o protocolo https:// , autenticando com o nome de usuário e a senha que acabou de configurar. No entanto, para simplesmente clonar projetos públicos, você nem precisa se inscrever - a conta que acabamos de criar entra em ação quando bifurcamos projetos e avançamos para nossos bifurcações um pouco mais tarde.

Se você quiser usar controles remotos SSH, precisará configurar uma chave pública. Se você ainda não tem um,

Machine Translated by Google

consulte Gerando sua chave pública SSH. Abra as configurações da sua conta usando o link no canto superior direito da janela:

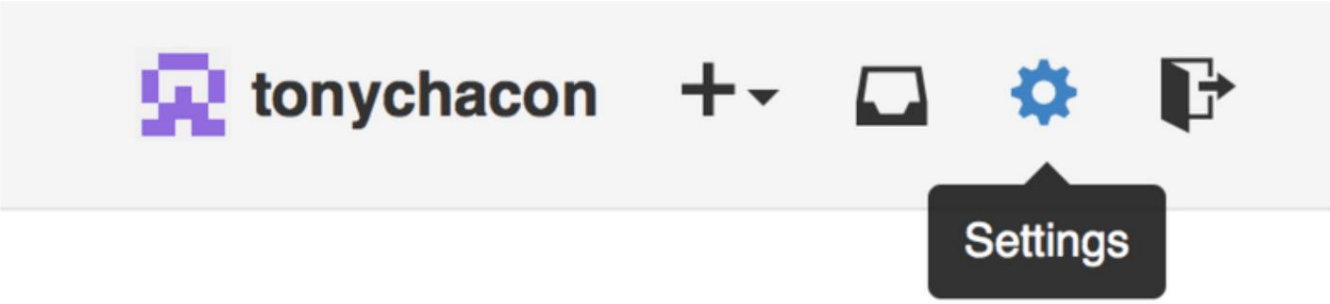


Figura 82. O link "Configurações da conta"

Em seguida, selecione a seção "Chaves SSH" no lado esquerdo.

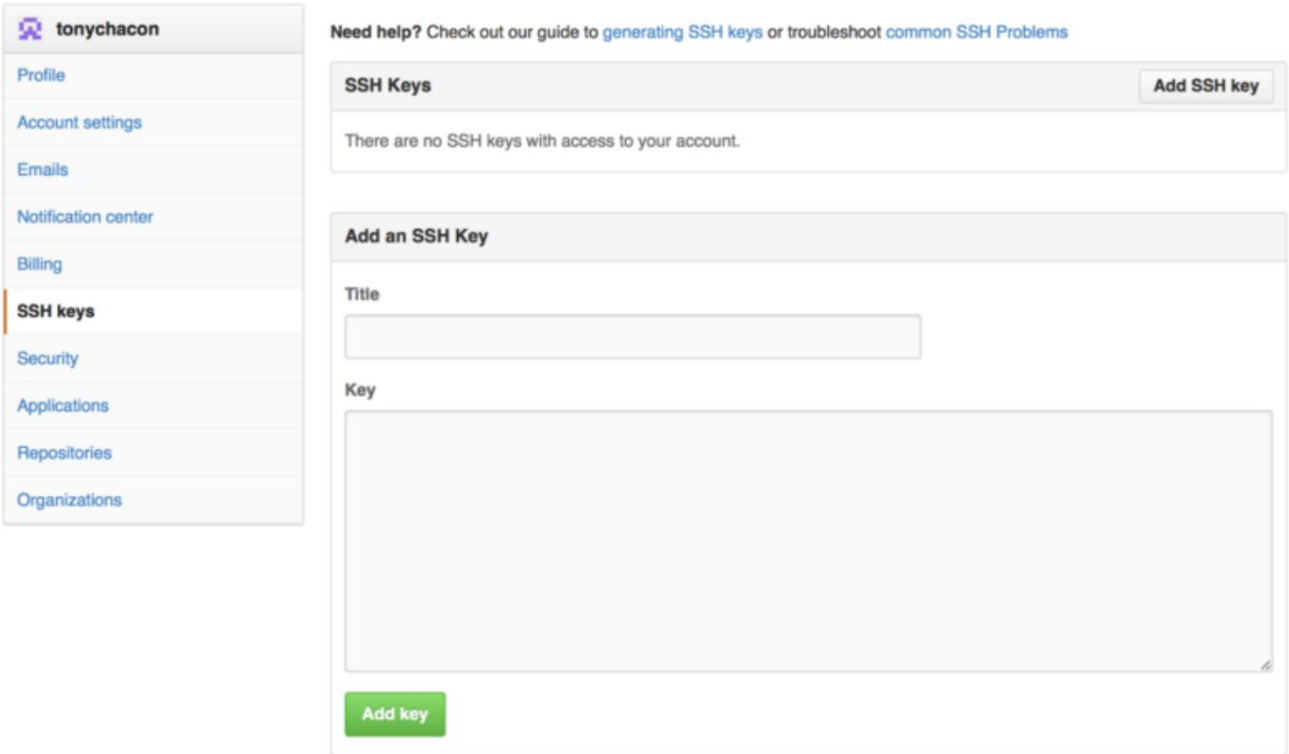


Figura 83. O link "Chaves SSH".

A partir daí, clique no botão "Adicionar uma chave SSH", dê um nome à sua chave, cole o conteúdo do seu arquivo de chave pública ~/.ssh/id_rsa.pub (ou qualquer outro nome) na área de texto e clique em "Adicionar chave".

> ÿ
>
> Certifique-se de nomear sua chave SSH com algo que você possa lembrar. Você pode nomear cada uma de suas chaves (por exemplo, "Meu laptop" ou "Conta de trabalho") para que, se precisar revogar uma chave mais tarde, possa saber facilmente qual delas está procurando.

Seu avatar

Em seguida, se desejar, você pode substituir o avatar gerado para você por uma imagem de sua escolha. Primeiro, vá para a guia "Perfil" (acima da guia Chaves SSH) e clique em "Carregar nova imagem".

Machine Translated by Google

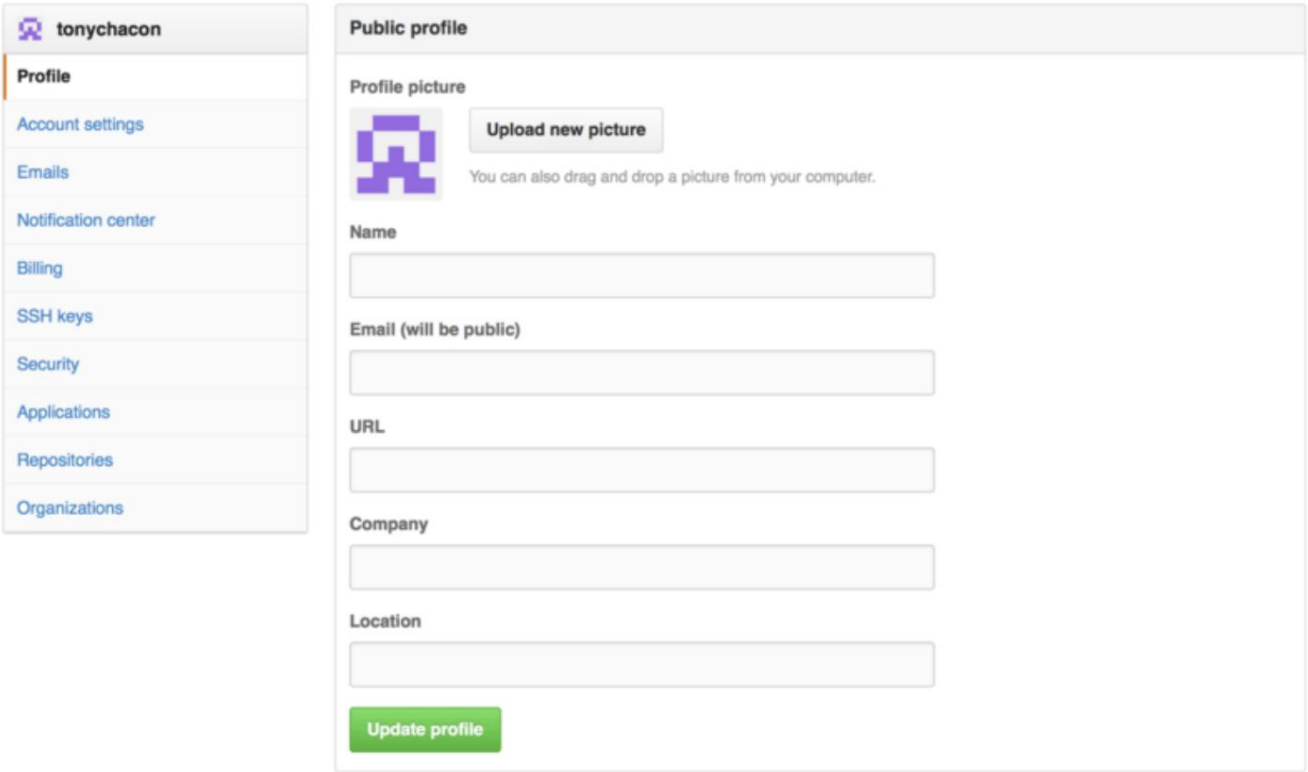


Figura 84. O link "Perfil"

Escolheremos uma cópia do logotipo do Git que está em nosso disco rígido e teremos a chance de cortá-la.

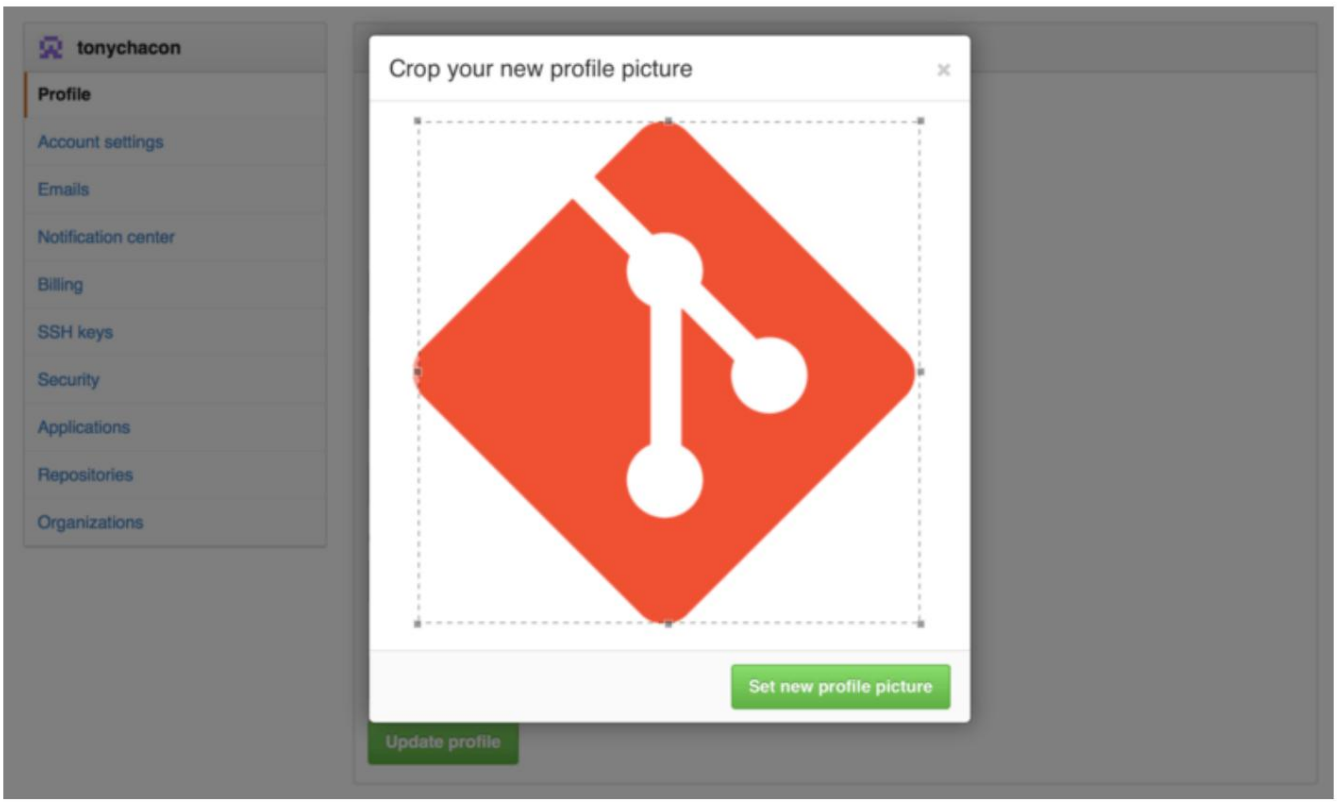


Figura 85. Recorte seu avatar

Agora, em qualquer lugar em que você interagir no site, as pessoas verão seu avatar ao lado do seu nome de usuário.

Se você tiver carregado um avatar para o popular serviço Gravatar (geralmente usado para contas do Wordpress), esse avatar será usado por padrão e você não precisará executar esta etapa.

Machine Translated by Google

Seus endereços de e-mail

A maneira como o GitHub mapeia seus commits do Git para o seu usuário é por endereço de e-mail. Se você usa vários endereços de e-mail em seus commits e deseja que o GitHub os vincule corretamente, você precisa adicionar todos os endereços de e-mail que você usou à seção Emails da seção admin.

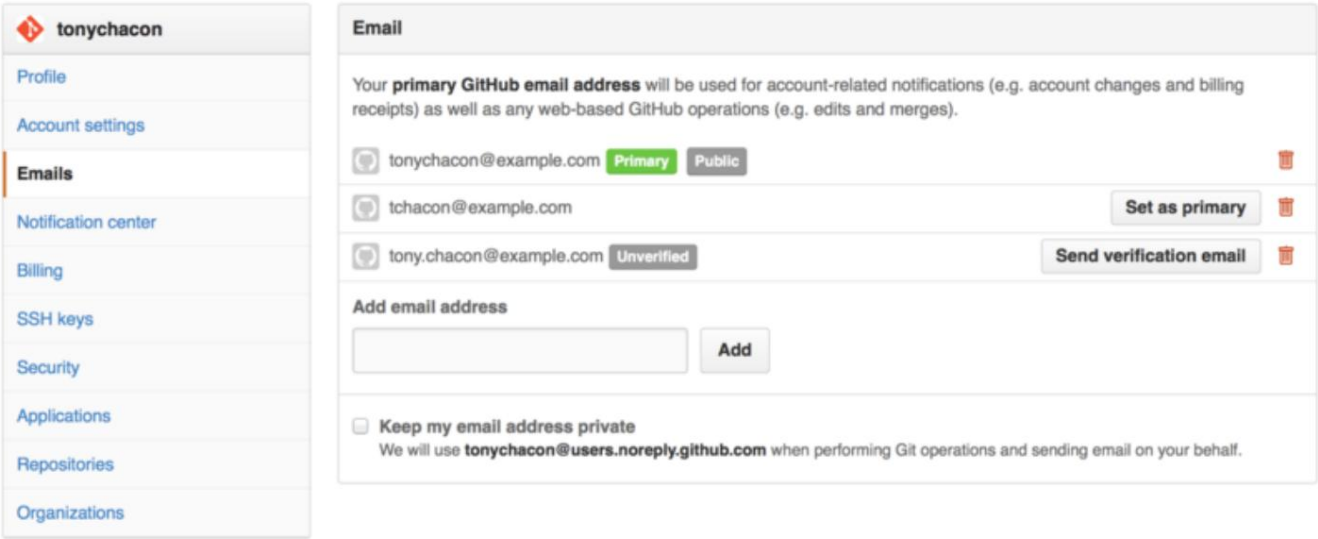


Figura 86. Adicionar endereços de e-mail

Em Adicionar endereços de e-mail, podemos ver alguns dos diferentes estados possíveis. O endereço principal é verificado e definido como o endereço principal, o que significa que é onde você receberá todas as notificações e recibos. O segundo endereço é verificado e, portanto, pode ser definido como o principal se você desejar trocá-los.
O endereço final não foi verificado, o que significa que você não pode torná-lo seu endereço principal. Se o GitHub encontrar qualquer um deles nas mensagens de confirmação em qualquer repositório do site, ele será vinculado ao seu usuário agora.

Autenticação de dois fatores

Finalmente, para segurança extra, você definitivamente deve configurar a autenticação de dois fatores ou “2FA”. Autenticação de dois fatores é um mecanismo de autenticação que está se tornando cada vez mais popular recentemente para mitigar o risco de sua conta ser comprometida se sua senha for roubada de alguma forma. Ativá-lo fará com que o GitHub solicite dois métodos diferentes de autenticação, de modo que, se um deles for comprometido, um invasor não conseguirá acessar sua conta.

Você pode encontrar a configuração da Autenticação de dois fatores na guia Segurança das configurações da sua conta.

Machine Translated by Google

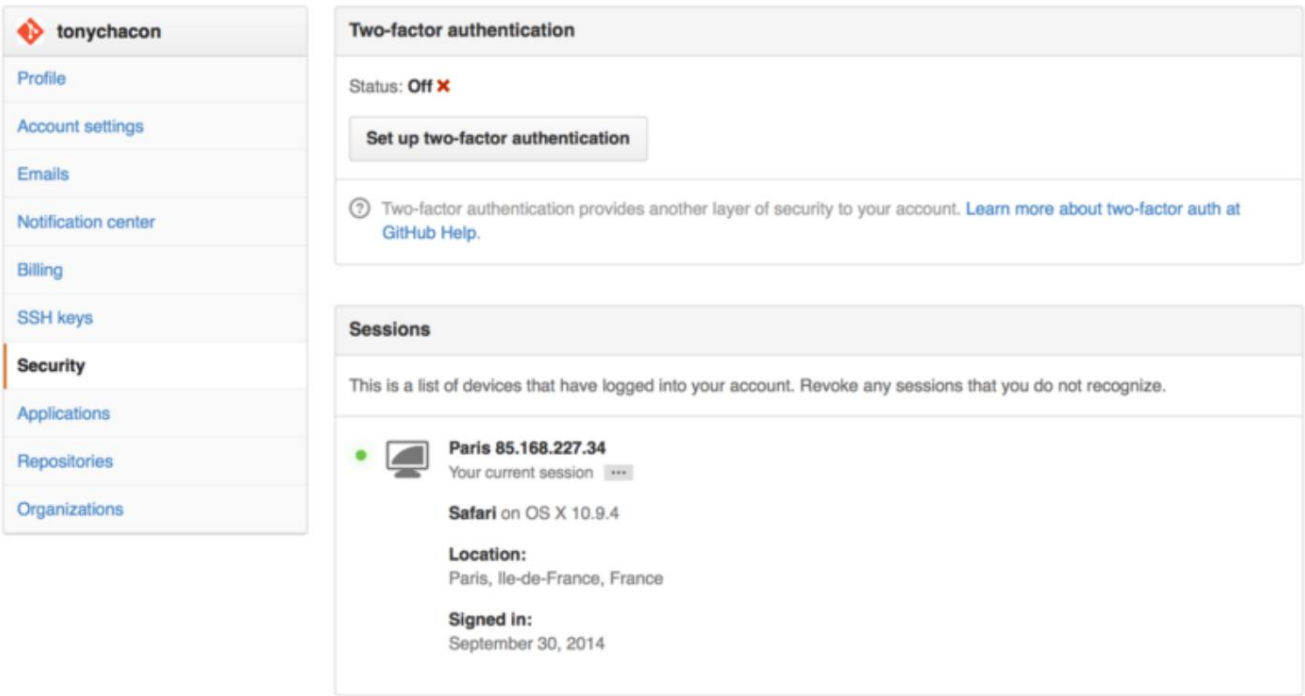


Figura 87. 2FA na guia Segurança

Se você clicar no botão “Configurar autenticação de dois fatores”, será direcionado para uma página de configuração onde você pode optar por usar um aplicativo de telefone para gerar seu código secundário (uma “senha de uso único baseada em tempo”) ou você pode fazer com que o GitHub envie um código via SMS sempre que precisar fazer login.

Depois de escolher o método de sua preferência e seguir as instruções para configurar o 2FA, sua conta ficará um pouco mais segura e você terá que fornecer um código além de sua senha sempre que fizer login no GitHub.

Contribuindo para um Projeto

Agora que nossa conta está configurada, vamos examinar alguns detalhes que podem ser úteis para ajudá-lo a contribuir com um projeto existente.

Projetos de Bifurcação

Se você deseja contribuir para um projeto existente ao qual não tem acesso push, pode “bifurcar” o projeto. Quando você “bifurca” um projeto, o GitHub fará uma cópia do projeto que é inteiramente sua; ele reside em seu namespace e você pode enviar para ele.

> ÿ
>
> Historicamente, o termo “fork” tem sido um pouco negativo no contexto, o que significa que alguém levou um projeto de código aberto em uma direção diferente, às vezes criando um projeto concorrente e dividindo os contribuidores. No GitHub, um “fork” é simplesmente o mesmo projeto em seu próprio namespace, permitindo que você faça alterações em um projeto publicamente como forma de contribuir de maneira mais aberta.

Dessa forma, os projetos não precisam se preocupar em adicionar usuários como colaboradores para dar a eles acesso push. As pessoas podem bifurcar um projeto, enviá-lo por push e contribuir com suas alterações de volta ao repositório original criando o que chamamos de Pull Request, que abordaremos a seguir. Isso abre um tópico de discussão com revisão de código, e o proprietário e o colaborador podem se comunicar

Machine Translated by Google

sobre a alteração até que o proprietário esteja satisfeito com ela, momento em que o proprietário pode integrá-la.

Para bifurcar um projeto, visite a página do projeto e clique no botão “Fork” no canto superior direito da página.

Figura 88. O botão “Fork”

Após alguns segundos, você será direcionado para a página do novo projeto, com sua própria cópia gravável do código.

O fluxo do GitHub

O GitHub foi projetado em torno de um fluxo de trabalho de colaboração específico, centrado em solicitações pull. Esse fluxo funciona se você estiver colaborando com uma equipe unida em um único repositório compartilhado ou uma empresa distribuída globalmente ou uma rede de estranhos contribuindo para um projeto por meio de dezenas de bifurcações. Ele é centrado no fluxo de trabalho Topic Branches abordado no Git Branching.

Veja como geralmente funciona:

1. Bifurque o projeto.
2. Crie uma ramificação de tópico a partir do mestre.
3. Faça alguns commits para melhorar o projeto.
4. Envie esta ramificação para seu projeto GitHub.
5. Abra uma solicitação pull no GitHub.
6. Discuta e, opcionalmente, continue se comprometendo.
7. O proprietário do projeto mescla ou fecha a solicitação pull.
8. Sincronize o master atualizado de volta ao seu fork.

Este é basicamente o fluxo de trabalho do Integration Manager abordado no Integration-Manager Workflow, mas em vez de usar o e-mail para comunicar e revisar as alterações, as equipes usam as ferramentas baseadas na Web do GitHub.

Vamos examinar um exemplo de proposta de alteração em um projeto de software livre hospedado no GitHub usando este fluxo.

ÿ

Você pode usar a ferramenta **CLI oficial do GitHub** em vez da interface da web do GitHub para maioria das coisas. A ferramenta pode ser usada em sistemas Windows, MacOS e Linux. Acesse a página inicial da CLI do GitHub para instruções de instalação e o manual.

Criando uma solicitação pull

Tony está procurando um código para rodar em seu microcontrolador programável Arduino e encontrou um ótimo arquivo de programa no GitHub em https://github.com/schacon/blink.

Machine Translated by Google

schacon / blink

Watch 0 Star 0 Fork 0

branch: master blink / blink.ino

schacon on Jun 12 my arduino blinking code (from arduino.cc)

1 contributor

25 lines (20 sloc) 0.71 kb Raw Blame History

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// 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);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

Figura 89. O projeto com o qual queremos contribuir

O único problema é que a taxa de intermitência é muito rápida. Achamos que é muito melhor esperar 3 segundos em vez de 1 entre cada mudança de estado. Então, vamos melhorar o programa e enviá-lo de volta ao projeto como uma proposta de mudança.

Primeiro, clicamos no botão 'Fork', conforme mencionado anteriormente, para obter nossa própria cópia do projeto. Nosso nome de usuário aqui é “tonychacon” então nossa cópia deste projeto está em https://github.com/tonychacon/blink e é aí que podemos editá-lo. Vamos cloná-lo localmente, criar uma ramificação de tópico, alterar o código e, finalmente, enviar essa alteração de volta ao GitHub.

```
$ git clone https://github.com/tonychacon/blink ÿ
Clonando em 'blink'...

$ cd blink
$ git checkout -b slow-blink ÿ
Mudou para uma nova ramificação 'piscada lenta'

$ sed -i '' 's/1000/3000/' blink.ino (macOS) ÿ # Se
você estiver em um sistema Linux, faça o seguinte:
# $ sed -i 's/1000/3000/' blink. ino ÿ

$ git diff --word-diff ÿ diff --
git a/blink.ino b/blink.ino index
15b9911..a6cc5a5 100644
--- a/blink.ino ++
+ b/blink.ino
```

Machine Translated by Google

```
@@ -18,7 +18,7 @@ void setup() { // a
rotina de loop é executada repetidamente para sempre: void loop()
{ digitalWrite(led, HIGH); // liga o LED (HIGH é o nível de tensão) [-
delay(1000);-]{+delay(3000);+} digitalWrite(led, LOW); [-atraso(1000);-]{+atraso(3000);+} }
                                                                    // espera um segundo
                                                  // desliga o LED tornando a voltagem BAIXA
                                                                    // espera um segundo



$ git commit -a -m 'Alterar atraso para 3 segundos' ÿ [slow-blink
 5ca509d] Alterar atraso para 3 segundos 1 arquivo alterado, 2
inserções (+), 2 exclusões (-)

$ git push origin slow-blink ÿ Nome de
usuário para 'https://github.com': tonychacon Senha para
'https://tonychacon@github.com': Counting objects: 5, done.

Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (3/3), concluído.
Objetos de escrita: 100% (3/3), 340 bytes | 0 byte/s, pronto.
Total 3 (delta 1), reutilizado 0 (delta 0)
Para https://github.com/tonychacon/blink * [novo
ramo]                          piscar lento -> piscar lento
```

ÿ Clone nossa bifurcação do projeto localmente.

ÿ Crie um ramo de tópico descritivo.

ÿ Faça nossa alteração no código.

ÿ Verifique se a alteração está boa.

ÿ Confirme nossa alteração no branch de tópico.

ÿ Envie nosso novo branch de tópico de volta para nosso fork do GitHub.

Agora, se voltarmos ao nosso fork no GitHub, podemos ver que o GitHub notou que enviamos um novo branch de tópico e nos apresenta um grande botão verde para verificar nossas alterações e abrir uma solicitação pull para o projeto original.

Alternativamente, você pode ir para a página “Branches” em https://github.com/<user>/<project>/branches para localizar sua branch e abrir uma nova Pull Request a partir daí.

Machine Translated by Google

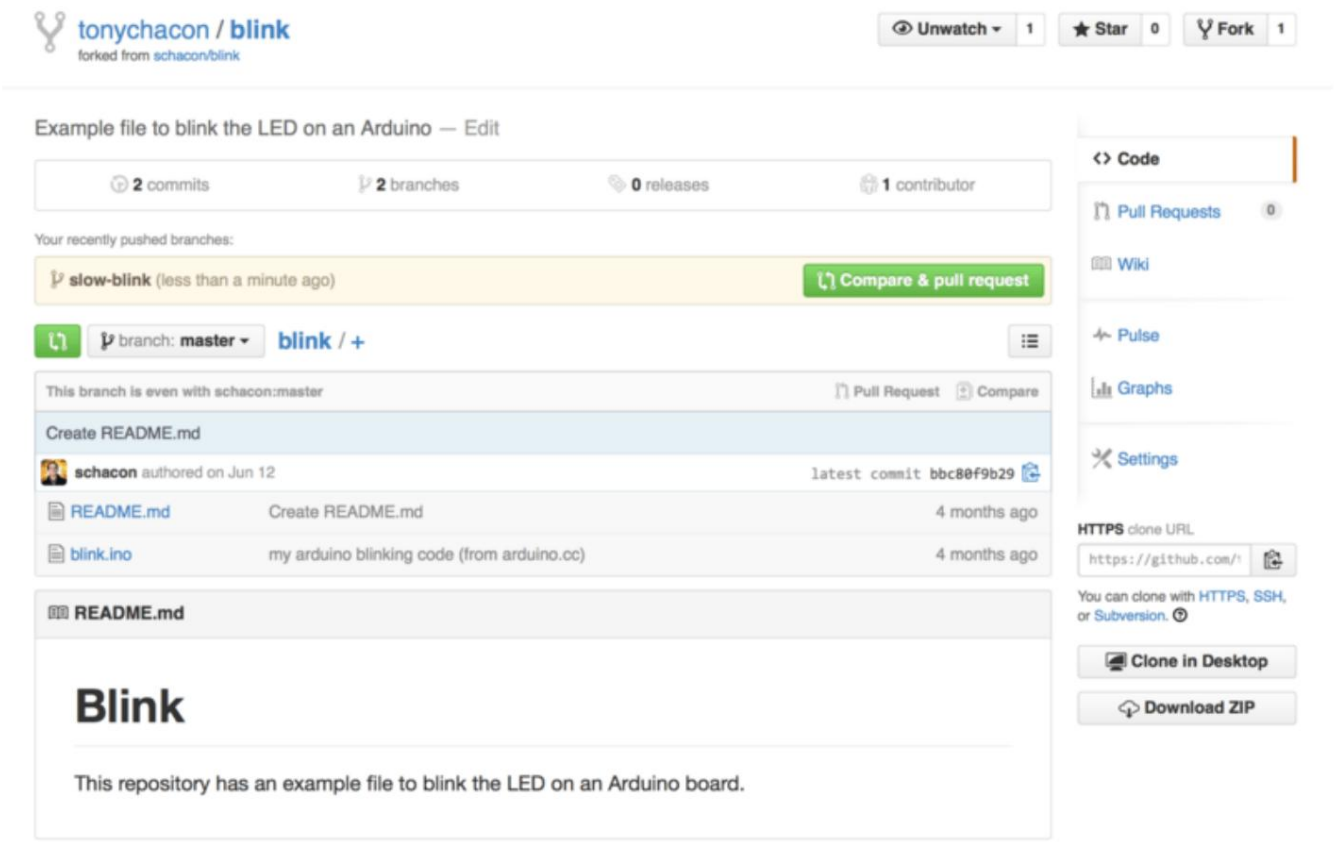


Figura 90. Botão Pull Request

Se clicarmos no botão verde, veremos uma tela que nos pede para dar um título e uma descrição ao nosso Pull Request. Quase sempre vale a pena colocar algum esforço nisso, pois uma boa descrição ajuda o proprietário do projeto original a determinar o que você estava tentando fazer, se as alterações propostas estão corretas e se aceitar as alterações melhoraria o projeto original.

Também vemos uma lista dos commits em nosso branch tópico que estão “à frente” do branch master (neste caso, apenas um) e um diff unificado de todas as alterações que serão feitas caso este branch seja mesclado pelo projeto proprietário.

Machine Translated by Google

tonychacon / blink
forked from schacon/blink

Unwatch 1 | Star 0 | Fork 1

schacon:master ... tonychacon:slow-blink | Edit

Three seconds is better

Write | Preview | Parsed as Markdown | Edit in fullscreen

Studies have shown that 3 seconds is a far better LED delay than 1 second.

http://studies.example.com/optimal-led-delays.html

Attach images by dragging & dropping or selecting them.

✔ Able to merge.
These branches can be automatically merged.

Create pull request

1 commit | 1 file changed | 0 commit comments | 1 contributor

Commits on Oct 01, 2014

tonychacon three seconds is better db44c53

Showing 1 changed file with 2 additions and 2 deletions. | Unified | Split

4 blink.ino | View

```
@@ -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    -  delay(1000);               // wait for a second
   21 +  delay(3000);               // wait for a second
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
24 24  }
```

Figura 91. Página de criação de pull request

Quando você clicar no botão 'Criar pull request' nesta tela, o proprietário do projeto que você bifurcou receberá uma notificação de que alguém está sugerindo uma alteração e criará um link para uma página que contém todas essas informações.

ÿ

Embora as Pull Requests sejam comumente usadas para projetos públicos como este, quando o colaborador tem uma mudança completa pronta para ser feita, ela também é frequentemente usada em projetos internos *no início* do ciclo de desenvolvimento. Como você pode continuar empurrando para a ramificação do tópico mesmo **após** a abertura da solicitação pull, ela geralmente é aberta antecipadamente e usada como uma forma de iterar no trabalho como uma equipe dentro de um contexto, em vez de ser aberta no final do processo.

Iterando em uma solicitação pull

Neste ponto, o proprietário do projeto pode olhar para a mudança sugerida e mesclá-la, rejeitá-la ou comentá-la. Digamos que ele goste da ideia, mas prefira um tempo um pouco mais longo para a luz ficar apagada do que acesa.

Machine Translated by Google

Onde essa conversa pode ocorrer por e-mail nos fluxos de trabalho apresentados no Distributed Git, no GitHub isso acontece online. O proprietário do projeto pode revisar as diferenças unificadas e deixar um comentário clicando em qualquer uma das linhas.

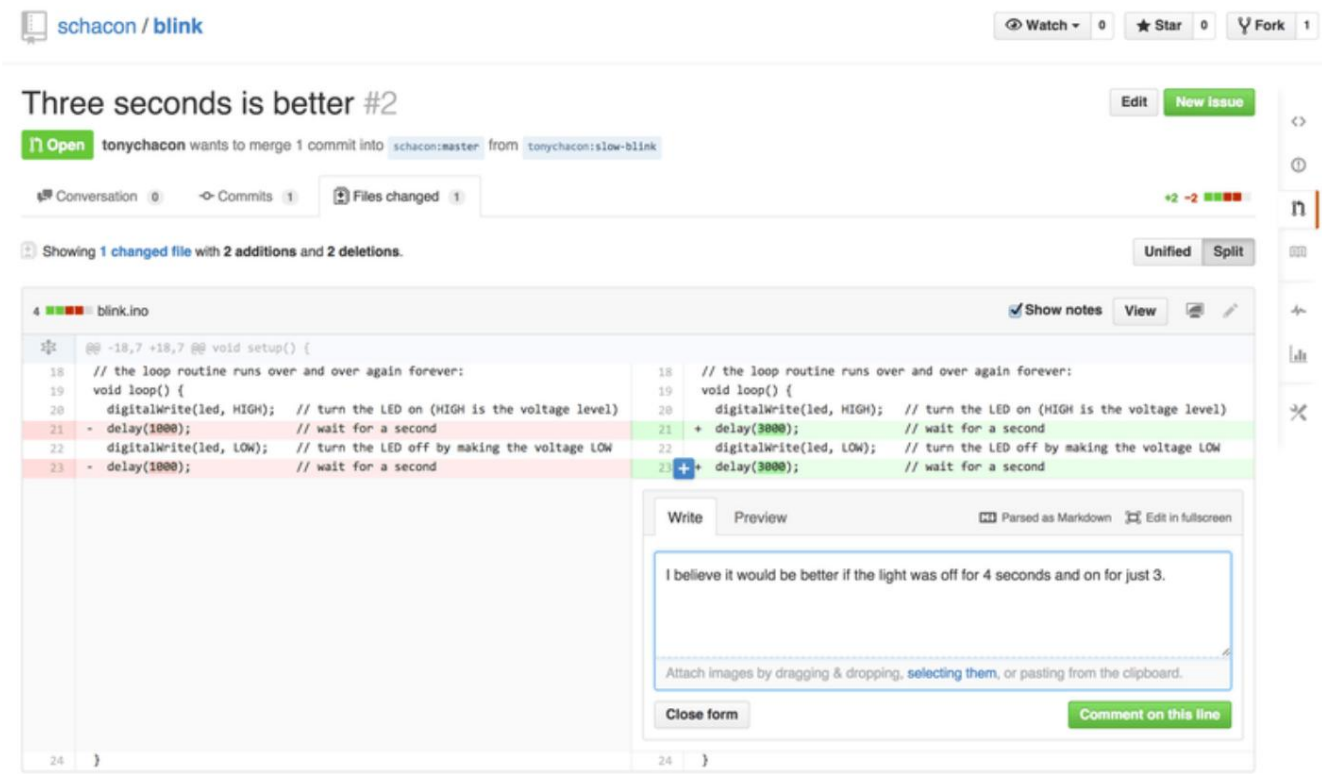


Figura 92. Comente uma linha específica de código em uma solicitação pull

Assim que o mantenedor fizer esse comentário, a pessoa que abriu o Pull Request (e, na verdade, qualquer outra pessoa que esteja observando o repositório) receberá uma notificação. Vamos personalizar isso mais tarde, mas se ele tivesse ativado as notificações por e-mail, Tony receberia um e-mail como este:

Figura 93. Comentários enviados como notificações por e-mail

Qualquer um também pode deixar comentários gerais sobre o Pull Request. Na página de discussão do Pull Request, podemos ver um exemplo do proprietário do projeto comentando em uma linha de código e deixando um comentário geral na seção de discussão. Você pode ver que os comentários do código também são incluídos na conversa.

Machine Translated by Google

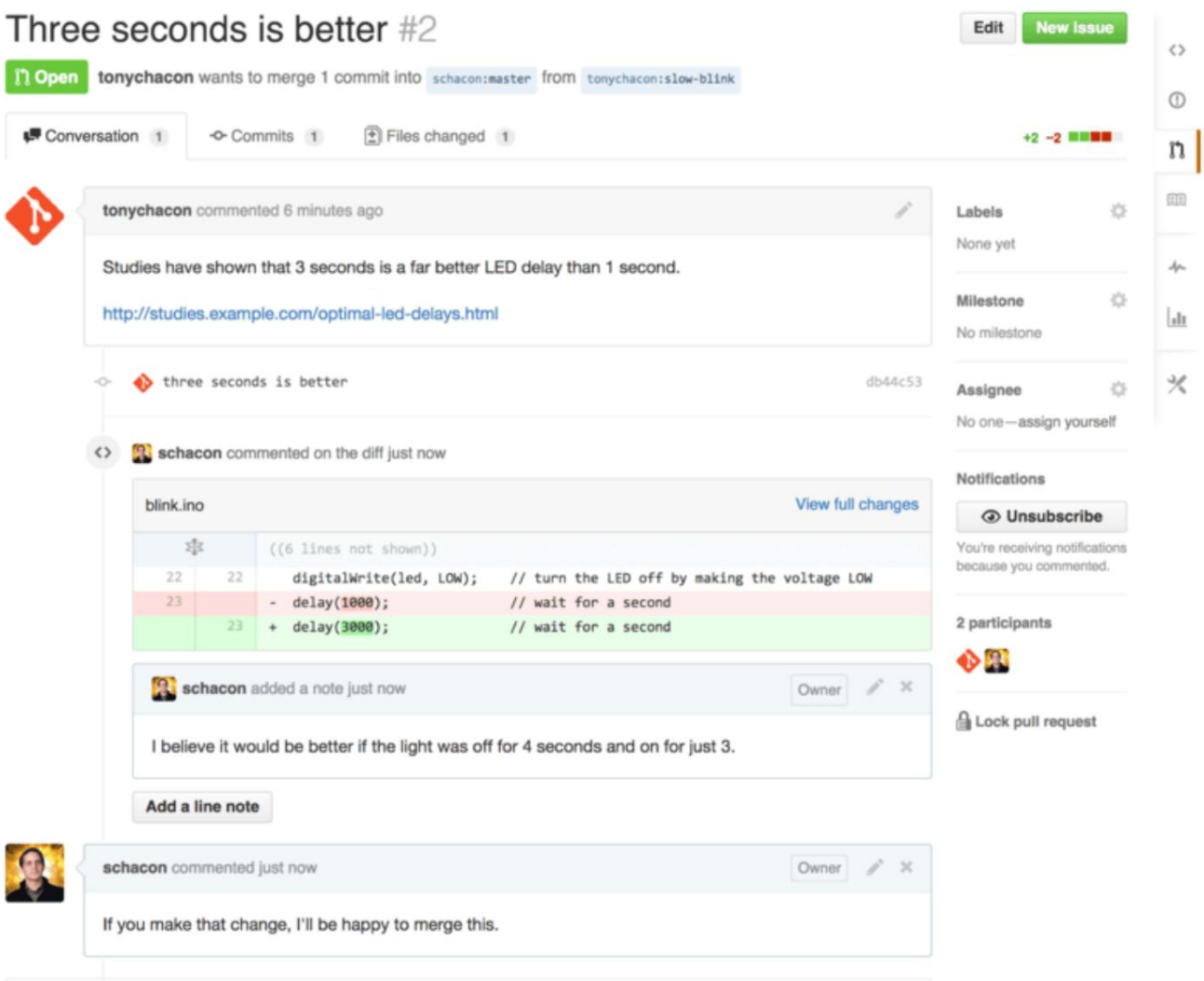


Figura 94. Página de discussão do Pull Request

Agora, o colaborador pode ver o que precisa fazer para que sua alteração seja aceita. Felizmente, isso é muito simples. Onde por e-mail você pode ter que rolar novamente sua série e reenviá-la para a lista de discussão, com o GitHub você simplesmente se compromete com a ramificação do tópico novamente e envia, o que atualizará automaticamente a solicitação pull. No final do Pull Request, você também pode ver que o comentário do código antigo foi recolhido no Pull Request atualizado, pois foi feito em uma linha que foi alterada desde então.

Adicionar commits a uma solicitação de pull existente não aciona uma notificação, portanto, assim que Tony envia suas correções, ele decide deixar um comentário para informar ao proprietário do projeto que ele fez a alteração solicitada.

Machine Translated by Google

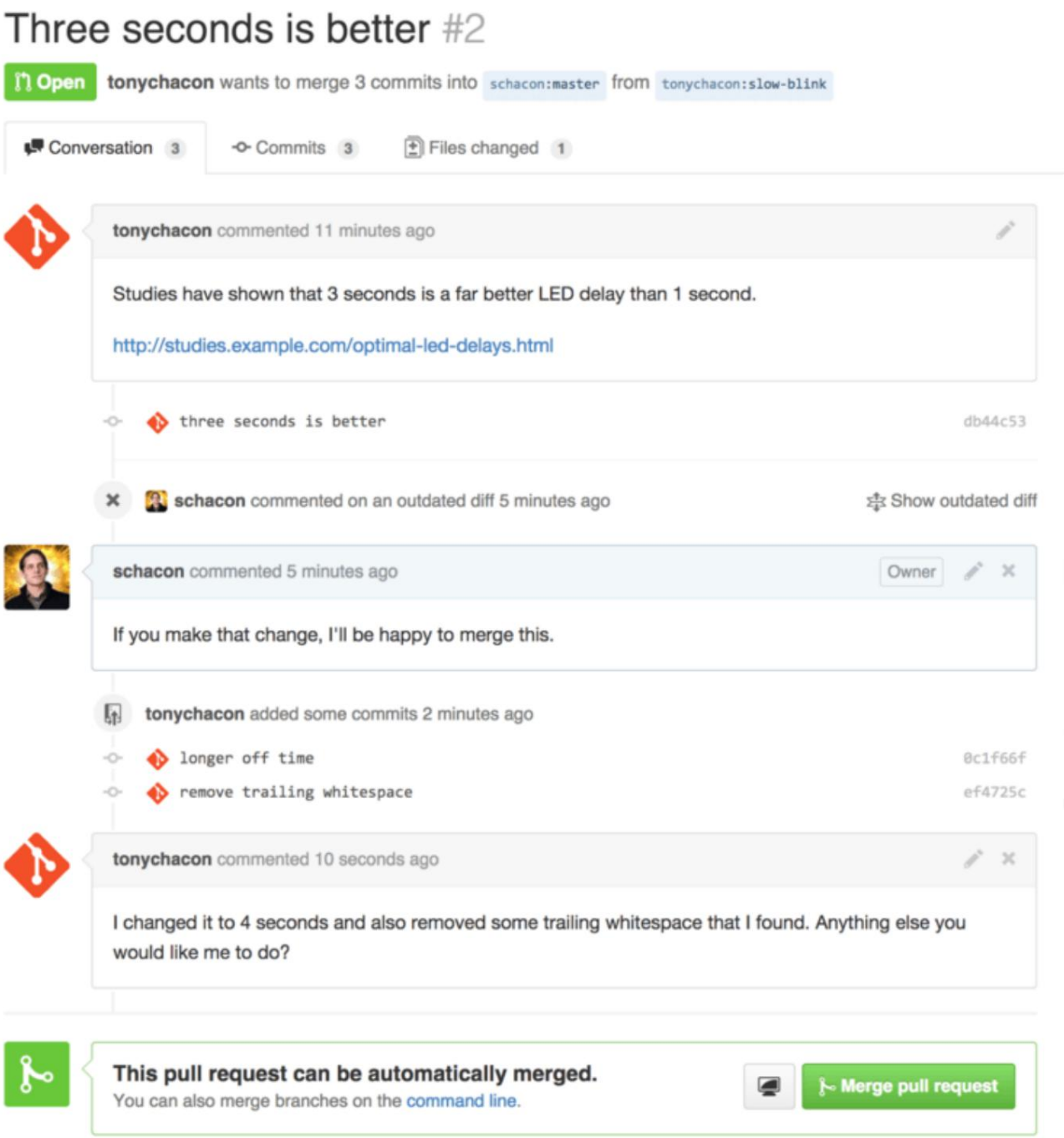


Figura 95. Pull Request final

Uma coisa interessante a notar é que se você clicar na aba “Files Changed” neste Pull Request, você obterá o diff “unificado” — ou seja, a diferença agregada total que seria introduzida em seu branch principal se este tópico branch foi mesclado. Nos termos do git diff , ele basicamente mostra automaticamente a você git diff master… <branch> para o branch em que esta solicitação pull é baseada. Consulte Determinando o que é introduzido para saber mais sobre esse tipo de diferença.

A outra coisa que você notará é que o GitHub verifica se a solicitação pull é mesclada corretamente e fornece um botão para fazer a mesclagem para você no servidor. Este botão só aparece se você tiver acesso de gravação ao repositório e uma mesclagem trivial for possível. Se você clicar nele, o GitHub executará uma mesclagem “sem avanço rápido”, o que significa que, mesmo que a mesclagem **possa** ser um avanço rápido, ela ainda criará uma confirmação de mesclagem.

Se preferir, você pode simplesmente puxar a ramificação para baixo e mesclá-la localmente. Se você mesclar isso

Machine Translated by Google

branch no branch master e empurre-o para o GitHub, o Pull Request será fechado automaticamente.

Esse é o fluxo de trabalho básico usado pela maioria dos projetos do GitHub. Ramificações de tópicos são criadas, Pull Requests são abertas nelas, uma discussão se inicia, possivelmente mais trabalho é feito na ramificação e, eventualmente, a solicitação é fechada ou mesclada.

ÿ

Não apenas garfos

É importante observar que você também pode abrir uma solicitação pull entre duas ramificações no mesmo repositório. Se você estiver trabalhando em um recurso com alguém e ambos tiverem acesso de gravação ao projeto, você pode enviar uma ramificação de tópico para o repositório e abrir uma solicitação pull para a ramificação principal desse mesmo projeto para iniciar a revisão do código e processo de discussão. Não é necessário bifurcar.

Solicitações pull avançadas

Agora que abordamos os fundamentos da contribuição para um projeto no GitHub, vamos abordar algumas dicas e truques interessantes sobre Pull Requests para que você possa usá-los com mais eficiência.

Solicitações pull como patches

É importante entender que muitos projetos realmente não pensam em Pull Requests como filas de patches perfeitos que devem ser aplicados de forma limpa em ordem, já que a maioria dos projetos baseados em listas de discussão pensa em contribuições em série de patches. A maioria dos projetos do GitHub considera as ramificações do Pull Request como conversas iterativas em torno de uma alteração proposta, culminando em um diff unificado que é aplicado por mesclagem.

Esta é uma distinção importante, porque geralmente a mudança é sugerida antes que o código seja considerado perfeito, o que é muito mais raro com contribuições de séries de patches baseadas em listas de discussão. Isso permite uma conversa anterior com os mantenedores para que chegar à solução adequada seja mais um esforço da comunidade. Quando o código é proposto com uma solicitação pull e os mantenedores ou a comunidade sugerem uma alteração, a série de patches geralmente não é refeita, mas a diferença é enviada como um novo commit para o branch, levando a conversa adiante com o contexto do trabalho anterior intacto.

Por exemplo, se você voltar e olhar novamente para o final do Pull Request, notará que o contribuidor não rebaseou seu commit e enviou outro Pull Request. Em vez disso, eles adicionaram novos commits e os enviaram para o branch existente. Dessa forma, se você voltar e olhar para este Pull Request no futuro, poderá encontrar facilmente todo o contexto do motivo pelo qual as decisões foram tomadas. Pressionar o botão “Mesclar” no site propositalmente cria um commit de mesclagem que faz referência à solicitação pull para que seja fácil voltar e pesquisar a conversa original, se necessário.

Acompanhando o Upstream

Se o seu Pull Request ficar desatualizado ou não for mesclado corretamente, você deve corrigi-lo para que o mantenedor possa facilmente fundi-lo. O GitHub testará isso para você e informará na parte inferior de cada solicitação pull se a mesclagem é trivial ou não.

Machine Translated by Google

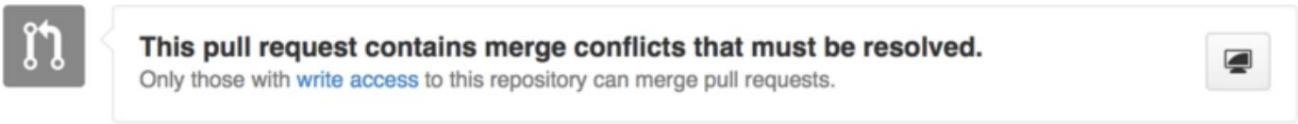


Figura 96. O pull request não é mesclado corretamente

Se você vir algo como Pull Request não mesclar corretamente, você deve corrigir sua ramificação para que fique verde e o mantenedor não precise fazer trabalho extra.

Você tem duas opções principais para fazer isso. Você pode rebasear sua ramificação em cima de qualquer que seja a ramificação de destino (normalmente a ramificação principal do repositório que você bifurcou) ou pode mesclar a ramificação de destino em sua ramificação.

A maioria dos desenvolvedores no GitHub escolherá fazer o último, pelos mesmos motivos que acabamos de ver na seção anterior. O que importa é o histórico e a mesclagem final, portanto, o rebase não oferece muito mais do que um histórico um pouco mais limpo e, em troca, é **muito** mais difícil e sujeito a erros.

Se você deseja mesclar na ramificação de destino para tornar sua solicitação pull mesclável, você deve adicionar o repositório original como um novo controle remoto, buscá-lo, mesclar a ramificação principal desse repositório em sua ramificação de tópico, corrigir quaisquer problemas e, finalmente, enviá-lo de volta para a mesma ramificação em que você abriu a solicitação pull.

Por exemplo, digamos que no exemplo “tonychacon” que estávamos usando antes, o autor original fez uma alteração que criaria um conflito no Pull Request. Vamos passar por essas etapas.

```
$ git remote add upstream https://github.com/schacon/blink ÿ

$ git fetch upstream ÿ remote:
Contando objetos: 3, concluído. remoto:
Comprimindo objetos: 100% (3/3), concluído.
Desempacotando objetos: 100% (3/3),
concluído. remoto: Total 3 (delta 0), reutilizado 0 (delta 0)
De https://github.com/schacon/blink * [nova
ramificação] -> upstream/mestre

$ git merge upstream/master ÿ
Mesclagem automática do blink.ino
CONFLICT (conteúdo): Conflito de mesclagem no blink.ino
Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.

$ vim blink.ino ÿ $ git
add blink.ino $ git commit
[slow-blink 3c8d735]
Merge remote-tracking branch 'upstream/master' \ em slow-blink


$ git push origin slow-blink ÿ Contando
objetos: 6, concluído.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (6/6), concluído.
```

Machine Translated by Google

```
Objetos de escrita: 100% (6/6), 682 bytes | 0 byte/s, pronto.
Total 6 (delta 2), reutilizado 0 (delta 0)
Para https://github.com/tonychacon/blink
ef4725c..3c8d735 piscar lento -> piscar lento
```

- Adicione o repositório original como um remoto chamado upstream.
- Busque o trabalho mais recente desse controle remoto.
- Mescle a ramificação principal desse repositório em sua ramificação de tópico.
- Corrija o conflito ocorrido.
- Empurre de volta para o mesmo ramo de tópico.

Depois de fazer isso, a solicitação pull será atualizada automaticamente e verificada novamente para ver se ela se mescla corretamente.

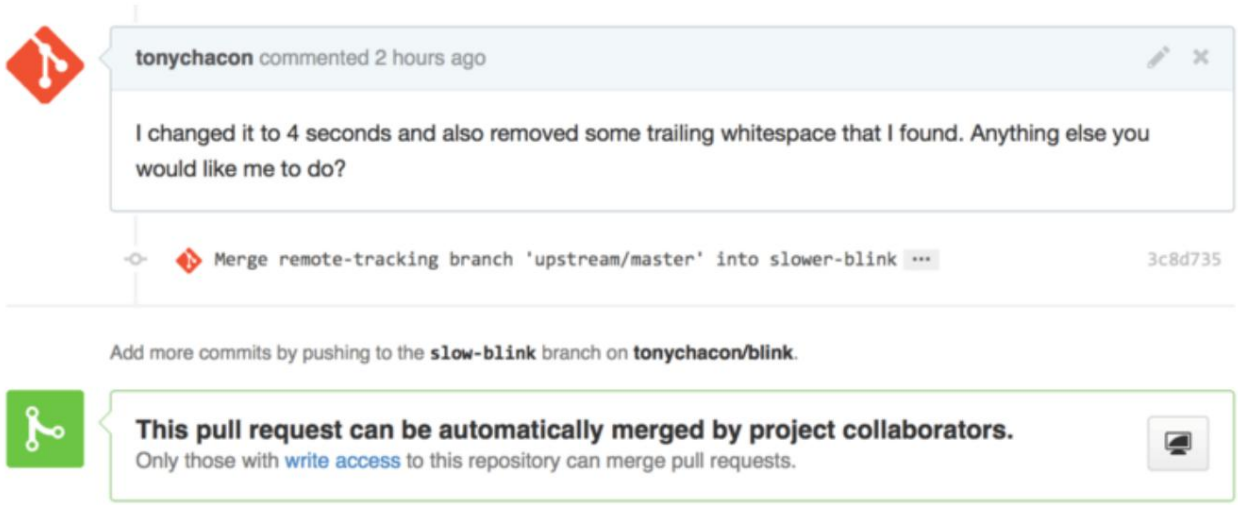


Figura 97. O Pull Request agora é mesclado de forma limpa

Uma das grandes vantagens do Git é que você pode fazer isso continuamente. Se você tiver um projeto de execução muito longa, poderá mesclar facilmente a partir da ramificação de destino repetidas vezes e só terá que lidar com os conflitos que surgiram desde a última vez que você fundiu, tornando o processo muito gerenciável.

Se você realmente deseja rebasear o branch para limpá-lo, certamente pode fazê-lo, mas é altamente recomendável não forçar o push sobre o branch em que o Pull Request já está aberto. Se outras pessoas o retiraram e trabalharam mais nele, você se depara com todos os problemas descritos em Os perigos do rebase. Em vez disso, envie o branch rebaseado para um novo branch no GitHub e abra um novo Pull Request referenciando o antigo e, em seguida, feche o original.

Referências

Sua próxima pergunta pode ser “Como faço referência ao antigo Pull Request?”. Acontece que existem muitas, muitas maneiras de fazer referência a outras coisas em quase qualquer lugar que você possa escrever no GitHub.

Vamos começar explicando como fazer referência cruzada a outra solicitação pull ou a um problema. Todos os Pull Requests e Issues são números atribuídos e são únicos dentro do projeto. Por exemplo, você não pode ter Pull Request nº 3 *e* Problema nº 3. Se você quiser fazer referência a qualquer solicitação pull ou problema de qualquer outro,

Machine Translated by Google

você pode simplesmente colocar #<num> em qualquer comentário ou descrição. Você também pode ser mais específico se a solicitação Issue ou Pull estiver em outro lugar; escreva nome de usuário#<num> se estiver se referindo a um problema ou solicitação pull em uma bifurcação do repositório em que você está, ou nome de usuário/repo#<num> para fazer referência a algo em outro repositório.

Vejamos um exemplo. Digamos que rebaseamos a ramificação no exemplo anterior, criamos uma nova solicitação pull para ela e agora queremos referenciar a solicitação pull antiga a partir da nova. Também queremos referenciar um problema na bifurcação do repositório e um problema em um projeto completamente diferente. Podemos preencher a descrição da mesma forma que as referências cruzadas em uma solicitação pull.

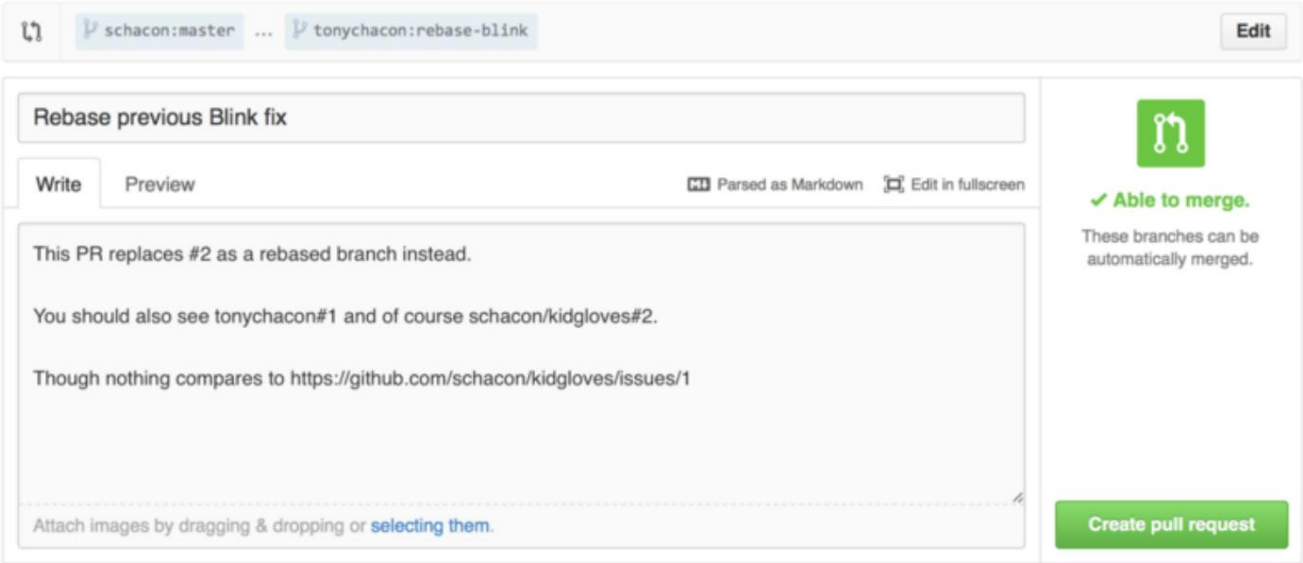


Figura 98. Referências cruzadas em uma solicitação pull

Quando enviarmos essa solicitação pull, veremos tudo isso renderizado como referências cruzadas renderizadas em uma solicitação pull.

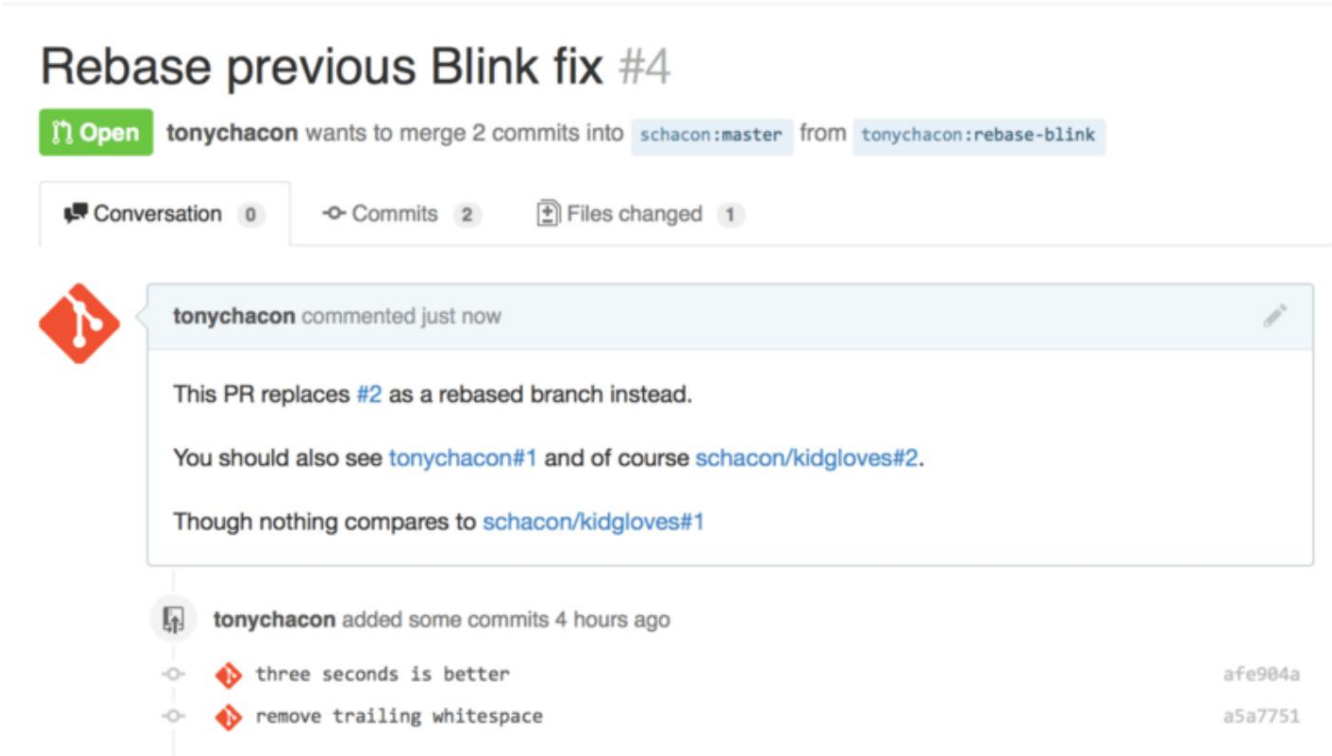


Figura 99. Referências cruzadas renderizadas em uma solicitação pull

Machine Translated by Google

Observe que o URL completo do GitHub que colocamos lá foi reduzido para apenas as informações necessárias.

Agora, se Tony voltar e fechar o Pull Request original, podemos ver que, ao mencioná-lo no novo, o GitHub criou automaticamente um evento de trackback na linha do tempo do Pull Request. Isso significa que qualquer pessoa que visitar este Pull Request e vir que ele está fechado pode facilmente vincular-se ao que o substituiu. O link se parecerá com Link de volta para a nova solicitação pull na linha do tempo fechada da solicitação pull.

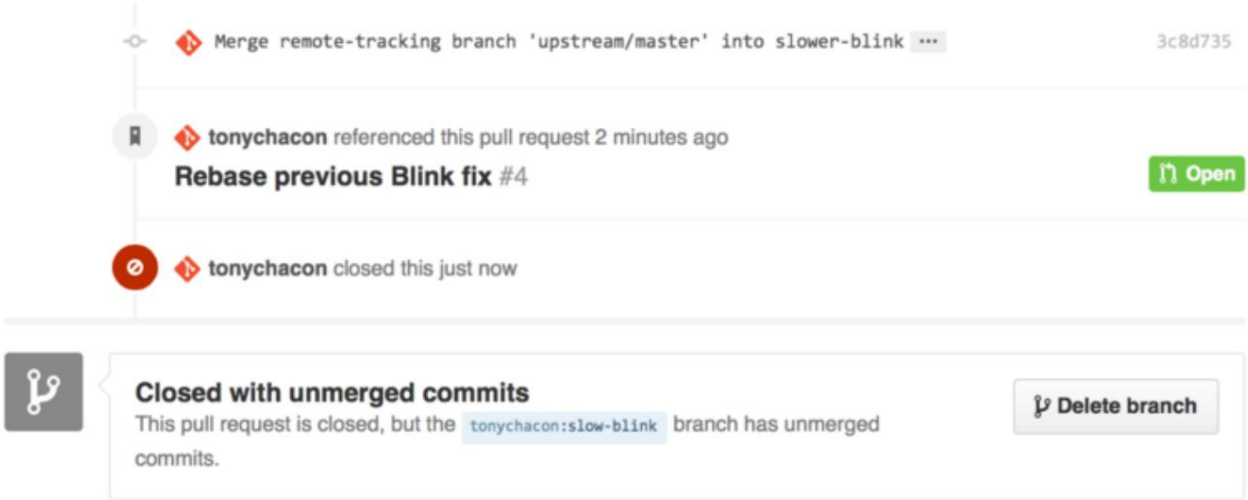


Figura 100. Link de volta para a nova solicitação pull na linha do tempo fechada da solicitação pull

Além dos números de emissão, você também pode fazer referência a um commit específico por SHA-1. Você precisa especificar um SHA-1 completo de 40 caracteres, mas se o GitHub vir isso em um comentário, ele será vinculado diretamente ao commit. Novamente, você pode fazer referência a commits em forks ou outros repositórios da mesma forma que fez com os problemas.

Marcação com sabor do GitHub

Vincular a outros problemas é apenas o começo das coisas interessantes que você pode fazer com praticamente qualquer caixa de texto no GitHub. Nas descrições de problemas e pull request, comentários, comentários de código e muito mais, você pode usar o que é chamado de “GitHub Flavored Markdown”. Markdown é como escrever em texto simples, mas renderizado de forma rica.

Consulte Um exemplo de GitHub Flavored Markdown como escrito e como renderizado para obter um exemplo de como comentários ou texto podem ser escritos e, em seguida, renderizados usando Markdown.

Machine Translated by Google

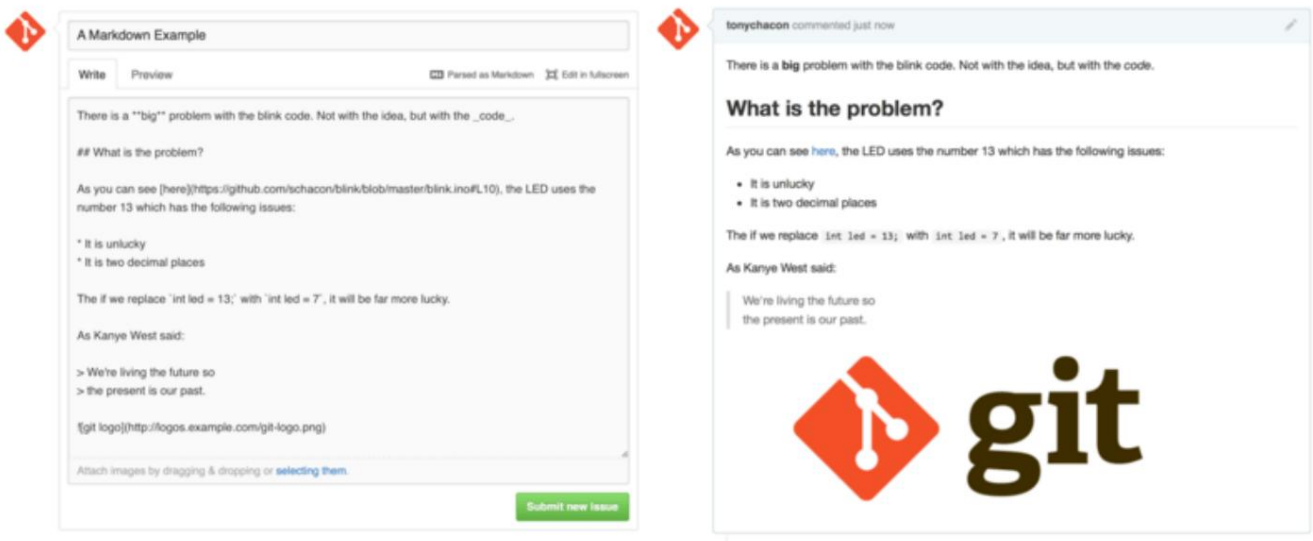


Figura 101. Um exemplo de GitHub Flavored Markdown conforme escrito e renderizado

O sabor GitHub do Markdown adiciona mais coisas que você pode fazer além da sintaxe básica do Markdown. Tudo isso pode ser realmente útil ao criar comentários ou descrições úteis de solicitação de pull ou problema.

Listas de Tarefas

O primeiro recurso Markdown específico do GitHub realmente útil, especialmente para uso em solicitações pull, é a lista de tarefas. Uma lista de tarefas é uma lista de caixas de seleção de coisas que você deseja realizar. Colocá-los em um problema ou pull request normalmente indica coisas que você deseja fazer antes de considerar o item concluído.

Você pode criar uma lista de tarefas como esta:

```
- [X] Escreva o código
- [ ] Escreva todos os testes
- [ ] Documente o código
```

Se incluirmos isso na descrição de nossa solicitação ou problema pull, veremos isso renderizado como listas de tarefas renderizadas em um comentário Markdown.

Figura 102. Listas de tarefas renderizadas em um comentário Markdown

Isso geralmente é usado em Pull Requests para indicar tudo o que você gostaria de fazer na ramificação antes que a Pull Request esteja pronta para ser mesclada. A parte muito legal é que você pode simplesmente clicar nas caixas de seleção para atualizar o comentário - você não precisa editar o Markdown diretamente para verificar as tarefas

Machine Translated by Google

desligado.

Além disso, o GitHub procurará listas de tarefas em seus problemas e solicitações pull e as mostrará como metadados nas páginas que as listam. Por exemplo, se você tiver uma Pull Request com tarefas e olhar para a página de visão geral de todas as Pull Requests, você pode ver o quão longe está feito. Isso ajuda as pessoas a dividir as solicitações pull em subtarefas e ajuda outras pessoas a acompanhar o progresso da ramificação.
Você pode ver um exemplo disso no Resumo da lista de tarefas na lista Pull Request.

Figura 103. Resumo da lista de tarefas na lista Pull Request

Eles são incrivelmente úteis quando você abre uma solicitação pull antecipadamente e a usa para acompanhar seu progresso na implementação do recurso.

Partes de codigo

Você também pode adicionar trechos de código aos comentários. Isso é especialmente útil se você quiser apresentar algo que *poderia* tentar fazer antes de realmente implementá-lo como um commit em sua ramificação.
Isso também é frequentemente usado para adicionar código de exemplo do que não está funcionando ou o que essa solicitação pull pode implementar.

Para adicionar um trecho de código, você deve “cercar” em backticks.

```
```java
for(int i=0 ;      i < 5 ; i++)

{ System.out.println("i is : }          " + eu);
```
```

Se você adicionar um nome de idioma como fizemos lá com 'java', o GitHub também tentará destacar a sintaxe do snippet. No caso do exemplo acima, acabaria sendo renderizado como exemplo de código limitado Renderizado.

Machine Translated by Google

Figura 104. Exemplo de código limitado renderizado

citando

Se estiver respondendo a uma pequena parte de um comentário longo, você pode citar seletivamente o outro comentário precedendo as linhas com o caractere > . Na verdade, isso é tão comum e útil que existe um atalho de teclado para isso. Se você destacar o texto em um comentário ao qual deseja responder diretamente e pressionar a tecla r , ele citará esse texto na caixa de comentários para você.

As citações são mais ou menos assim:

```
> Se é mais nobre da mente sofrer
> As fundas e flechas da fortuna ultrajante,

Qual é o tamanho dessas fundas e, em particular, dessas flechas?
```

Depois de renderizado, o comentário se parecerá com exemplo de citação renderizado.

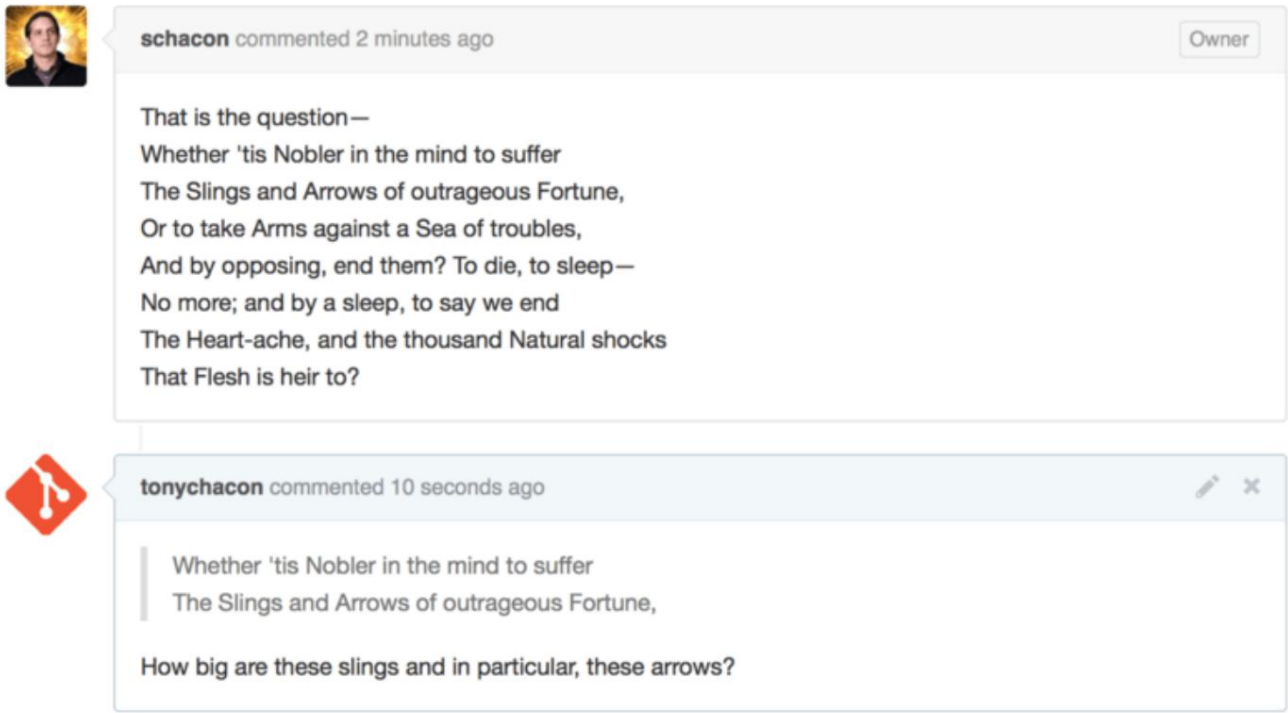


Figura 105. Exemplo de citação renderizado

Machine Translated by Google

Emoji

Finalmente, você também pode usar emoji em seus comentários. Na verdade, isso é usado extensivamente em comentários que você vê em muitos problemas do GitHub e solicitações pull. Existe até um auxiliar de emoji no GitHub. Se você estiver digitando um comentário e começar com um caractere : , um preenchimento automático ajudará você a encontrar o que está procurando.

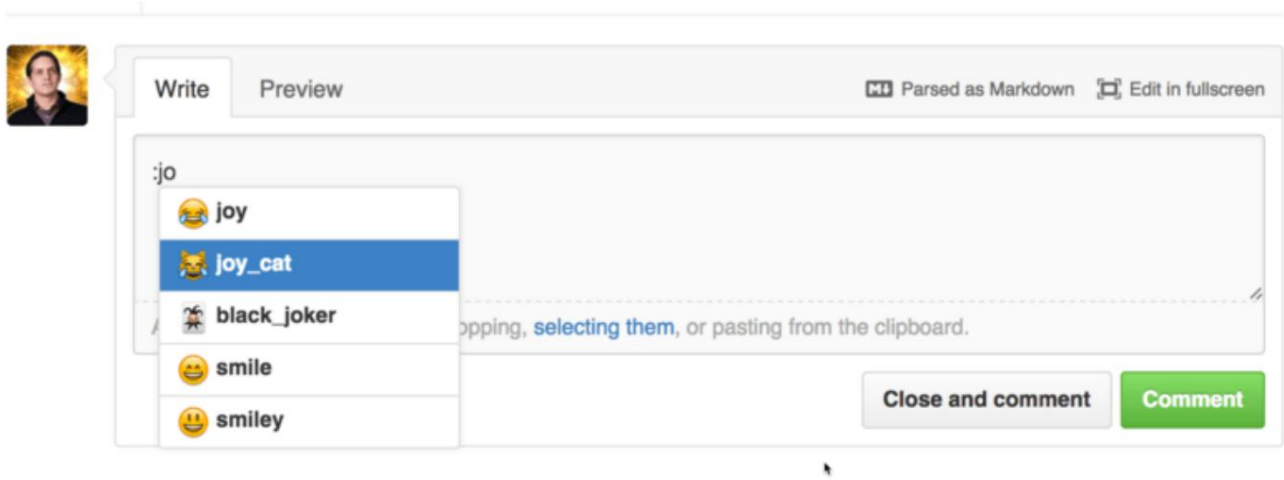


Figura 106. Autocompletador de emojis em ação

Os emojis assumem a forma de :<nome>: em qualquer lugar do comentário. Por exemplo, você poderia escrever algo assim:

```
Eu :eyes: aquele :bug: e eu :cold_sweat:.

:troféu: para :microscópio: isso.

:+1: e :sparkles: neste :navio:, é :fogo::cocô:!

:clap::tada::panda_face:
```

Quando renderizado, pareceria algo como comentário de emoji pesado.

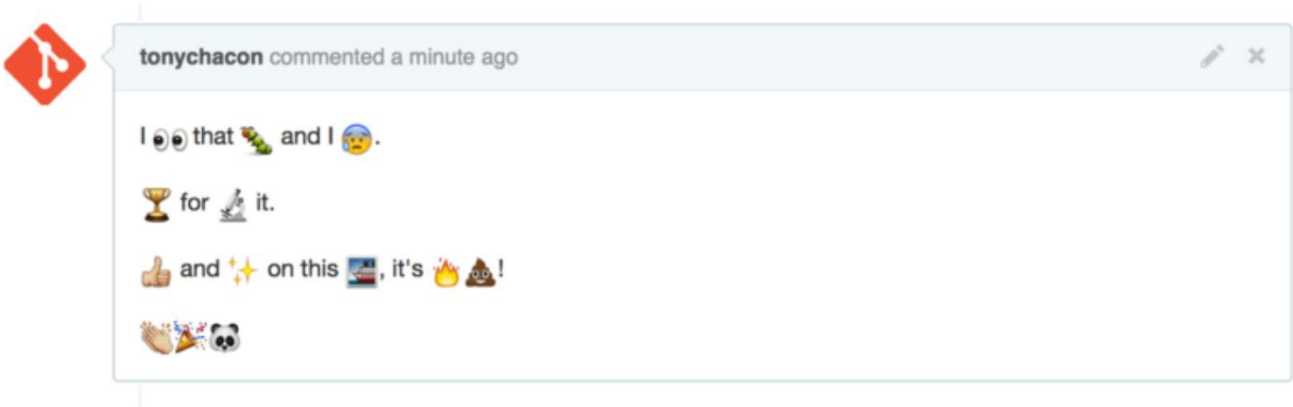


Figura 107. Comentário de emojis pesados

Não que isso seja incrivelmente útil, mas adiciona um elemento de diversão e emoção a um meio que, de outra forma, é difícil de transmitir emoção.

Machine Translated by Google

ÿ

Atualmente, existem vários serviços da Web que usam caracteres emoji. Uma ótima folha de referência para encontrar emojis que expressam o que você quer dizer pode ser encontrada em:

https://www.webfx.com/tools/emoji-cheat-sheet/

Imagens

Isso não é tecnicamente o GitHub Flavored Markdown, mas é incrivelmente útil. Além de adicionar links de imagem Markdown a comentários, que podem ser difíceis de encontrar e incorporar URLs, o GitHub permite arrastar e soltar imagens em áreas de texto para incorporá-las.

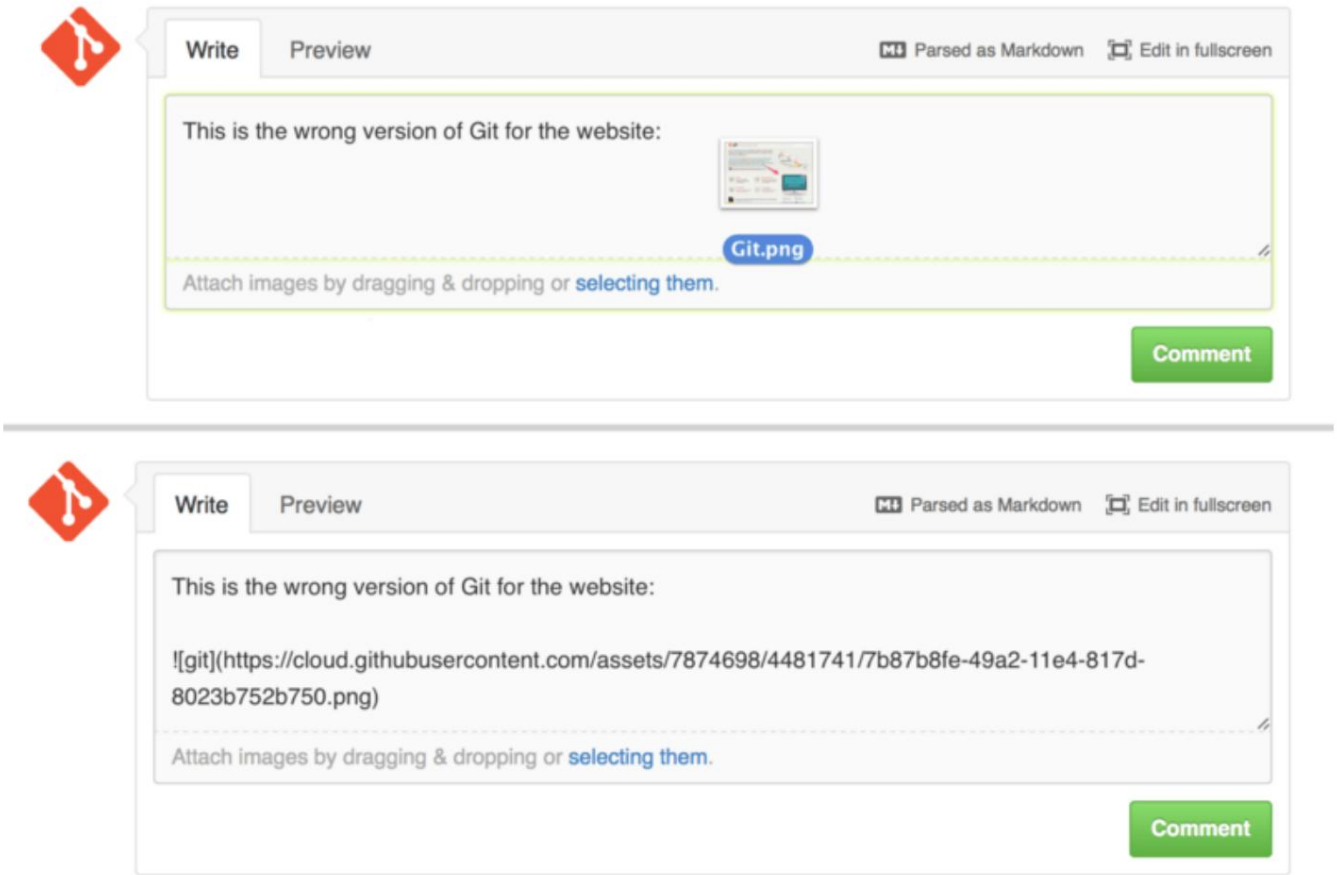


Figura 108. Arraste e solte as imagens para carregá-las e incorporá-las automaticamente

Se você olhar para Arrastar e soltar imagens para carregá-las e incorporá-las automaticamente, poderá ver uma pequena dica “Analisada como Markdown” acima da área de texto. Clicar nele fornecerá uma folha de dicas completa de tudo o que você pode fazer com o Markdown no GitHub.

Mantenha seu repositório público do GitHub atualizado

Depois de bifurcar um repositório do GitHub, seu repositório (seu "fork") existe independentemente do original. Em particular, quando o repositório original possui novos commits, o GitHub o informa por meio de uma mensagem como:

> Este branch está 5 commits atrás de progit:master.

Machine Translated by Google

Mas seu repositório GitHub nunca será atualizado automaticamente pelo GitHub; isso é algo que você deve fazer sozinho. Felizmente, isso é muito fácil de fazer.

Uma possibilidade de fazer isso não requer configuração. Por exemplo, se você fork de https://github.com/progit/progit2.git, você pode manter seu branch master atualizado assim:

```
$ git checkout master ÿ $
git pull https://github.com/progit/progit2.git ÿ $ git push
origin master ÿ
```

ÿ Se você estava em outro branch, retorne ao master.

ÿ Busque alterações em https://github.com/progit/progit2.git e mesclá-los em master.

ÿ Empurre sua ramificação principal para a origem.

Isso funciona, mas é um pouco tedioso ter que soletrar o URL de busca todas as vezes. Você pode automatizar esse trabalho com um pouco de configuração:

```
$ git remote add progit https://github.com/progit/progit2.git ÿ $ git
fetch progit ÿ $ git branch --set-upstream-to=progit/master master ÿ
$ git config --local remote.pushDefault origin ÿ
```

ÿ Adicione o repositório de origem e dê um nome a ele. Aqui, escolhi chamá-lo de progit.

ÿ Obtenha uma referência sobre os branches do progit, em particular o master.

ÿ Defina sua ramificação master para buscar no remoto progit .

ÿ Defina o repositório push padrão para a origem.

Feito isso, o fluxo de trabalho se torna muito mais simples:

```
$ git checkout master ÿ $
git pull ÿ $ git push ÿ
```

ÿ Se você estava em outro branch, retorne ao master.

ÿ Busque as alterações do progit e mescle as alterações no master.

ÿ Empurre sua ramificação principal para a origem.

Essa abordagem pode ser útil, mas não é isenta de desvantagens. O Git ficará feliz em fazer esse trabalho para você silenciosamente, mas não irá avisá-lo se você fizer um commit no master, puxar do progit e então enviar para o origin — todas essas operações são válidas com essa configuração. Portanto, você terá que tomar cuidado para nunca se comprometer diretamente com o master, já que esse branch pertence efetivamente ao repositório upstream.

Machine Translated by Google

Mantendo um Projeto

Agora que nos sentimos confortáveis em contribuir para um projeto, vamos ver o outro lado: criar, manter e administrar seu próprio projeto.

Criando um novo repositório

Vamos criar um novo repositório para compartilhar o código do nosso projeto. Comece clicando no botão “Novo repositório” no lado direito do painel ou no botão + na barra de ferramentas superior ao lado do seu nome de usuário, conforme visto no menu suspenso “Novo repositório”.

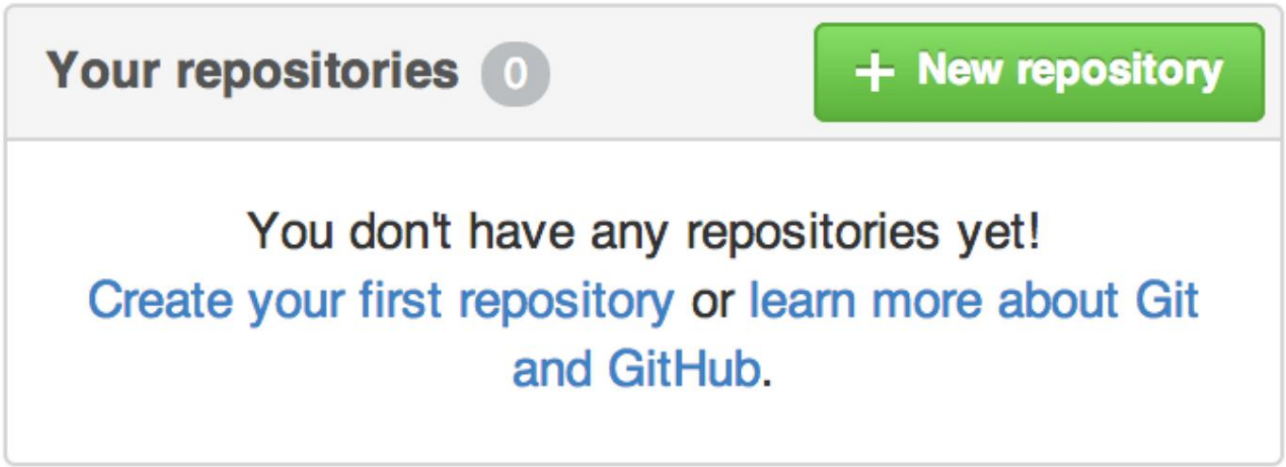


Figura 109. A área “Seus repositórios”

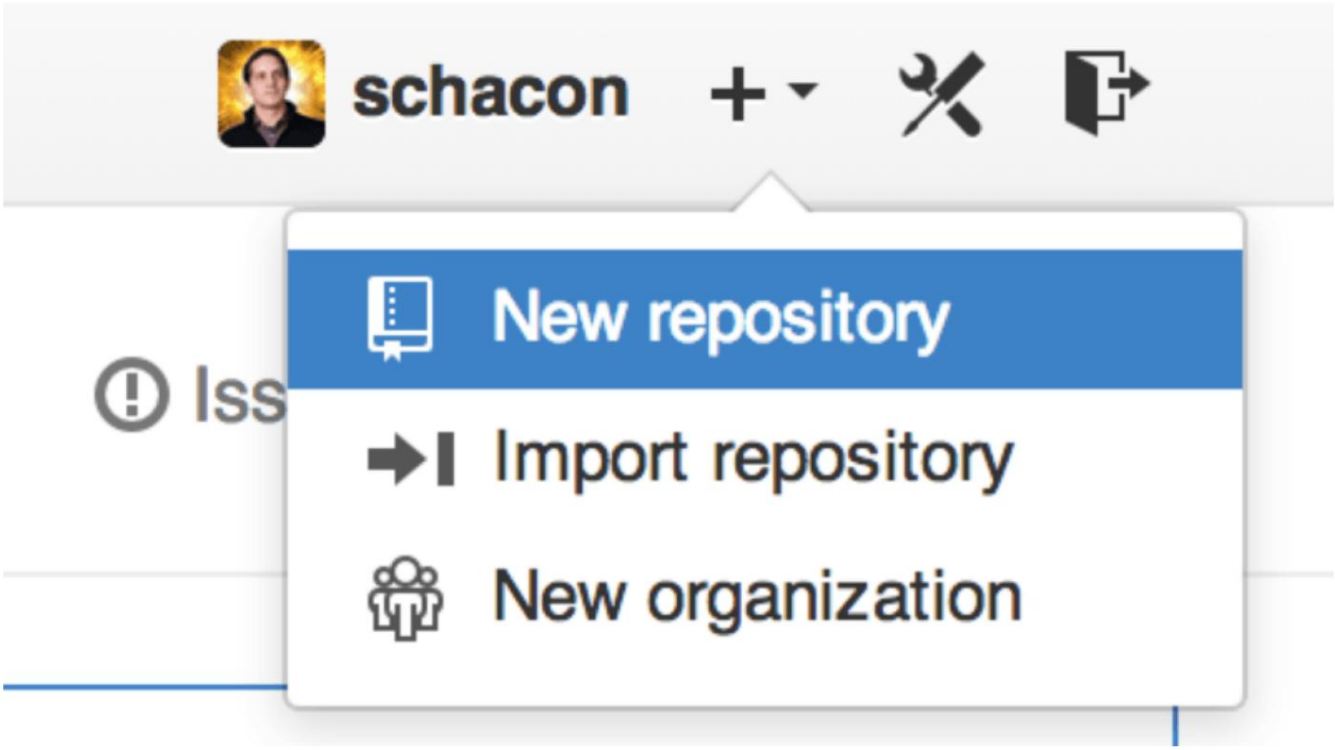


Figura 110. A lista suspensa “Novo repositório”

Isso leva você ao formulário “novo repositório”:

Machine Translated by Google

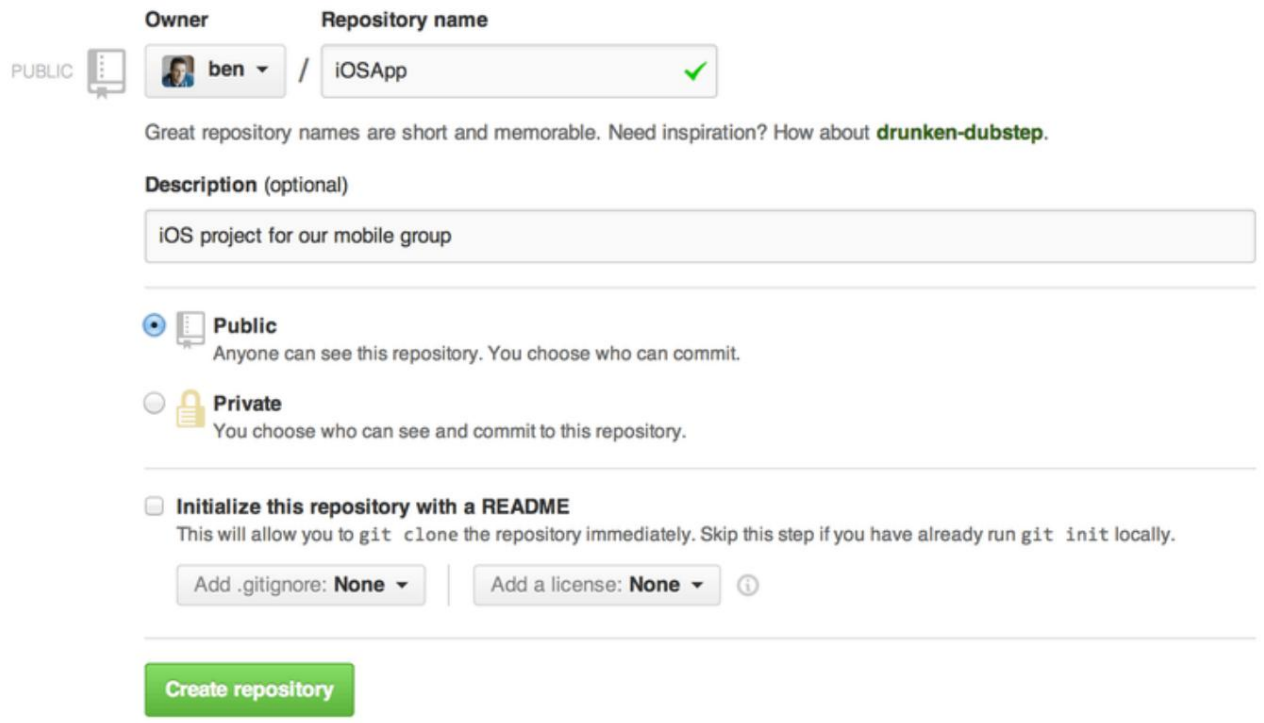


Figura 111. O formulário "novo repositório"

Tudo o que você realmente precisa fazer aqui é fornecer um nome de projeto; o resto dos campos são totalmente opcionais. Por enquanto, basta clicar no botão "Create Repository" e pronto – você tem um novo repositório no GitHub, chamado <user>/<project_name>.

Como você ainda não tem código, o GitHub mostrará instruções sobre como criar um novo repositório Git ou conectar um projeto Git existente. Não vamos insistir nisso aqui; se precisar de uma atualização, confira Git Basics.

Agora que seu projeto está hospedado no GitHub, você pode fornecer a URL para qualquer pessoa com quem deseja compartilhar seu projeto. Todo projeto no GitHub pode ser acessado por HTTPS como https://github.com/<user>/<project_name> ou por SSH como git@github.com:<user>/<project_name>.
O Git pode buscar e enviar para essas duas URLs, mas elas são controladas pelo acesso com base nas credenciais do usuário que se conecta a elas.

ÿ

Muitas vezes, é preferível compartilhar a URL baseada em HTTPS para um projeto público, pois o usuário não precisa ter uma conta do GitHub para acessá-la para clonagem. Os usuários terão que ter uma conta e uma chave SSH carregada para acessar seu projeto se você fornecer a URL SSH. O HTTPS também é exatamente o mesmo URL que eles colariam em um navegador para visualizar o projeto lá.

Adicionando Colaboradores

Se você estiver trabalhando com outras pessoas às quais deseja conceder acesso de confirmação, precisará adicioná-las como "colaboradores". Se Ben, Jeff e Louise se inscreverem em contas no GitHub e você quiser conceder a eles acesso push ao seu repositório, poderá adicioná-los ao seu projeto. Fazer isso dará a eles acesso "push", o que significa que eles têm acesso de leitura e gravação ao projeto e ao repositório Git.

Clique no link "Configurações" na parte inferior da barra lateral direita.

Machine Translated by Google

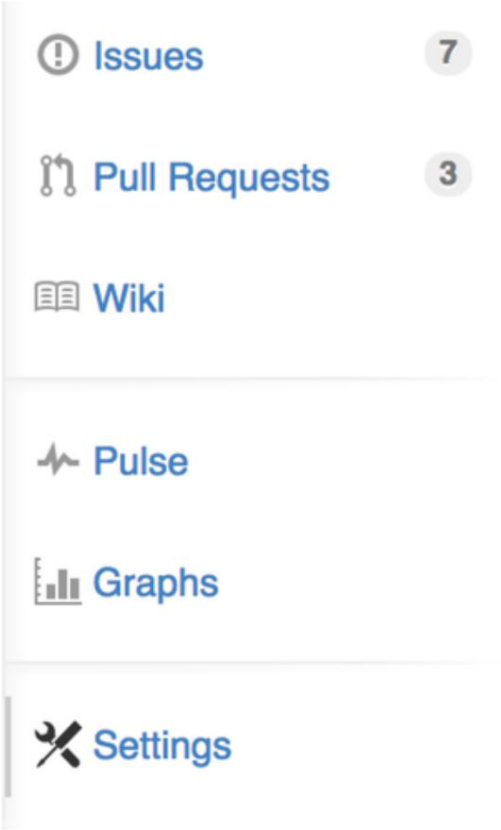


Figura 112. O link de configurações do repositório

Em seguida, selecione “Colaboradores” no menu à esquerda. Em seguida, basta digitar um nome de usuário na caixa e clicar em “Adicionar colaborador”. Você pode repetir isso quantas vezes quiser para conceder acesso a todos que desejar. Se você precisar revogar o acesso, basta clicar no “X” no lado direito do linha.

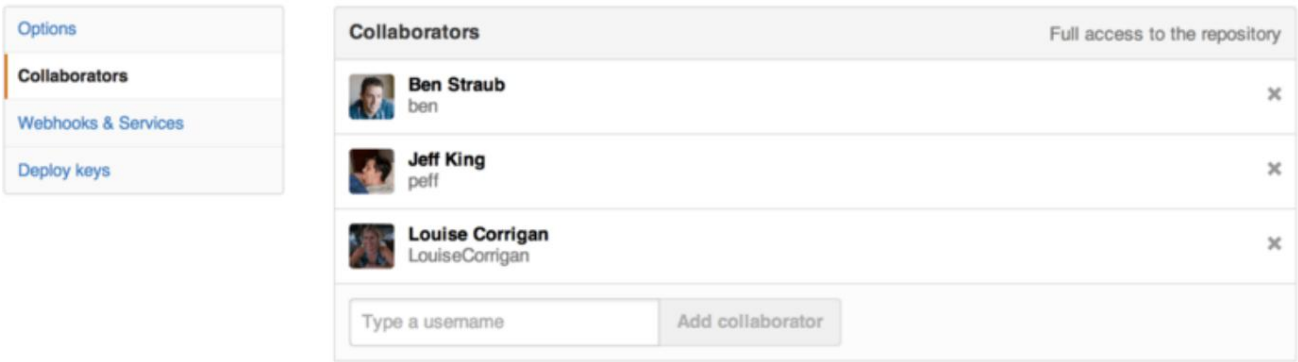


Figura 113. Colaboradores do repositório

Gerenciando solicitações pull

Agora que você tem um projeto com algum código e talvez até mesmo alguns colaboradores que também tenham acesso push, vamos ver o que fazer quando você mesmo receber uma solicitação pull.

As solicitações pull podem vir de uma ramificação em uma bifurcação do seu repositório ou podem vir de outra ramificação no mesmo repositório. A única diferença é que os que estão em uma bifurcação geralmente são de pessoas onde você não pode fazer push para o branch deles e eles não podem fazer push para o seu, enquanto com Pull Requests internos geralmente ambas as partes podem acessar o branch.

Machine Translated by Google

Para esses exemplos, vamos supor que você seja “tonychacon” e tenha criado um novo projeto de código Arduino chamado “fade”.

Notificações por e-mail

Alguém chega e faz uma alteração no seu código e envia uma solicitação pull. Você deve receber um e-mail notificando sobre a nova solicitação pull e deve se parecer com uma notificação por e-mail de uma nova solicitação pull.

[fade] Wait longer to see the dimming effect better (#1)

Scott Chacon <notifications@github.com> 10:05 AM (0 minutes ago)
to tonychacon/fade Unsubscribe

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

https://github.com/tonychacon/fade/pull/1

Commit Summary

- wait longer to see the dimming effect better

File Changes

- **M** fade.ino (2)

Patch Links:

- https://github.com/tonychacon/fade/pull/1.patch
- https://github.com/tonychacon/fade/pull/1.diff

—
Reply to this email directly or view it on GitHub.

Figura 114. Notificação por e-mail de uma nova solicitação pull

Há algumas coisas a serem observadas sobre este e-mail. Ele lhe dará um pequeno diffstat - uma lista de arquivos que foram alterados no Pull Request e por quanto. Ele fornece um link para o Pull Request no GitHub. Ele também fornece alguns URLs que você pode usar na linha de comando.

Se você notar a linha que diz git pull <url> patch-1, esta é uma maneira simples de mesclar em uma ramificação remota sem ter que adicionar um remoto. Examinamos isso rapidamente em Check-out de ramificações remotas. Se desejar, você pode criar e alternar para uma ramificação de tópico e, em seguida, executar este comando para mesclar as alterações do Pull Request.

As outras URLs interessantes são as URLs .diff e .patch , que, como você pode imaginar, fornecem versões unificadas de diff e patch do Pull Request. Você poderia tecnicamente mesclar no trabalho de solicitação de pull com algo assim:

```
$ curl https://github.com/tonychacon/fade/pull/1.patch | git am
```

Machine Translated by Google

Colaborando no Pull Request

Conforme abordamos no The GitHub Flow, agora você pode conversar com a pessoa que abriu o Pull Request. Você pode comentar em linhas específicas de código, comentar em commits inteiros ou comentar em toda a solicitação de pull, usando o GitHub Flavored Markdown em todos os lugares.

Toda vez que alguém comentar no Pull Request, você continuará recebendo notificações por e-mail para saber que há atividade acontecendo. Cada um deles terá um link para o Pull Request onde a atividade está acontecendo e você também pode responder diretamente ao e-mail para comentar o tópico do Pull Request.

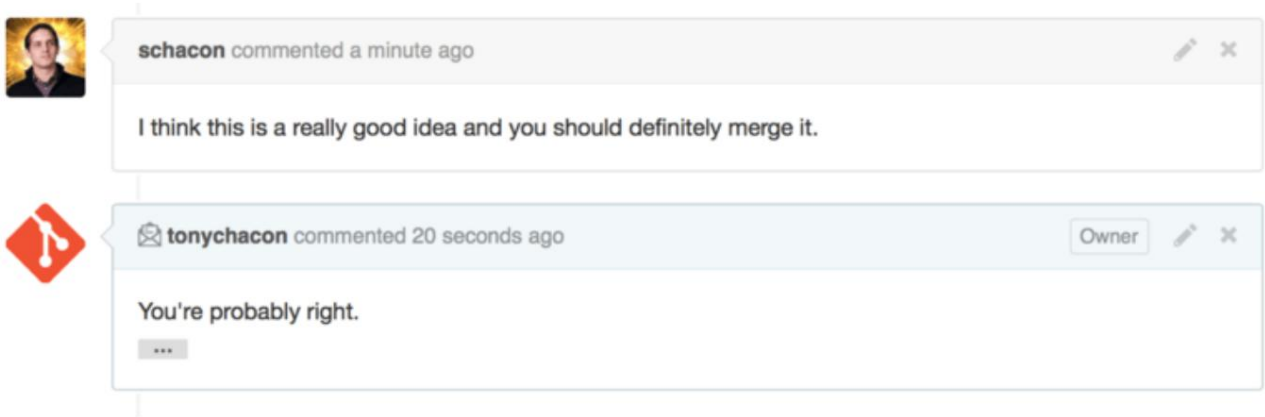


Figura 115. As respostas aos e-mails são incluídas no thread

Uma vez que o código esteja em um lugar que você goste e queira mesclá-lo, você pode puxar o código para baixo e mesclá-lo localmente, seja com a sintaxe git pull <url> <branch> que vimos anteriormente ou adicionando a bifurcação como um controle remoto e busca e fusão.

Se a mesclagem for trivial, você também pode clicar no botão “Mesclar” no site do GitHub. Isso fará uma mesclagem “sem avanço rápido”, criando uma confirmação de mesclagem mesmo se uma mesclagem de avanço rápido fosse possível. Isso significa que não importa o que aconteça, toda vez que você apertar o botão de mesclagem, um commit de mesclagem é criado. Como você pode ver no botão Mesclar e nas instruções para mesclar uma solicitação pull manualmente, o GitHub fornece todas essas informações se você clicar no link da dica.

Machine Translated by Google

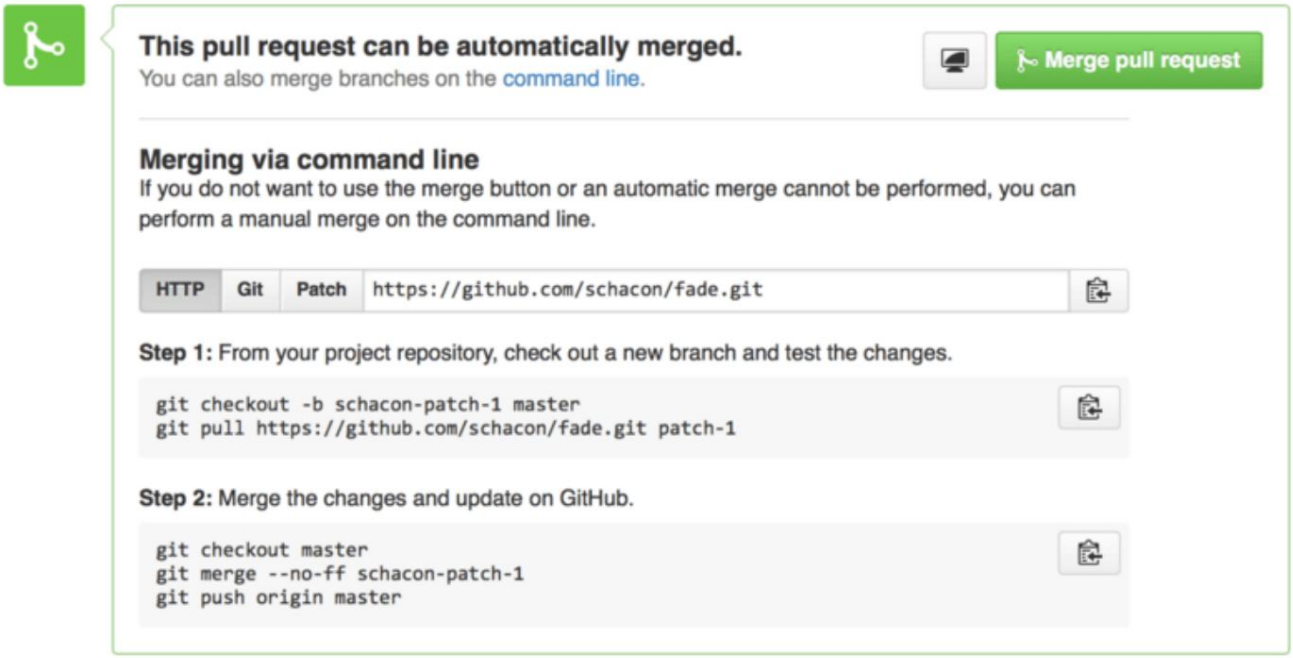


Figura 116. Botão de mesclagem e instruções para mesclar uma solicitação pull manualmente

Se você decidir que não deseja mesclar, também pode simplesmente fechar o Pull Request e a pessoa que o abriu será notificada.

Referências de solicitação de pull

Se você está lidando com muitas **solicitações** de pull e não deseja adicionar um monte de controles remotos ou fazer pulls únicos todas as vezes, há um truque legal que o GitHub permite que você faça. Este é um truque um pouco avançado e veremos os detalhes dele um pouco mais no The Refspec, mas pode ser bastante útil.

Na verdade, o GitHub anuncia as ramificações da solicitação pull para um repositório como uma espécie de pseudo-ramificações no servidor. Por padrão, você não os obtém ao clonar, mas eles estão lá de maneira obscura e você pode acessá-los com bastante facilidade.

Para demonstrar isso, vamos usar um comando de baixo nível (muitas vezes chamado de comando “encanamento”, sobre o qual leremos mais em Encanamento e porcelana) chamado ls-remote. Esse comando geralmente não é usado nas operações diárias do Git, mas é útil para nos mostrar quais referências estão presentes no servidor.

Se executarmos este comando no repositório “blink” que estávamos usando anteriormente, obteremos uma lista de todos os branches, tags e outras referências no repositório.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d HEAD
10D539600D86723087810EC636870a504f4fee4D    refs/heads/master
6A83107c62950be9453AAC297B0193fd743CD6E     refs/pull/1/head refs/
AFE83C2D1A70674505cc1D8B7D380D5E0766ed3     pull/1/merge refs/
3C8D735e                                    pull/2/head refs/pull/
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    2/merge refs/pull/4/
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    head refs/pull/4/
31a45fc257e8433c8d8804e3e848cf61c9d3166c    merge
```

Machine Translated by Google

Claro, se você estiver em seu repositório e executar git ls-remote origin ou qualquer controle remoto que queira verificar, ele mostrará algo semelhante a isso.

Se o repositório estiver no GitHub e você tiver alguma solicitação pull aberta, obterá essas referências prefixadas com refs/pull/. Estes são basicamente ramos, mas como eles não estão sob refs/heads/ você não os obtém normalmente quando clona ou busca no servidor — o processo de busca os ignora normalmente.

Existem duas referências por Pull Request - aquela que termina em /head aponta exatamente para o mesmo commit que o último commit na ramificação Pull Request. Portanto, se alguém abrir um Pull Request em nosso repositório e seu branch for chamado bug-fix e apontar para o commit a5a775, então em **nosso** repositório não teremos um bug-fix branch (já que está em seu fork), mas *teremos* pull/<pr#>/head que aponta para a5a775. Isso significa que podemos facilmente puxar para baixo todas as ramificações do Pull Request de uma só vez, sem ter que adicionar um monte de controles remotos.

Agora, você pode fazer algo como buscar a referência diretamente.

```
$ git fetch origin refs/pull/958/head De
https://github.com/libgit2/libgit2 * branch
refs/pull/958/head -> FETCH_HEAD
```

Isso diz ao Git: “Conecte-se ao controle remoto de origem e baixe o ref chamado refs/pull/958/head”. O Git obedece alegremente e baixa tudo o que você precisa para construir esse ref, e coloca um ponteiro para o commit que você deseja em .git/FETCH_HEAD. Você pode acompanhar isso com git merge FETCH_HEAD em uma ramificação na qual deseja testá-lo, mas essa mensagem de confirmação de mesclagem parece um pouco estranha. Além disso, se você estiver revisando muitas **solicitações** pull, isso se tornará entediante.

Há também uma maneira de buscar *todas* as solicitações pull e mantê-las atualizadas sempre que você se conectar ao controle remoto. Abra .git/config em seu editor favorito e procure pelo controle remoto de origem . Deve ficar um pouco assim:

```
[remote "origin"]
url = https://github.com/libgit2/libgit2 fetch =
+refs/heads/*:refs/remotes/origin/*
```

Essa linha que começa com fetch = é um “refspec.” É uma forma de mapear nomes no controle remoto com nomes em seu diretório .git local . Este em particular diz ao Git, "as coisas no controle remoto que estão em refs/heads devem ir para meu repositório local em refs/remotes/origin." Você pode modificar esta seção para adicionar outra refspec:

```
[remote "origin"]
url = https://github.com/libgit2/libgit2.git fetch =
+refs/heads/*:refs/remotes/origin/* fetch = +refs/
pull/*/head:refs/ controles remotos/origem/pr/*
```

Essa última linha diz ao Git: “Todas as referências que se parecem com refs/pull/123/head devem ser armazenadas localmente como

Machine Translated by Google

refs/remotes/origin/pr/123." Agora, se você salvar esse arquivo e fizer um git fetch:

```
$ git fetch #
… * [nova ref]
* [nova ref] *          refs/puxar/1/cabeça -> origem/pr/1 refs/
[nova ref] # …          puxar/2/cabeça -> origem/pr/2 refs/
                        puxar/4/cabeça -> origem/pr/4
```

Agora, todas as solicitações pull remotas são representadas localmente com refs que agem como rastreamento de ramificações; eles são somente leitura e são atualizados quando você faz uma busca. Isso torna super fácil tentar o código de uma solicitação pull localmente:

```
$ git checkout pr/2
Verificando arquivos: 100% (3769/3769), concluído.
Filial pr/2 configurada para rastrear ramificação remota pr/2 da origem.
Mudou para uma nova ramificação 'pr/2'
```

Os olhos de águia entre vocês notariam a cabeça no final da parte remota do refspec.
Há também um refs/pull/#/merge ref no lado do GitHub, que representa o commit que resultaria se você apertasse o botão "merge" no site. Isso pode permitir que você teste a mesclagem antes mesmo de apertar o botão.

Solicitações pull em solicitações pull

Você não apenas pode abrir Pull Requests direcionados à ramificação principal ou master , como também pode abrir uma Pull Request direcionada a qualquer branch na rede. Na verdade, você pode até direcionar outra solicitação pull.

Se você vir uma solicitação pull que está se movendo na direção certa e tiver uma ideia para uma alteração que depende dela ou não tiver certeza se é uma boa ideia, ou simplesmente não tiver acesso push à ramificação de destino, você pode abrir um Pull Request diretamente para ele.

Quando você abre uma solicitação pull, há uma caixa no topo da página que especifica para qual ramificação você está solicitando pull e de qual você está solicitando pull. Se você clicar no botão "Editar" à direita dessa caixa, poderá alterar não apenas as ramificações, mas também qual bifurcação.

Machine Translated by Google

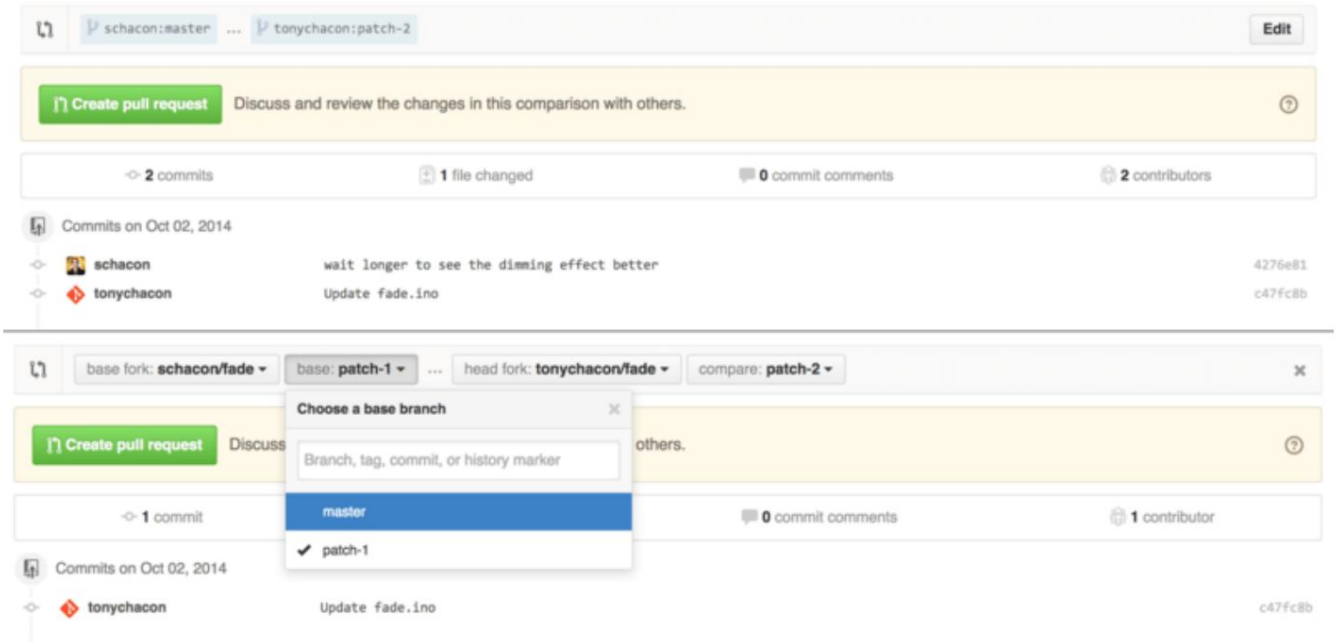


Figura 117. Altere manualmente a bifurcação e ramificação de destino da solicitação pull

Aqui você pode especificar facilmente para mesclar sua nova ramificação em outra solicitação pull ou outra bifurcação do projeto.

Menções e Notificações

O GitHub também possui um sistema de notificações muito bom integrado que pode ser útil quando você tiver dúvidas ou precisar de feedback de indivíduos ou equipes específicas.

Em qualquer comentário, você pode começar a digitar um caractere @ e ele começará a ser preenchido automaticamente com os nomes e nomes de usuário das pessoas que são colaboradores ou contribuintes do projeto.

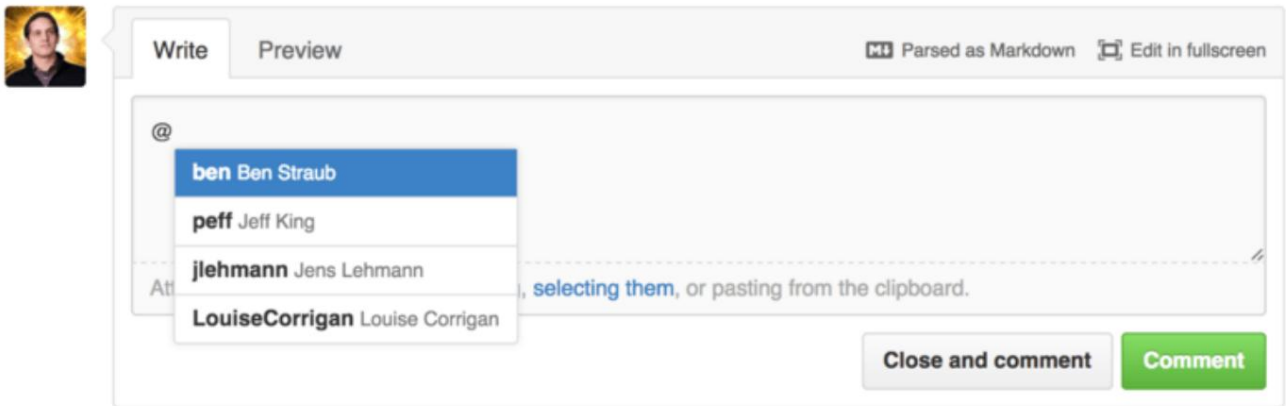


Figura 118. Comece a digitar @ para mencionar alguém

Você também pode mencionar um usuário que não está nesse menu suspenso, mas geralmente o preenchimento automático pode torná-lo mais rápido.

Depois de postar um comentário com uma menção de usuário, esse usuário será notificado. Isso significa que essa pode ser uma maneira realmente eficaz de atrair as pessoas para conversas, em vez de fazê-las enquetes. Muitas vezes, em Pull Requests no GitHub, as pessoas puxam outras pessoas em suas equipes ou em sua empresa para revisar um problema ou Pull Request.

Machine Translated by Google

Se alguém for mencionado em um pull request ou problema, ele será “inscrito” e continuará recebendo notificações sempre que alguma atividade ocorrer nele. Você também estará inscrito em algo se o abrir, se estiver assistindo ao repositório ou se comentar sobre algo. Se você não deseja mais receber notificações, há um botão “Cancelar inscrição” na página em que você pode clicar para parar de receber atualizações sobre ela.

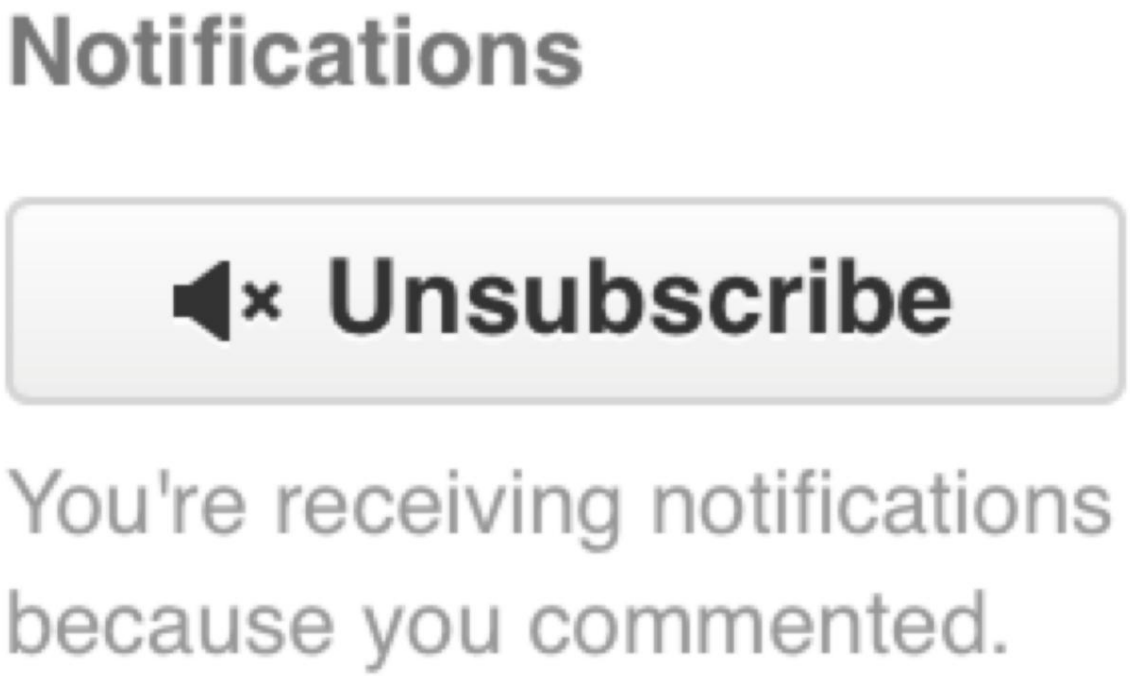


Figura 119. Cancelar assinatura de um problema ou pull request

A página de notificações

Quando mencionamos “notificações” aqui em relação ao GitHub, queremos dizer uma maneira específica que o GitHub tenta entrar em contato com você quando os eventos acontecem e existem algumas maneiras diferentes de configurá-los. Se você acessar a guia “Central de notificações” na página de configurações, poderá ver algumas das opções disponíveis.

Machine Translated by Google

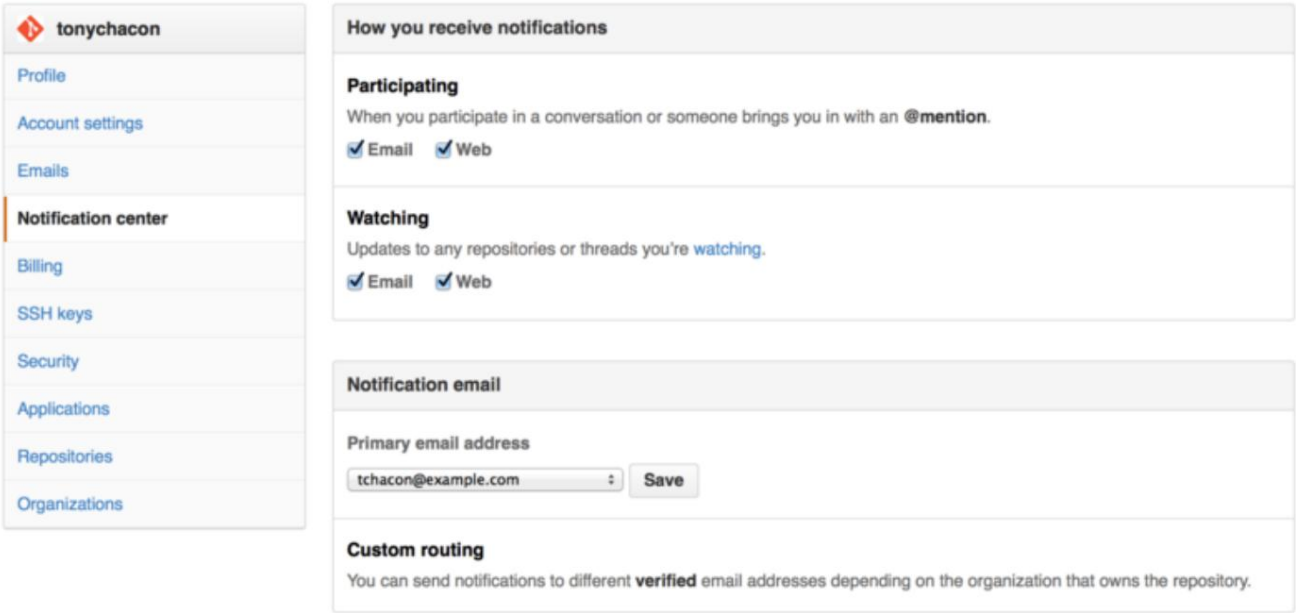


Figura 120. Opções do centro de notificação

As duas opções são receber notificações por “E-mail” e “Web” e você pode escolher uma, nenhuma ou ambas para quando você participa ativamente de coisas e para atividades nos repositórios que está assistindo.

Notificações da Web

As notificações da Web existem apenas no GitHub e você só pode verificá-las no GitHub. Se você tiver essa opção selecionada em suas preferências e uma notificação for acionada para você, você verá um pequeno ponto azul sobre o ícone de notificações na parte superior da tela, conforme visto na Central de notificações .

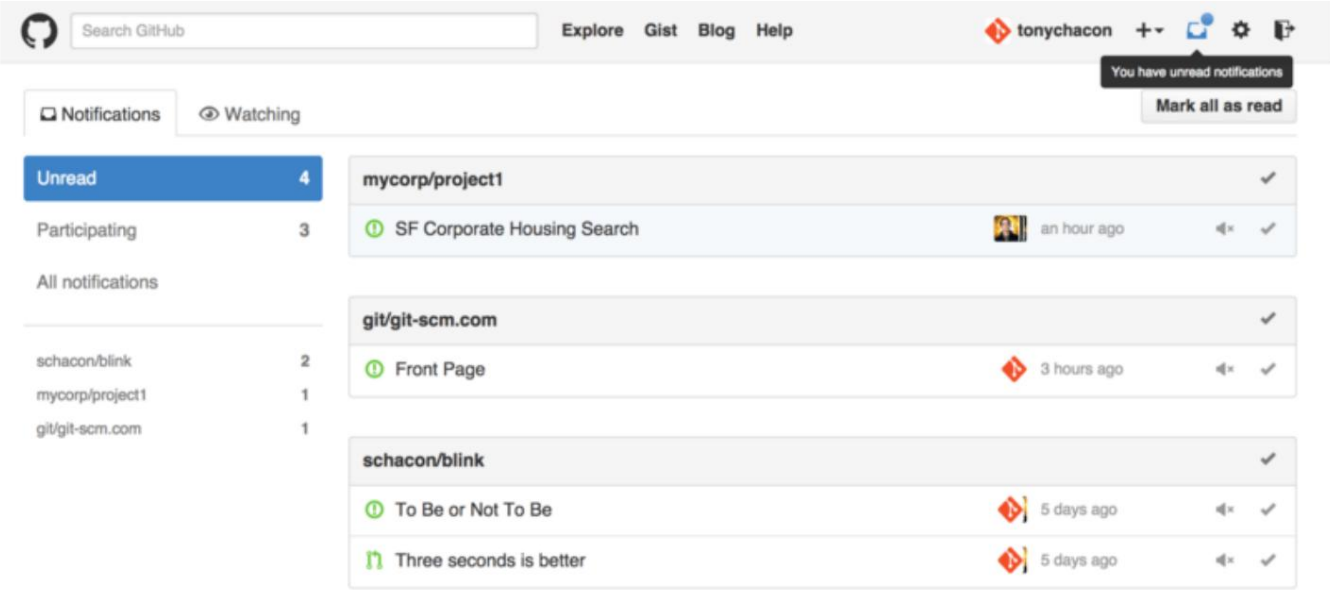


Figura 121. Central de notificações

Se você clicar nele, verá uma lista de todos os itens sobre os quais foi notificado, agrupados por projeto. Você pode filtrar as notificações de um projeto específico clicando em seu nome na barra lateral esquerda. Você também pode reconhecer a notificação clicando no ícone de marca de seleção ao lado de qualquer

Machine Translated by Google

notificação ou confirme *todas* as notificações em um projeto clicando na marca de seleção na parte superior do grupo. Há também um botão mudo ao lado de cada marca de seleção que você pode clicar para não receber mais notificações sobre esse item.

Todas essas ferramentas são muito úteis para lidar com um grande número de notificações. Muitos usuários avançados do GitHub simplesmente desativam totalmente as notificações por e-mail e gerenciam todas as suas notificações por meio dessa tela.

Notificações por e-mail

As notificações por e-mail são a outra maneira de lidar com as notificações por meio do GitHub. Se você tiver ativado, receberá e-mails para cada notificação. Vimos exemplos disso em comentários enviados como notificações por e-mail e notificação por e-mail de uma nova solicitação pull. Os e-mails também serão encadeados corretamente, o que é bom se você estiver usando um cliente de e-mail de encadeamento.

Há também uma boa quantidade de metadados incorporados nos cabeçalhos dos e-mails que o GitHub envia você, o que pode ser muito útil para configurar filtros e regras personalizados.

Por exemplo, se olharmos para os cabeçalhos de e-mail reais enviados para Tony no e-mail mostrado na notificação por e-mail de uma nova solicitação pull, veremos o seguinte entre as informações enviadas:

```
Para: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Espere mais para ver melhor o efeito de escurecimento (#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> Cancelar inscrição
da lista: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

Há algumas coisas interessantes aqui. Se você deseja destacar ou redirecionar e-mails para este projeto específico ou até mesmo Pull Request, as informações em Message-ID fornecem todos os dados no formato <user>/<project>/<type>/<id> . Se isso fosse um problema, por exemplo, o campo <type> seria “issues” em vez de “pull”.

Os campos List-Post e List-Unsubscribe significam que, se você tiver um cliente de e-mail que os entenda, você pode facilmente postar na lista ou “Cancelar inscrição” do tópico. Isso seria essencialmente o mesmo que clicar no botão “mudo” na versão da web da notificação ou “Cancelar inscrição” na própria página do problema ou da solicitação pull.

Também é importante observar que, se você tiver notificações por e-mail e pela Web ativadas e ler a versão por e-mail da notificação, a versão da Web também será marcada como lida se você tiver imagens permitidas em seu cliente de e-mail.

Arquivos Especiais

Existem alguns arquivos especiais que o GitHub notará se estiverem presentes em seu repositório.

Machine Translated by Google

Leia-me

O primeiro é o arquivo README , que pode ser de praticamente qualquer formato que o GitHub reconheça como prosa. Por exemplo, pode ser README, README.md, README.asciidoc, etc. Se o GitHub encontrar um arquivo README em sua fonte, ele o renderizará na página inicial do projeto.

Muitas equipes usam esse arquivo para manter todas as informações relevantes do projeto para alguém que pode ser novo no repositório ou projeto. Isso geralmente inclui coisas como:

- Para que serve o projeto
- Como configurar e instalar
- Um exemplo de como usá-lo ou executá-lo
- A licença sob a qual o projeto é oferecido
- Como contribuir para isso

Como o GitHub renderizará esse arquivo, você pode incorporar imagens ou links nele para maior facilidade de compreensão.

CONTRIBUINDO

O outro arquivo especial que o GitHub reconhece é o arquivo CONTRIBUTING . Se você tiver um arquivo chamado CONTRIBUTING com qualquer extensão de arquivo, o GitHub mostrará Abrindo uma solicitação pull quando existir um arquivo CONTRIBUTING quando alguém começar a abrir uma solicitação pull.

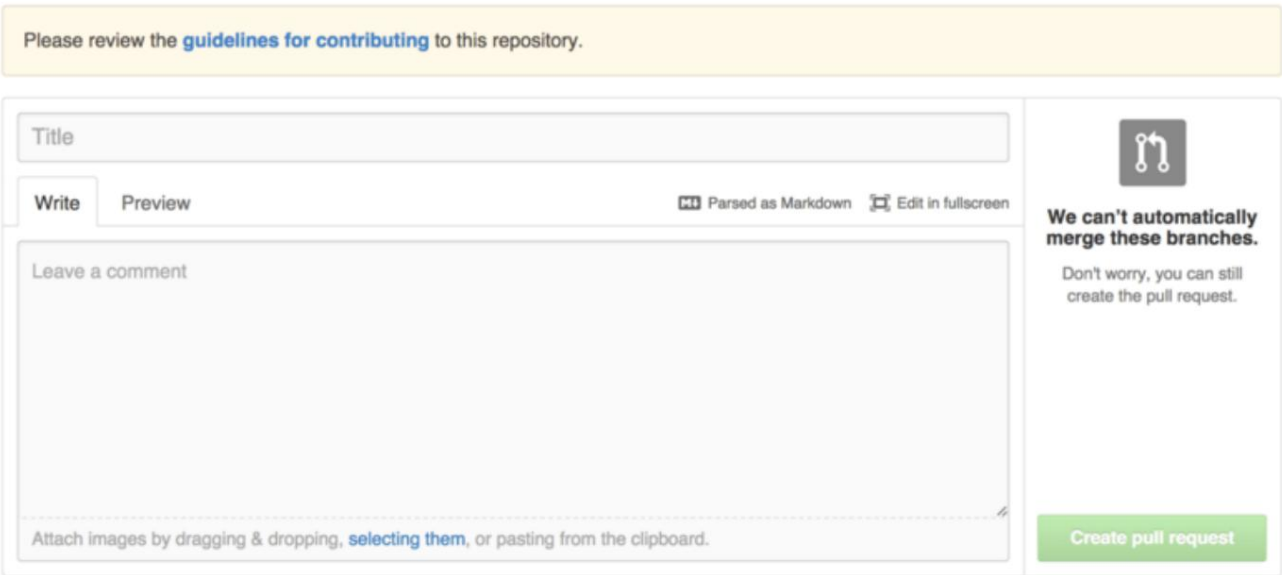


Figura 122. Abrindo uma solicitação pull quando existe um arquivo CONTRIBUTING

A ideia aqui é que você pode especificar coisas específicas que deseja ou não em uma solicitação pull enviada ao seu projeto. Dessa forma, as pessoas podem realmente ler as diretrizes antes de abrir o Pull Request.

Administração de Projetos

Geralmente não há muitas coisas administrativas que você pode fazer com um único projeto, mas há alguns itens que podem ser interessantes.

Machine Translated by Google

Alteração da ramificação padrão

Se você estiver usando uma ramificação diferente de “master” como sua ramificação padrão na qual deseja que as pessoas abram solicitações pull ou vejam por padrão, você pode alterar isso na página de configurações do seu repositório na guia “Opções”.

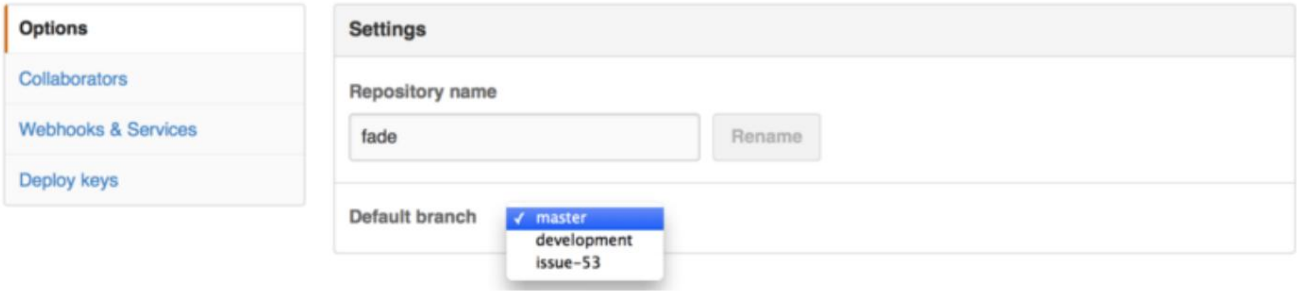


Figura 123. Alterar a ramificação padrão de um projeto

Basta alterar o branch padrão no menu suspenso e esse será o padrão para todas as principais operações a partir de então, incluindo qual branch é verificado por padrão quando alguém clona o repositório.

Transferindo um Projeto

Se você deseja transferir um projeto para outro usuário ou organização no GitHub, há uma opção “Transferir propriedade” na parte inferior da mesma guia “Opções” da página de configurações do repositório que permite fazer isso.

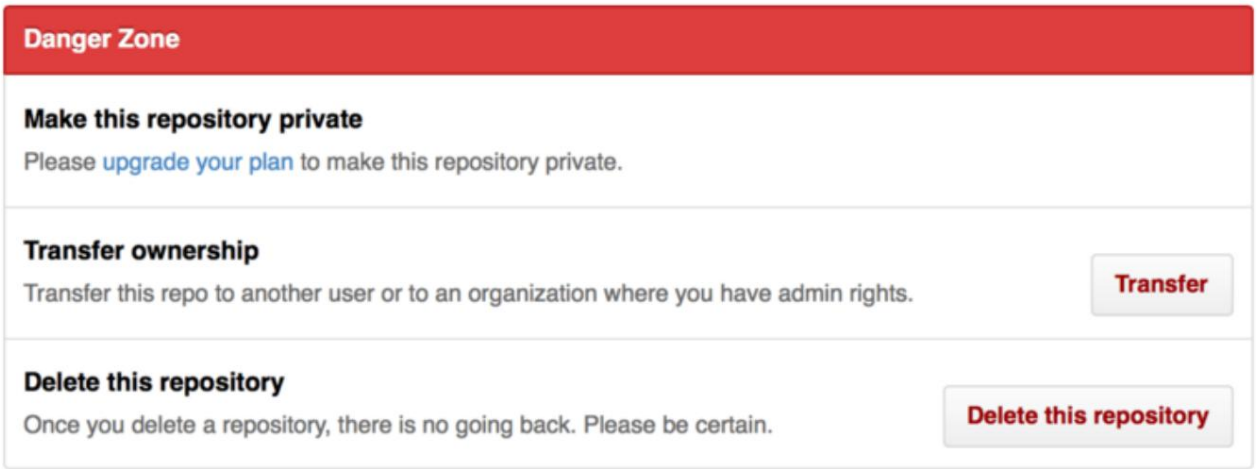


Figura 124. Transferir um projeto para outro usuário ou organização do GitHub

Isso é útil se você estiver abandonando um projeto e alguém quiser assumi-lo, ou se seu projeto estiver crescendo e quiser transferi-lo para uma organização.

Isso não apenas move o repositório junto com todos os observadores e estrelas para outro local, mas também configura um redirecionamento de sua URL para o novo local. Ele também redirecionará clones e buscas do Git, não apenas solicitações da web.

Machine Translated by Google

Gerenciando uma organização

Além das contas de usuário único, o GitHub possui o que chamamos de Organizações. Assim como as contas pessoais, as contas organizacionais têm um namespace onde todos os seus projetos existem, mas muitas outras coisas são diferentes. Essas contas representam um grupo de pessoas com propriedade compartilhada de projetos e existem muitas ferramentas para gerenciar subgrupos dessas pessoas. Normalmente, essas contas são usadas para grupos de código aberto (como “perl” ou “rails”) ou empresas (como “google” ou “twitter”).

Noções básicas de organização

Uma organização é muito fácil de criar; basta clicar no ícone “+” no canto superior direito de qualquer página do GitHub e selecionar “Nova organização” no menu.

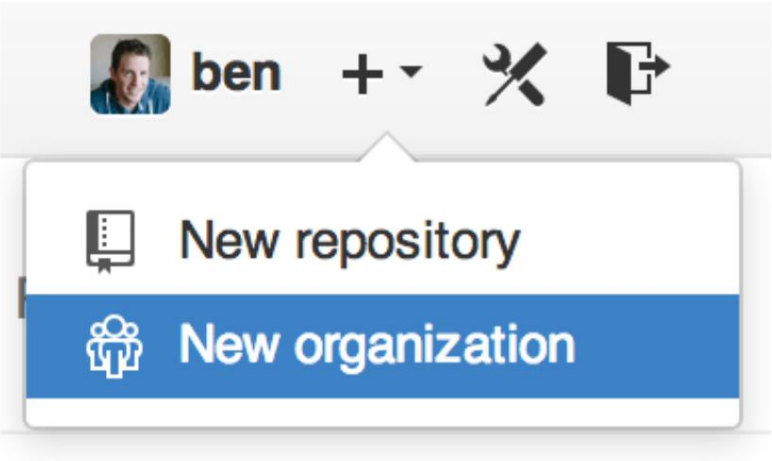


Figura 125. O item de menu “Nova organização”

Primeiro, você precisará nomear sua organização e fornecer um endereço de e-mail para um ponto de contato principal do grupo. Em seguida, você pode convidar outros usuários para serem coproprietários da conta, se desejar.

Siga estas etapas e em breve você será o proprietário de uma organização totalmente nova. Assim como as contas pessoais, as organizações são gratuitas se tudo o que você planeja armazenar for de código aberto.

Como proprietário de uma organização, ao bifurcar um repositório, você terá a opção de bifurcá-lo para o namespace de sua organização. Quando você cria novos repositórios, você pode criá-los em sua conta pessoal ou em qualquer uma das organizações das quais você é proprietário. Você também “observa” automaticamente qualquer novo repositório criado nessas organizações.

Assim como em Your Avatar, você pode enviar um avatar para sua organização para personalizá-lo um pouco. Além disso, assim como as contas pessoais, você tem uma página inicial para a organização que lista todos os seus repositórios e pode ser visualizada por outras pessoas.

Agora vamos cobrir algumas das coisas que são um pouco diferentes com uma conta organizacional.

Equipes

As organizações são associadas a pessoas individuais por meio de equipes, que são simplesmente um agrupamento de contas e repositórios de usuários individuais dentro da organização e que tipo de acesso essas pessoas têm nesses repositórios.

Por exemplo, digamos que sua empresa tenha três repositórios: frontend, backend e deployscripts. Você gostaria que seus desenvolvedores de HTML/CSS/JavaScript tivessem acesso ao front-end e talvez back-end, e seu pessoal de operações tivesse acesso ao back-end e scripts de implantação. As equipes facilitam isso, sem precisar gerenciar os colaboradores de cada repositório individual.

A página Organização mostra um painel simples de todos os repositórios, usuários e equipes que estão sob esta organização.

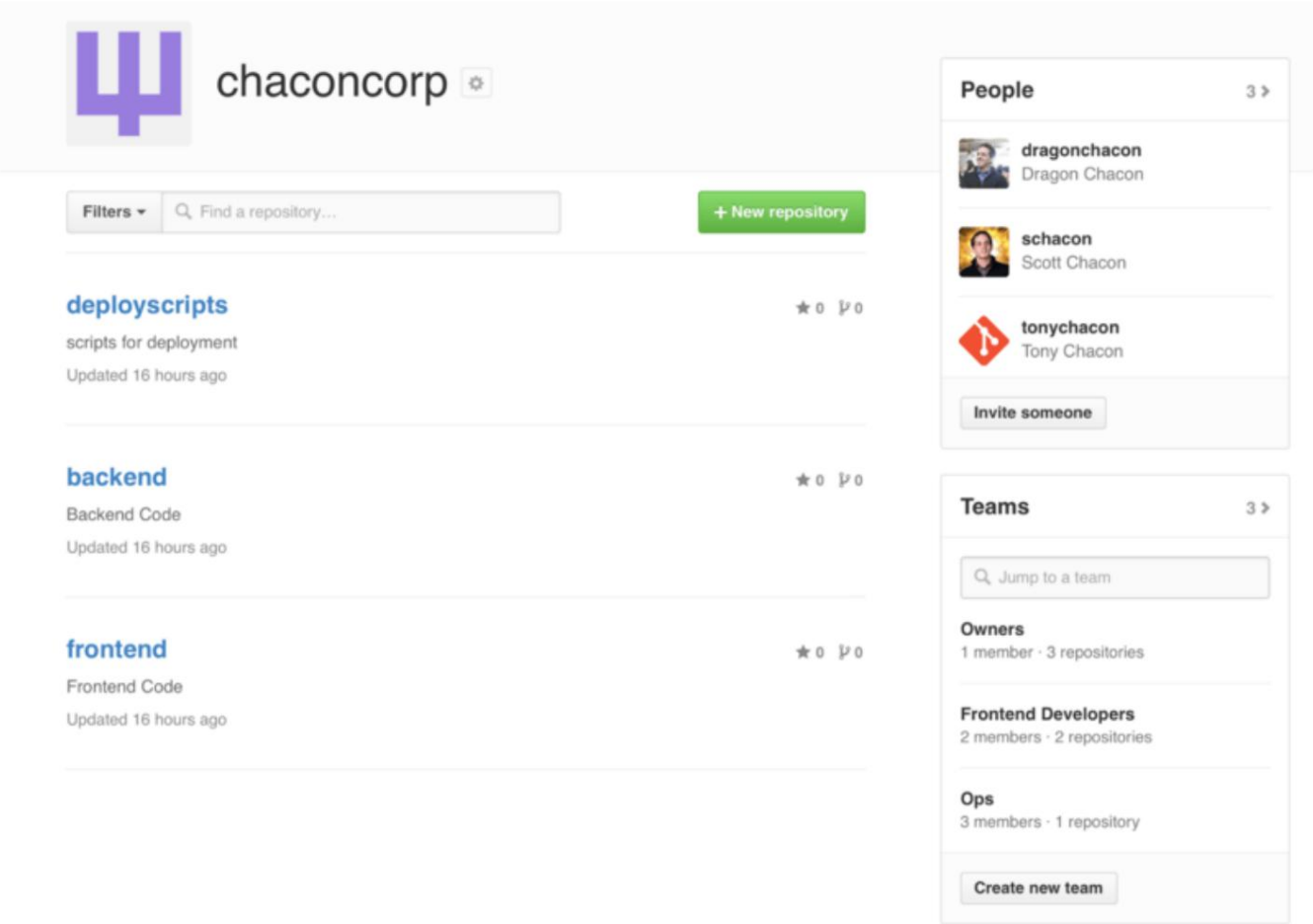


Figura 126. A página Organização

Para gerenciar suas equipes, você pode clicar na barra lateral de equipes no lado direito da página na página da organização. Isso o levará a uma página que você pode usar para adicionar membros à equipe, adicionar repositórios à equipe ou gerenciar as configurações e níveis de controle de acesso para a equipe. Cada equipe pode ter acesso somente leitura, leitura/gravação ou acesso administrativo aos repositórios. Você pode alterar esse nível clicando no botão "Configurações" na página da equipe.

Machine Translated by Google

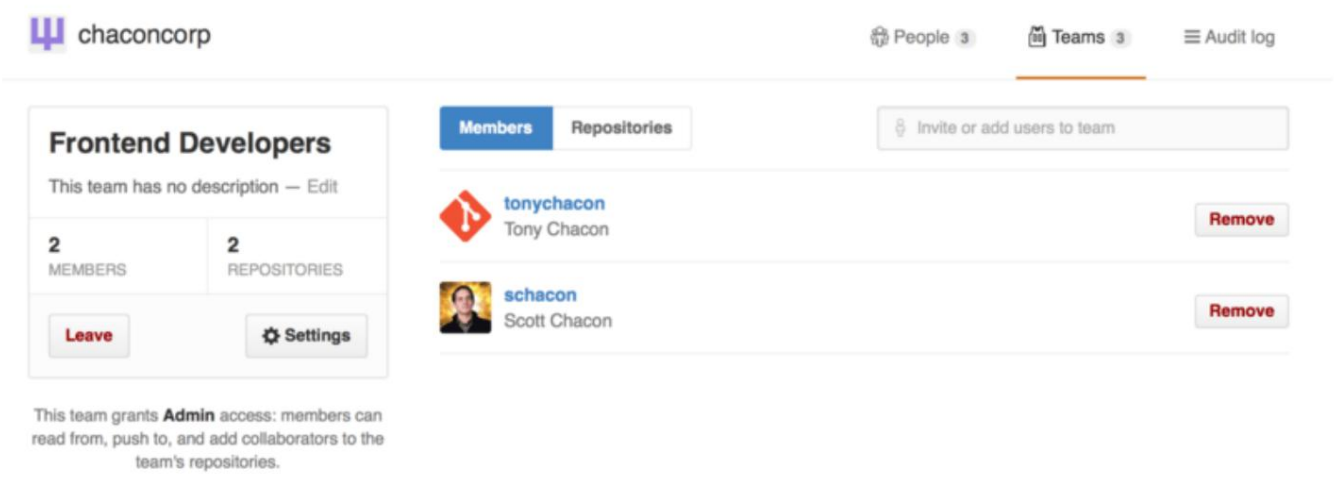


Figura 127. A página Equipe

Quando você convida alguém para uma equipe, eles recebem um e-mail informando que foram convidados.

Além disso, as @menções da equipe (como @acmecorp/frontend) funcionam da mesma forma que com usuários individuais, exceto que **todos os** membros da equipe são inscritos no encadeamento. Isso é útil se você deseja a atenção de alguém de uma equipe, mas não sabe exatamente a quem pedir.

Um usuário pode pertencer a qualquer número de equipes, portanto, não se limite apenas às equipes de controle de acesso. Equipes de interesse especial como ux, css ou refatoração são úteis para certos tipos de questões, e outras como jurídica e daltônica para um tipo totalmente diferente.

Registro de Auditoria

As organizações também fornecem aos proprietários acesso a todas as informações sobre o que aconteceu sob a organização. Você pode ir para a guia 'Audit Log' e ver quais eventos aconteceram no nível da organização, quem os realizou e onde no mundo eles foram realizados.

Machine Translated by Google

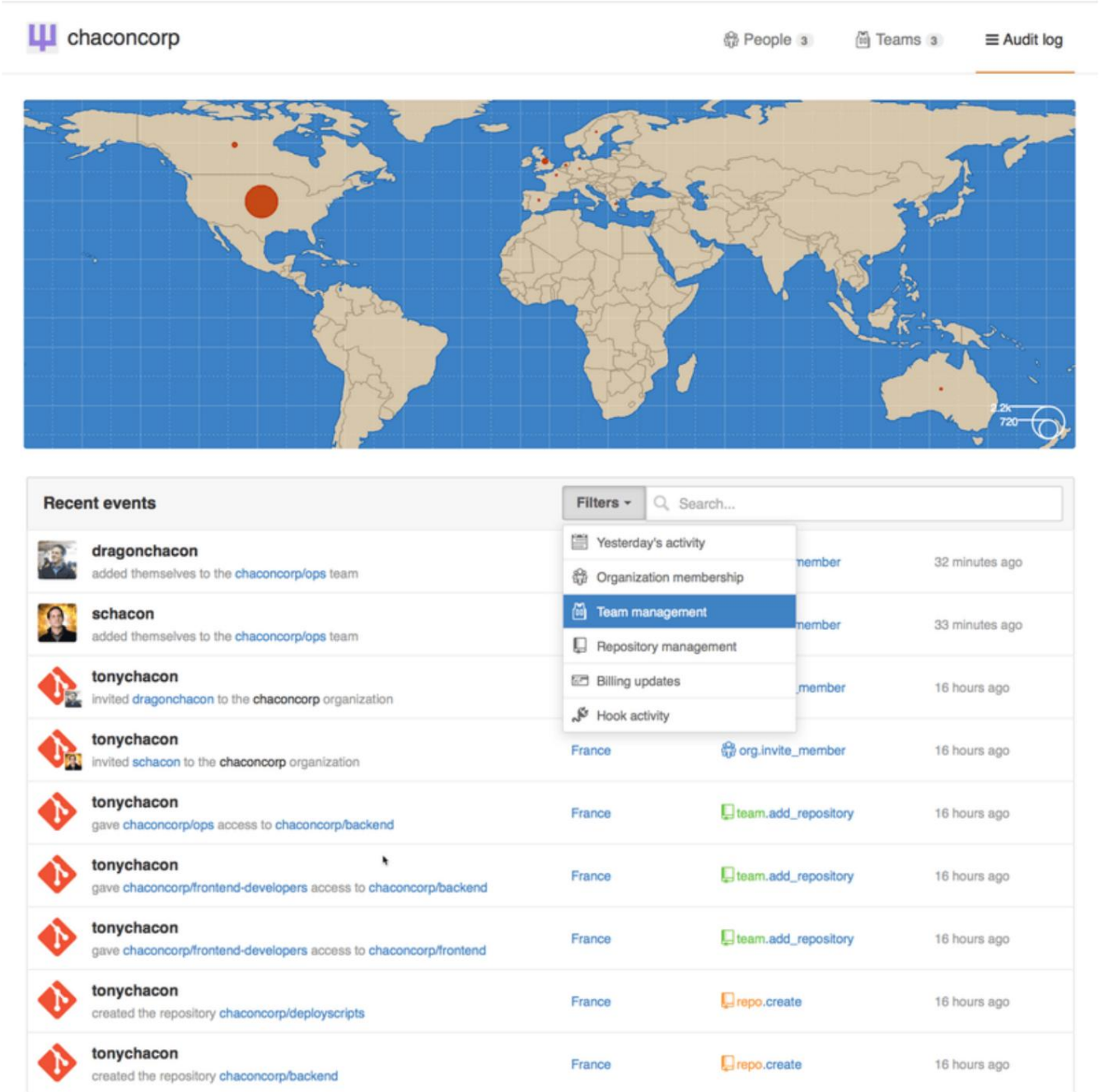


Figura 128. O registro de auditoria

Você também pode filtrar para tipos específicos de eventos, lugares específicos ou pessoas específicas.

Script GitHub

Agora, cobrimos todos os principais recursos e fluxos de trabalho do GitHub, mas qualquer grande grupo ou projeto terá personalizações que deseja fazer ou serviços externos que deseja integrar.

Felizmente para nós, o GitHub é bastante hackeável de várias maneiras. Nesta seção, abordaremos como usar o sistema de hooks do GitHub e sua API para fazer o GitHub funcionar como queremos.

Serviços e Ganchos

A seção Hooks and Services da administração do repositório GitHub é a maneira mais fácil de ter

Machine Translated by Google

O GitHub interage com sistemas externos.

Serviços

Primeiro vamos dar uma olhada em Serviços. Tanto as integrações de Hooks quanto de Serviços podem ser encontradas na seção Configurações do seu repositório, onde vimos anteriormente como adicionar Colaboradores e alterar a ramificação padrão do seu projeto. Na guia “Webhooks and Services”, você verá algo como a seção de configuração de Serviços e Hooks.

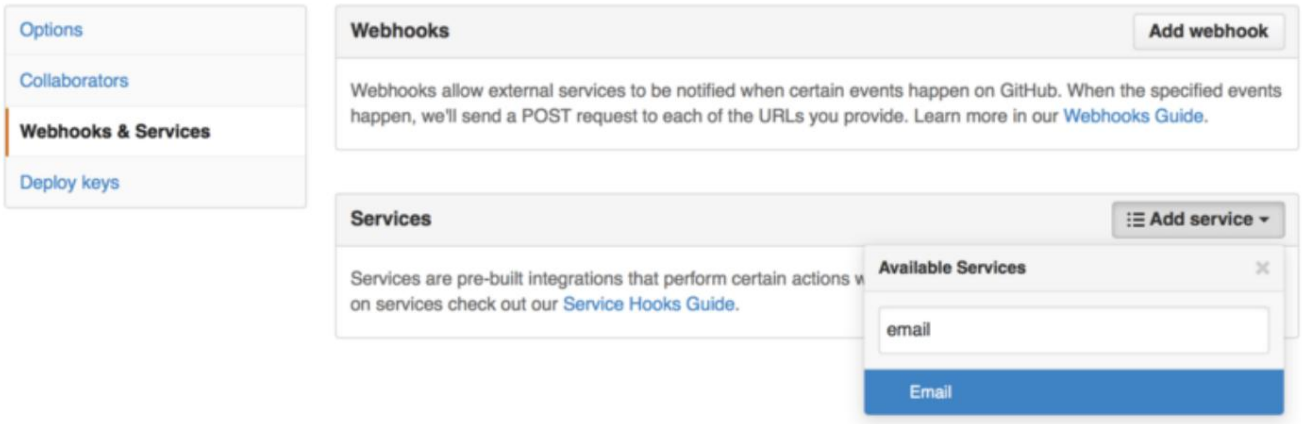


Figura 129. Seção de configuração de serviços e ganchos

Existem dezenas de serviços que você pode escolher, a maioria deles integrações em outros sistemas comerciais e de código aberto. A maioria deles é para serviços de Integração Contínua, rastreadores de bugs e problemas, sistemas de salas de bate-papo e sistemas de documentação. Vamos percorrer a configuração de um muito simples, o gancho de e-mail. Se você escolher "e-mail" no menu suspenso "Adicionar serviço", verá uma tela de configuração como Configuração do serviço de e-mail.

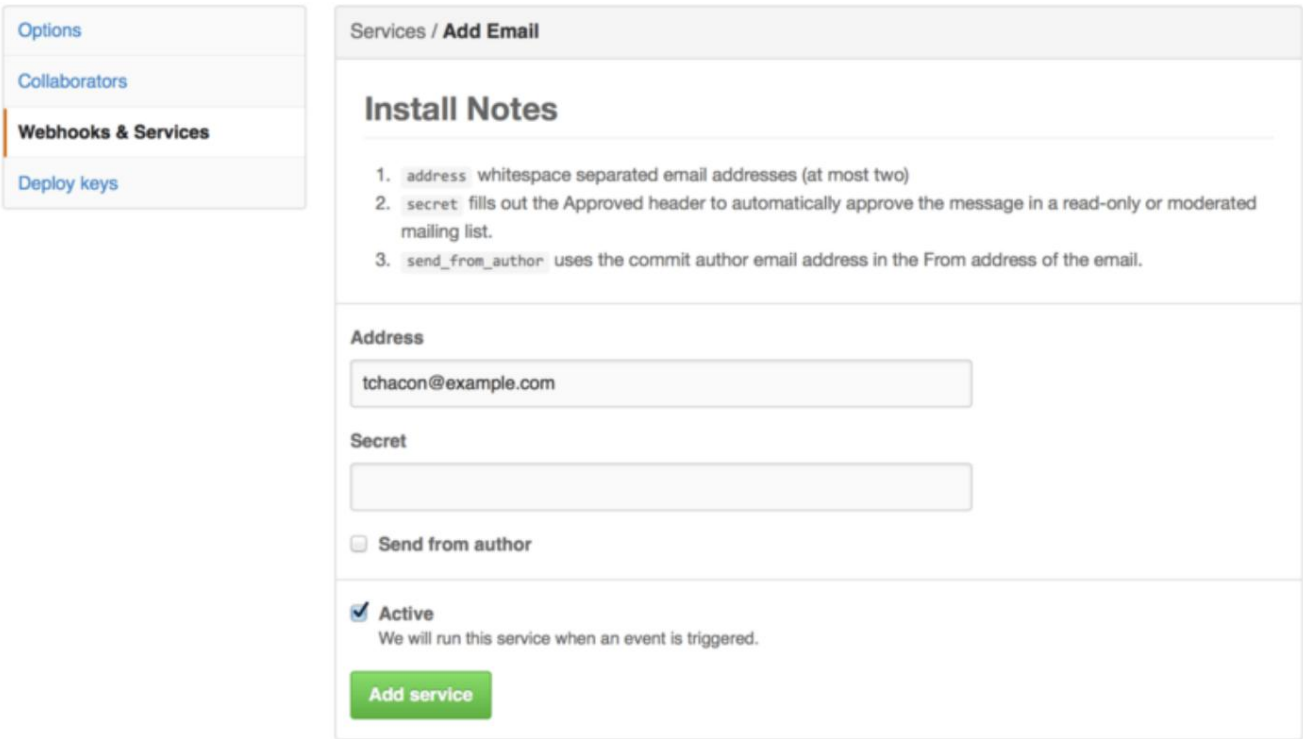


Figura 130. Configuração do serviço de e-mail

Machine Translated by Google

Nesse caso, se clicarmos no botão “Adicionar serviço”, o endereço de e-mail que especificamos receberá um e-mail toda vez que alguém acessar o repositório. Os serviços podem escutar vários tipos diferentes de eventos, mas a maioria apenas escuta eventos push e então faz algo com esses dados.

Se você estiver usando um sistema que gostaria de integrar com o GitHub, verifique aqui se há uma integração de serviço existente disponível. Por exemplo, se você estiver usando o Jenkins para executar testes em sua base de código, poderá habilitar a integração de serviço interna do Jenkins para iniciar uma execução de teste toda vez que alguém enviar para o seu repositório.

ganchos

Se você precisa de algo mais específico ou deseja integrar com um serviço ou site que não está incluído nesta lista, você pode usar o sistema de ganchos mais genérico. Os ganchos do repositório do GitHub são bem simples. Você especifica uma URL e o GitHub postará uma carga HTTP nessa URL em qualquer evento que você desejar.

Geralmente, a maneira como isso funciona é que você pode configurar um pequeno serviço da Web para ouvir uma carga de gancho do GitHub e, em seguida, fazer algo com os dados quando eles forem recebidos.

Para ativar um gancho, clique no botão “Adicionar webhook” na seção de configuração de serviços e ganchos.
Isso o levará a uma página que se parece com a configuração do Web hook.

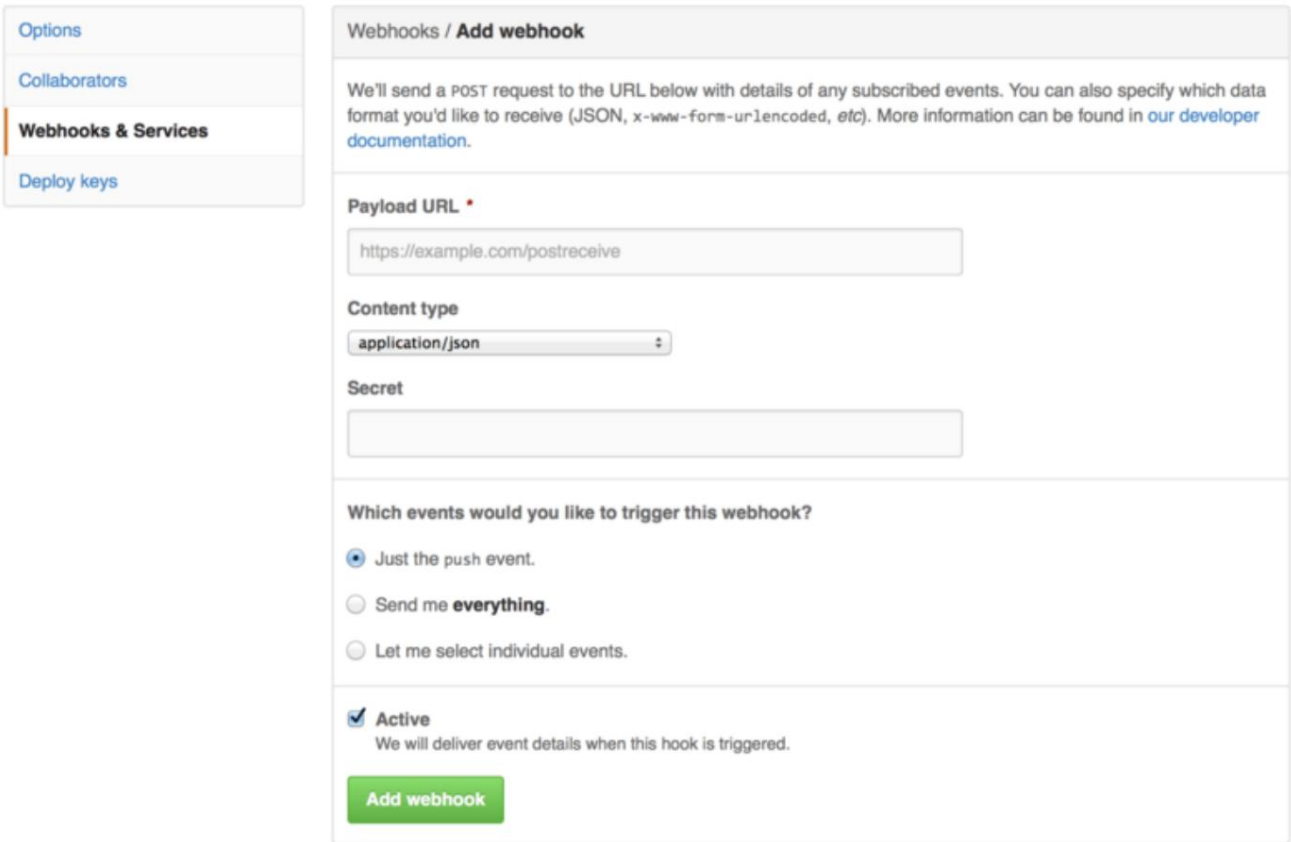


Figura 131. Configuração do web hook

A configuração de um web hook é bastante simples. Na maioria dos casos, basta inserir um URL e uma chave secreta e clicar em “Adicionar webhook”. Existem algumas opções para quais eventos você deseja que o GitHub envie uma carga para você — o padrão é obter apenas uma carga para o evento push , quando alguém envia um novo código para qualquer branch do seu repositório.

Machine Translated by Google

Vejamos um pequeno exemplo de um serviço da web que você pode configurar para lidar com um web hook. Usaremos o framework web Ruby Sinatra, pois ele é bastante conciso e você poderá ver facilmente o que estamos fazendo.

Digamos que queremos receber um e-mail se uma pessoa específica fizer um push para uma ramificação específica do nosso projeto, modificando um arquivo específico. Poderíamos facilmente fazer isso com um código como este:

```
exigir 'sinatra' exigir 'json'
exigir 'mail'


post '/payload' do push =
JSON.parse(request.body.read) # analisa o JSON

# coleta os dados que estamos procurando pusher =
push["pusher"]["name"] branch = push["ref"]


# obtém uma lista de todos os arquivos tocados files =
push["commits"].map do |commit| commit['adicionado'] +
commit['modificado'] + commit['removido'] end files = files.flatten.uniq



# verifique nossos critérios if pusher ==
'schacon' && branch == 'ref/heads/special-
        branch' && files.include?('special-file.txt')


Mail.deliver do
                        'tchacon@example.com'
        para            'tchacon@example.com' assunto
        'Scott mudou o arquivo' corpo "ALARME" fim


fim
fim
```

Aqui estamos pegando a carga útil JSON que o GitHub nos entrega e procurando quem a enviou, para qual ramificação eles enviaram e quais arquivos foram tocados em todos os commits que foram enviados. Em seguida, verificamos isso de acordo com nossos critérios e enviamos um e-mail se corresponder.

Para desenvolver e testar algo assim, você tem um bom console de desenvolvedor na mesma tela em que configurou a conexão. Você pode ver as últimas entregas que o GitHub tentou fazer para esse webhook. Para cada gancho, você pode pesquisar quando foi entregue, se foi bem-sucedido e o corpo e os cabeçalhos da solicitação e da resposta. Isso torna incrivelmente fácil testar e depurar seus ganchos.

Machine Translated by Google

Recent Deliveries

⚠	4aeae280-4e38-11e4-9bac-c130e992644b	2014-10-07 17:40:41
✔	aff20880-4e37-11e4-9089-35319435e08b	2014-10-07 17:36:21
✔	90f37680-4e37-11e4-9508-227d13b2ccfc	2014-10-07 17:35:29

Request | Response 200 | Completed in 0.61 seconds. | Redeliver

Headers

```
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
```

Payload


```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bffaf827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c
```


Figura 132. Informações de depuração do web hook

A outra grande característica disso é que você pode reenviar qualquer uma das cargas para testar seu serviço facilmente.

Para obter mais informações sobre como escrever webhooks e todos os diferentes tipos de eventos que você pode ouvir, acesse a documentação do GitHub Developer em https://developer.github.com/webhooks/.

A API do GitHub

Os serviços e ganchos fornecem uma maneira de receber notificações push sobre eventos que acontecem em seus repositórios, mas e se você precisar de mais informações sobre esses eventos? E se você precisar automatizar algo como adicionar colaboradores ou rotular problemas?

É aqui que a API do GitHub é útil. O GitHub tem toneladas de endpoints de API para fazer quase

Machine Translated by Google

tudo o que você pode fazer no site de forma automatizada. Nesta seção, aprenderemos como autenticar e conectar-se à API, como comentar um problema e como alterar o status de uma solicitação pull por meio da API.

Uso básico

A coisa mais básica que você pode fazer é uma solicitação GET simples em um endpoint que não requer autenticação. Isso pode ser um usuário ou informações somente leitura em um projeto de código aberto. Por exemplo, se quisermos saber mais sobre um usuário chamado “schacon”, podemos executar algo assim:

```
$ 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" }
```

Existem vários endpoints como esse para obter informações sobre organizações, projetos, problemas, confirmações — praticamente qualquer coisa que você possa ver publicamente no GitHub. Você pode até usar a API para renderizar Markdown arbitrário ou encontrar um modelo .gitignore .

```
$ curl https://api.github.com/gitignore/templates/Java { "name": "Java",
"source": "*.class



# Ferramentas móveis para Java
 (J2ME) .mtj.tmp/

# Arquivos de pacote
# *.jar *.war

*.orelha


# logs de falha da máquina virtual, consulte
https://www.java.com/en/download/help/error_hotspot.xml hs_err_pid*
"

}
```

Machine Translated by Google

Comentando sobre um problema

No entanto, se você quiser realizar uma ação no site, como comentar sobre um problema ou pull request, ou se quiser visualizar ou interagir com conteúdo privado, precisará se autenticar.

Existem várias maneiras de autenticar. Você pode usar autenticação básica apenas com seu nome de usuário e senha, mas geralmente é uma ideia melhor usar um token de acesso pessoal. Você pode gerar isso na guia "Aplicativos" da sua página de configurações.

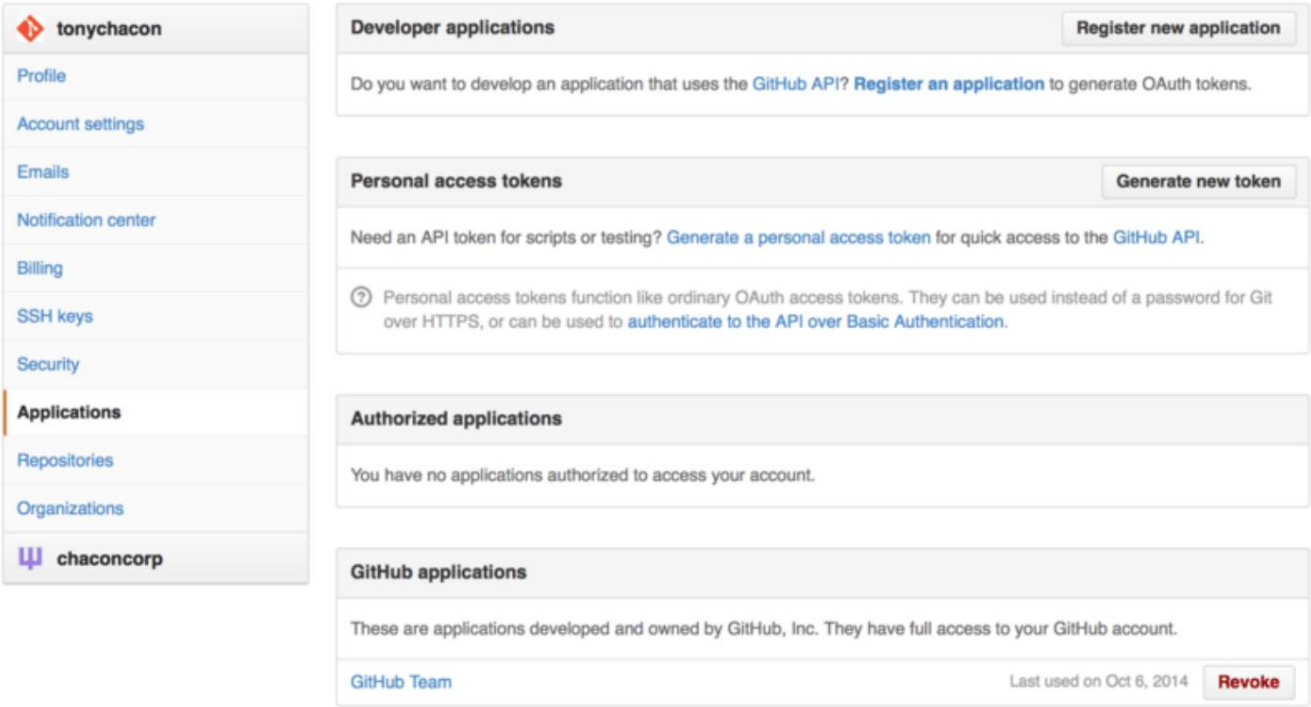


Figura 133. Gere seu token de acesso na guia "Aplicativos" da página de configurações

Ele perguntará quais escopos você deseja para este token e uma descrição. Certifique-se de usar uma boa descrição para se sentir confortável em remover o token quando seu script ou aplicativo não for mais usado.

O GitHub mostrará o token apenas uma vez, portanto, certifique-se de copiá-lo. Agora você pode usar isso para autenticar em seu script em vez de usar um nome de usuário e senha. Isso é bom porque você pode limitar o escopo do que deseja fazer e o token é revogável.

Isso também tem a vantagem adicional de aumentar seu limite de taxa. Sem autenticação, você ficará limitado a 60 solicitações por hora. Se você autenticar, poderá fazer até 5.000 solicitações por hora.

Então, vamos usá-lo para fazer um comentário sobre um de nossos problemas. Digamos que queremos deixar um comentário sobre um problema específico, o número 6. Para fazer isso, temos que fazer uma solicitação HTTP POST para repos/<user>/<repo>/issues/<num>/comments com o token que acabamos de gerar como um cabeçalho de autorização.

```
$ curl -H "Tipo de conteúdo: aplicativo/json" \
        -H "Autorização: token TOKEN" \ --data '{"body":"Um novo comentário, :
        +1:"}' \ https://api.github.com/repos/schacon/blink/issues/6 /comentários
```

Machine Translated by Google

{ "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": "Um novo comentário, :+1:" }

Agora, se você for para esse problema, poderá ver o comentário que acabamos de postar com sucesso como em Um comentário postado da API do GitHub.

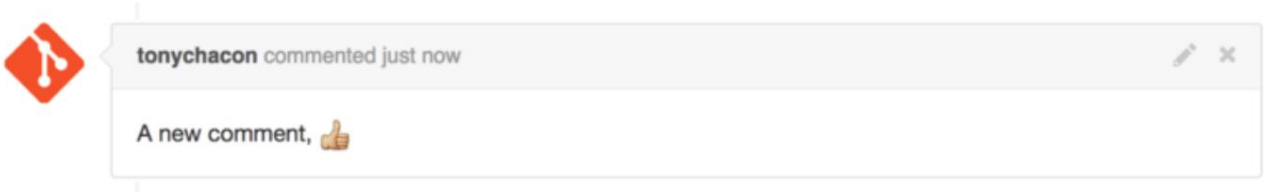


Figura 134. Um comentário postado da API do GitHub

Você pode usar a API para fazer praticamente qualquer coisa que você pode fazer no site — criar e definir marcos, atribuir pessoas a problemas e solicitações pull, criar e alterar rótulos, acessar dados de commit, criar novos commits e branches, abrir, fechar ou mesclar Pull Requests, criando e editando equipes, comentando linhas de código em um Pull Request, pesquisando no site e assim por diante.

Alterando o status de uma solicitação pull

Há um exemplo final que veremos, pois é muito útil se você estiver trabalhando com solicitações pull.
Cada commit pode ter um ou mais status associados a ele e existe uma API para adicionar e consultar esse status.

A maioria dos serviços de Integração Contínua e de teste usa essa API para reagir a pushes, testando o código que foi enviado e, em seguida, reporta se esse commit passou em todos os testes. Você também pode usar isso para verificar se a mensagem do commit está formatada corretamente, se o remetente seguiu todas as suas diretrizes de contribuição, se o commit foi assinado de forma válida — uma série de coisas.

Digamos que você configure um webhook em seu repositório que atinge um pequeno serviço da web que verifica uma string Signed-off-by na mensagem de confirmação.

```
exigir 'httparty'
exigir 'sinatra'
exigir 'json'
```

```
post '/payload' do push =

JSON.parse(request.body.read) # analisa o JSON repo_name = push['repository']['full_name']



# examina cada mensagem de commit push["commits"].each do |commit|



# procure por uma string Desconectado por if /Desconectado por/.match commit['message'] state

= 'success' description = 'Desconectado com sucesso!' outro estado = 'falha'



            descrição = 'Nenhuma aprovação
encontrada.' fim


# postar status no GitHub sha = commit["id"] status_url = "https://

api.github.com/repos/#{repo_name}/statuses/#{sha}"



status = { "state" => estado,

                "description" => descrição, "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']}" }



) fim fim
```

Espero que isso seja bastante simples de seguir. Neste manipulador de gancho da web, examinamos cada confirmação que acabou de ser enviada, procuramos a string 'Signed-off-by' na mensagem de confirmação e, finalmente, POSTamos via HTTP para /repos/<user>/<repo> / status/<commit_sha> terminal da API com o status.

Nesse caso, você pode enviar um estado ('sucesso', 'falha', 'erro'), uma descrição do que aconteceu, um URL de destino ao qual o usuário pode acessar para obter mais informações e um “contexto” caso haja vários status para um único commit. Por exemplo, um serviço de teste pode fornecer um status e um serviço de validação como este também pode fornecer um status — o campo “contexto” é como eles são diferenciados.

Se alguém abrir uma nova solicitação pull no GitHub e esse gancho estiver configurado, você poderá ver algo como Commit status por meio da API.

Machine Translated by Google

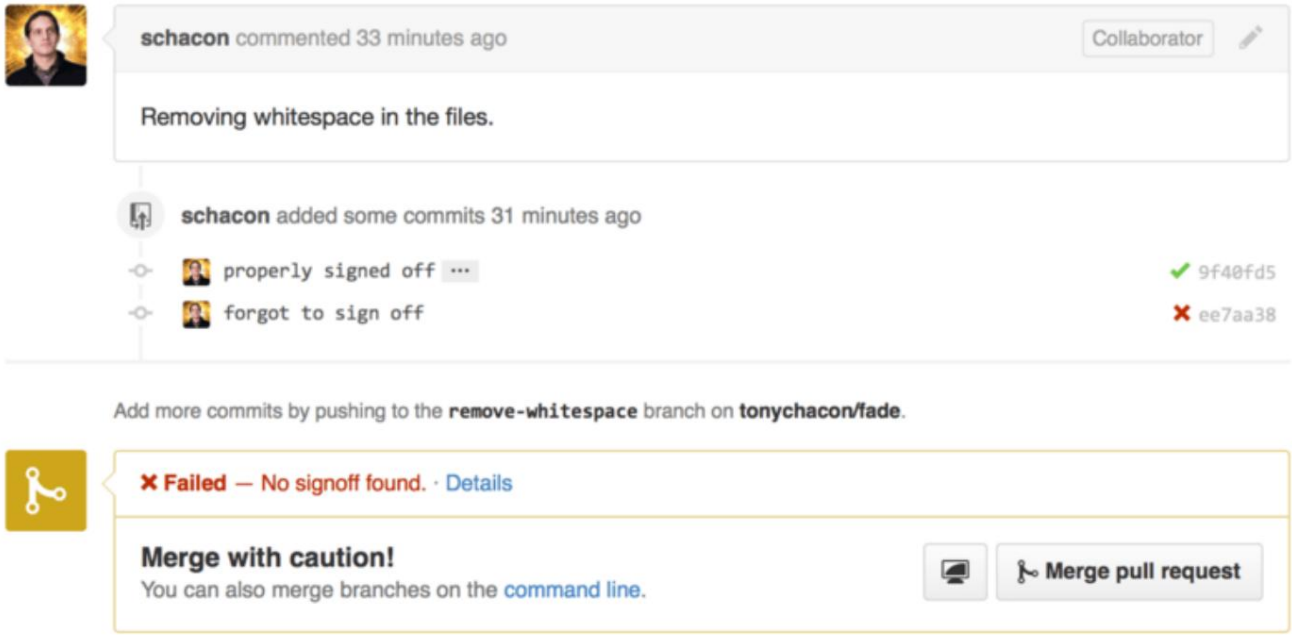


Figura 135. Status de confirmação por meio da API

Agora você pode ver uma pequena marca de seleção verde ao lado do commit que tem uma string “Assinado por” na mensagem e uma cruz vermelha naquela em que o autor esqueceu de assinar. Você também pode ver que o Pull Request assume o status do último commit na ramificação e avisa se for uma falha.
Isso é realmente útil se você estiver usando essa API para resultados de teste, para não mesclar acidentalmente algo em que o último commit está falhando nos testes.

OctoKitGenericName

Embora tenhamos feito quase tudo por meio de curl e solicitações HTTP simples nesses exemplos, existem várias bibliotecas de código aberto que disponibilizam essa API de maneira mais idiomática.
No momento da redação deste artigo, as linguagens com suporte incluem Go, Objective-C, Ruby e .NET. Confira https://github.com/octokit para obter mais informações sobre eles, pois eles lidam com grande parte do HTTP para você.

Esperamos que essas ferramentas possam ajudá-lo a personalizar e modificar o GitHub para funcionar melhor em seus fluxos de trabalho específicos. Para obter a documentação completa sobre toda a API, bem como guias para tarefas comuns, confira https://developer.github.com.

Resumo

Agora você é um usuário do GitHub. Você sabe como criar uma conta, gerenciar uma organização, criar e enviar para repositórios, contribuir com projetos de outras pessoas e aceitar contribuições de outras pessoas. No próximo capítulo, você aprenderá ferramentas e dicas mais poderosas para lidar com situações complexas, que farão de você um verdadeiro mestre do Git.

Machine Translated by Google

Git Tools

Até agora, você aprendeu a maioria dos comandos e fluxos de trabalho do dia-a-dia necessários para gerenciar ou manter um repositório Git para seu controle de código-fonte. Você realizou as tarefas básicas de rastreamento e confirmação de arquivos e aproveitou o poder da área de preparação e da ramificação e mesclagem de tópicos leves.

Agora você explorará uma série de coisas muito poderosas que o Git pode fazer que você pode não necessariamente usar no dia-a-dia, mas que você pode precisar em algum momento.

Seleção de revisão

O Git permite que você se refira a um único commit, conjunto de commits ou intervalo de commits de várias maneiras. Eles não são necessariamente óbvios, mas é útil saber.

Revisões únicas

Obviamente, você pode se referir a qualquer commit único por seu hash SHA-1 completo de 40 caracteres, mas também existem maneiras mais amigáveis de se referir a commits. Esta seção descreve as várias maneiras pelas quais você pode se referir a qualquer confirmação.

SHA-1 curto

O Git é inteligente o suficiente para descobrir a qual commit você está se referindo se você fornecer os primeiros caracteres do hash SHA-1, contanto que esse hash parcial tenha pelo menos quatro caracteres e não seja ambíguo; ou seja, nenhum outro objeto no banco de dados de objetos pode ter um hash que comece com o mesmo prefixo.

Por exemplo, para examinar um commit específico ao qual você sabe que adicionou determinada funcionalidade, você pode primeiro executar o comando git log para localizar o commit:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Autor: Scott Chacon <schacon@gmail.com> Data: Sex Jan 2
18:32:33 2009 -0800

Corrigir manipulação de refs, adicionar gc auto, atualizar testes

cometer d921970aadf03b3cf0e71becdaab3147ba71cdef
Mesclar: 1c002dd... 35cfb2b...
Autor: Scott Chacon <schacon@gmail.com> Data:
Qui 11 de dezembro 15:08:43 2008 -0800

Mesclar commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Autor: Scott Chacon <schacon@gmail.com> Data: qui 11 de
dezembro 14:58:32 2008 -0800
```

Machine Translated by Google

```
Adicione alguma culpa e mescle coisas
```

Nesse caso, digamos que você esteja interessado no commit cujo hash começa com 1c002dd…. Você pode inspecionar esse commit com qualquer uma das seguintes variações de git show (supondo que as versões mais curtas não sejam ambíguas):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f $ git show 1c002d
```

O Git pode descobrir uma abreviação curta e exclusiva para seus valores SHA-1. Se você passar --abbrev-commit para o comando git log , a saída usará valores mais curtos, mas os manterá exclusivos; o padrão é usar sete caracteres, mas os torna mais longos, se necessário, para manter o SHA-1 inequívoco:

```
$ git log --abbrev-commit --pretty=oneline ca82a6d
Altere o número da versão 085bb3b Remova o
código de teste desnecessário a11bef0
Confirmação inicial
```

Geralmente, oito a dez caracteres são mais do que suficientes para serem únicos dentro de um projeto. Por exemplo, em fevereiro de 2019, o kernel do Linux (que é um projeto bastante considerável) tinha mais de 875.000 confirmações e quase sete milhões de objetos em seu banco de dados de objetos, sem dois objetos cujos SHA-1s fossem idênticos nos primeiros 12 caracteres.

ÿ

UMA BREVE NOTA SOBRE SHA-1

Muitas pessoas ficam preocupadas em algum momento que, por acaso, terão dois objetos distintos em seu repositório com hash para o mesmo valor SHA-1. O que então?

Se acontecer de você confirmar um objeto com hashes para o mesmo valor SHA-1 como um objeto *diferente* anterior em seu repositório, o Git verá o objeto anterior já em seu banco de dados Git, assumirá que já foi escrito e simplesmente o reutilizará. Se você tentar verificar esse objeto novamente em algum momento, sempre obterá os dados do primeiro objeto.

No entanto, você deve estar ciente de como esse cenário é ridiculamente improvável. O resumo SHA-1 tem 20 bytes ou 160 bits. O número de objetos hash aleatórios necessários para garantir uma probabilidade de 50% de uma única colisão é cerca de 280 (a fórmula para determinar a probabilidade de colisão é p = (n(n-1)/2) * (1/2^160)). 280 é 1,2 x 1024 ou 1 milhão de bilhões de bilhões. Isso é 1.200 vezes o número de grãos de areia na Terra.

Aqui está um exemplo para lhe dar uma ideia do que seria necessário para obter uma colisão SHA-1. Se todos os 6,5 bilhões de humanos na Terra estivessem programando, e a cada segundo, cada um estivesse produzindo um código equivalente a todo o kernel do Linux

Machine Translated by Google

> history (6,5 milhões de objetos Git) e colocá-lo em um enorme repositório Git, levaria cerca de 2 anos até que esse repositório contivesse objetos suficientes para ter uma probabilidade de 50% de uma única colisão de objeto SHA-1. Assim, uma colisão SHA-1 orgânica é menos provável do que cada membro de sua equipe de programação ser atacado e morto por lobos em incidentes não relacionados na mesma noite.
>
> Se você dedicar vários milhares de dólares em poder de computação a ele, é possível sintetizar dois arquivos com o mesmo hash, conforme comprovado em https://shattered.io/ em fevereiro de 2017. O Git está adotando o SHA256 como o algoritmo de hash padrão, que é muito mais resiliente a ataques de colisão e possui código para ajudar a mitigar esse ataque (embora não possa eliminá-lo completamente).

Referências de filiais

Uma maneira direta de se referir a um commit específico é se for o commit na ponta de uma ramificação; nesse caso, você pode simplesmente usar o nome do branch em qualquer comando do Git que espera uma referência a um commit. Por exemplo, se você deseja examinar o último objeto de confirmação em uma ramificação, os seguintes comandos são equivalentes, assumindo que a ramificação topic1 aponta para a confirmação ca82a6d…:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Se você quiser ver para qual SHA-1 específico uma ramificação aponta, ou se quiser ver o que qualquer um desses exemplos se resume em termos de SHA-1s, você pode usar uma ferramenta de encanamento do Git chamada rev-parse . Você pode consultar o Git Internals para obter mais informações sobre ferramentas de canalização; basicamente, rev-parse existe para operações de nível inferior e não é projetado para ser usado em operações do dia-a-dia. No entanto, pode ser útil às vezes quando você precisa ver o que realmente está acontecendo. Aqui você pode executar rev-parse em sua ramificação.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

Abreviaturas RefLog

Uma das coisas que o Git faz em segundo plano enquanto você está trabalhando é manter um “reflog” — um registro de onde seu HEAD e referências de ramificação estiveram nos últimos meses.

Você pode ver seu reflog usando git reflog:

```
$ git reflog
734713b HEAD@{0}: commit: Corrige tratamento de refs, adiciona gc auto, atualiza
testes d921970 HEAD@{1}: mescla phedders/rdocs: Mescla feita pela estratégia 'recursiva'.
1c002dd HEAD@{2}: commit: Adicionar alguma culpa e mesclar coisas 1c36188 HEAD@{3}:
rebase -i (squash): atualizando HEAD 95df984 HEAD@{4}: commit: # Esta é uma combinação
de dois commits.
```

Machine Translated by Google

```
1c36188 HEAD@{5}: rebase -i (squash): atualizando o HEAD
7e05da5 HEAD@{6}: rebase -i (escolha): atualizando o HEAD
```

Toda vez que sua dica de ramificação é atualizada por qualquer motivo, o Git armazena essas informações para você nesse histórico temporário. Você também pode usar seus dados de reflog para se referir a commits mais antigos. Por exemplo, se você quiser ver o quinto valor anterior do HEAD do seu repositório, você pode usar a referência @{5} que você vê na saída do reflog:

```
$ git show HEAD@{5}
```

Você também pode usar essa sintaxe para ver onde uma ramificação estava há algum tempo atrás. Por exemplo, para ver onde seu branch master estava ontem, você pode digitar:

```
$ git show master@{ontem}
```

Isso mostraria a você onde estava a ponta do seu ramo mestre ontem. Esta técnica só funciona para dados que ainda estão em seu reflog, então você não pode usá-la para procurar commits com mais de alguns meses.

Para ver as informações do reflog formatadas como a saída git log , você pode executar git log -g:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Mensagem de reflog: commit: Corrigir manipulação de refs, adicionar gc auto, atualizar
testes Autor: Scott Chacon <schacon@gmail.com> Data: Sex Jan 2 18:32:33 2009 -0800


Corrigir manipulação de refs, adicionar gc auto, atualizar testes

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Mesclar feito por recursive.
Autor: Scott Chacon <schacon@gmail.com> Data:
Qui 11 de dezembro 15:08:43 2008 -0800


Mesclar commit 'phedders/rdocs'
```

É importante observar que as informações do reflog são estritamente locais — é um registro apenas do que *você* fez em *seu* repositório. As referências não serão as mesmas na cópia do repositório de outra pessoa; Além disso, logo após você inicialmente clonar um repositório, você terá um reflog vazio, pois nenhuma atividade ocorreu ainda em seu repositório. A execução de git show HEAD@{2.months.ago} mostrará o commit correspondente apenas se você clonou o projeto há pelo menos dois meses — se você o clonou mais recentemente, verá apenas seu primeiro commit local.

ÿ

Pense no reflog como a versão do Git do histórico do shell

Se você tem experiência em UNIX ou Linux, pode pensar no reflog como o Git's

Machine Translated by Google

versão do shell history, que enfatiza que o que está lá é claramente relevante apenas para você e sua “sessão”, e não tem nada a ver com qualquer outra pessoa que possa estar trabalhando na mesma máquina.

Escapando chaves no PowerShell

Ao usar o PowerShell, colchetes como { e } são caracteres especiais e devem ter escape. Você pode escapá-los com um backtick ou colocar a referência do commit entre aspas:

```
$ git show HEAD@{0} # NÃO funcionará $ git
show HEAD@`{0`} # OK $ git show "HEAD@{0}"
# OK
```

Referências de Ascendência

A outra maneira principal de especificar um commit é por meio de seu ancestral. Se você colocar um ^ (sinal circunflexo) no final de uma referência, o Git resolverá isso como o pai desse commit. Suponha que você veja o histórico do seu projeto:

```
$ git log --pretty=format:'%h %s' --graph * 734713b
Correção do tratamento de refs, adição de gc auto, atualização de
testes * d921970 Mesclar commit 'phedders/rdocs' |\ | * 35cfb2b
 Algumas mudanças rdoc * | 1c002dd Adicionar alguma culpa e
 mesclar coisas |/ * 1c36188 Ignorar *.gem * 9b29157 Adicionar
open3_detach à lista de arquivos gemspec
```

Então, você pode ver o commit anterior especificando HEAD^, que significa “o pai de HEAD”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef

Mesclar: 1c002dd... 35cfb2b...
Autor: Scott Chacon <schacon@gmail.com> Data:
Qui 11 de dezembro 15:08:43 2008 -0800


Mesclar commit 'phedders/rdocs'
```

Escapando o cursor no Windows

No Windows, em cmd.exe, ^ é um caractere especial e precisa ser tratado de maneira diferente. Você pode duplicá-lo ou colocar a referência do commit entre aspas:

```
$ git show HEAD^          # NÃO funcionará no Windows
```

Machine Translated by Google

```
$ git show CABEÇA^^        # OK
$ git show "HEAD^" # OK
```

Você também pode especificar um número após o ^ para identificar *qual* pai deseja; por exemplo, d921970^2 significa “o segundo pai de d921970”. Esta sintaxe é útil apenas para merge commits, que têm mais de um pai — o *primeiro* pai de um merge commit é do branch em que você estava quando fez o merge (frequentemente master), enquanto o *segundo* pai de um merge commit é do branch que foi mesclado (digamos, tópico):

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Autor: Scott Chacon <schacon@gmail.com> Data: Qui 11 de
dezembro 14:58:32 2008 -0800

Adicione alguma culpa e mescle coisas

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Autor: Paul Hedderly <paul+git@mjr.org> Data: quarta-feira,
10 de dezembro 22:22:03 2008 +0000

Algumas mudanças rdoc
```

A outra especificação de ancestralidade principal é o ~ (til). Isso também se refere ao primeiro pai, então HEAD~ e HEAD^ são equivalentes. A diferença se torna aparente quando você especifica um número. HEAD~2 significa “o primeiro pai do primeiro pai” ou “o avô” — ele percorre os primeiros pais o número de vezes que você especificar. Por exemplo, no histórico listado anteriormente, HEAD~3 seria:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Autor: Tom Preston-Werner <tom@mojombo.com> Data:
Sex Nov 7 13:47:59 2008 -0500

Ignorar *.gem
```

Isso também pode ser escrito HEAD~~~, que novamente é o primeiro pai do primeiro pai do primeiro pai:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Autor: Tom Preston-Werner <tom@mojombo.com> Data: Sex
Nov 7 13:47:59 2008 -0500

Ignorar *.gem
```

Você também pode combinar essas sintaxes — você pode obter o segundo pai da referência anterior (assumindo que foi um merge commit) usando HEAD~3^2, e assim por diante.

Machine Translated by Google

Intervalos de compromisso

Agora que você pode especificar commits individuais, vamos ver como especificar intervalos de commits. Isso é particularmente útil para gerenciar suas ramificações - se você tiver muitas ramificações, poderá usar as especificações de intervalo para responder a perguntas como: "Que trabalho há nesta ramificação que ainda não foi mesclado em minha ramificação principal?"

ponto duplo

A especificação de intervalo mais comum é a sintaxe de ponto duplo. Isso basicamente pede ao Git para resolver uma série de commits que podem ser acessados de um commit, mas não podem ser acessados de outro. Por exemplo, digamos que você tenha um histórico de confirmação parecido com o histórico de exemplo para seleção de intervalo.

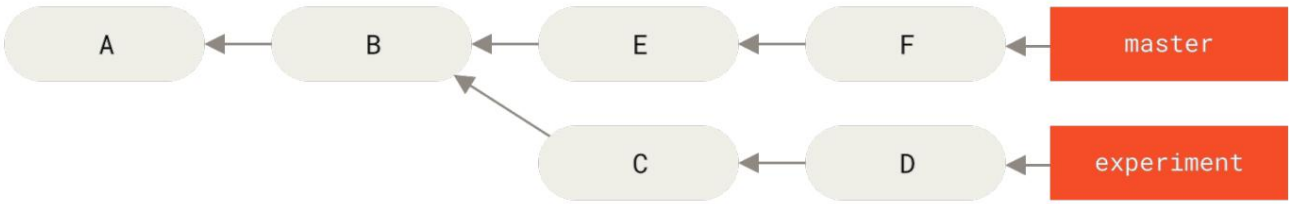


Figura 136. Histórico de exemplo para seleção de intervalo

Digamos que você queira ver o que está em seu branch experimental que ainda não foi mesclado em seu branch master . Você pode pedir ao Git para mostrar um log apenas daqueles commits com master..experiment — isso significa “todos os commits acessíveis a partir do experimento que não podem ser acessados pelo master”. Por uma questão de brevidade e clareza nesses exemplos, as letras dos objetos commit do diagrama são usadas no lugar da saída do log real na ordem em que seriam exibidas:

```
$ git log master..experiment D

C
```

Se, por outro lado, você quiser ver o oposto — todos os commits no master que não estão em experimento — você pode inverter os nomes dos branches. experiment..master mostra tudo no master que não pode ser acessado pelo experimento:

```
$ git log experimento..master FE
```

Isso é útil se você deseja manter a ramificação da experiência atualizada e visualizar o que está prestes a mesclar. Outro uso frequente dessa sintaxe é ver o que você está prestes a enviar para um controle remoto:

```
$ git log origin/master..HEAD
```

Este comando mostra todos os commits em sua ramificação atual que não estão na ramificação master em seu remoto de origem . Se você executar um git push e sua ramificação atual estiver rastreando origin/master, o

Machine Translated by Google

os commits listados por git log origin/master..HEAD são os commits que serão transferidos para o servidor. Você também pode deixar de fora um lado da sintaxe para que o Git assuma HEAD. Por exemplo, você pode obter os mesmos resultados do exemplo anterior digitando git log origin/master. — Git substitui HEAD se um lado estiver faltando.

Múltiplos Pontos

A sintaxe de ponto duplo é útil como um atalho, mas talvez você queira especificar mais de dois branches para indicar sua revisão, como ver quais commits estão em qualquer um dos vários branches que não estão no branch em que você está atualmente. O Git permite que você faça isso usando o caractere ^ ou --not antes de qualquer referência da qual você não deseja ver os commits acessíveis. Assim, os três comandos a seguir são equivalentes:

```
$ git log refA..refB $ git
log ^refA refB $ git log
refB --não refA
```

Isso é bom porque com essa sintaxe você pode especificar mais de duas referências em sua consulta, o que não pode ser feito com a sintaxe de ponto duplo. Por exemplo, se você quiser ver todos os commits que podem ser acessados de refA ou refB , mas não de refC, você pode usar um dos seguintes:

```
$ git log refA refB ^refC $ git
log refA refB --não refC
```

Isso cria um sistema de consulta de revisão muito poderoso que deve ajudá-lo a descobrir o que está em suas ramificações.

ponto triplo

A última grande sintaxe de seleção de intervalo é a sintaxe de ponto triplo, que especifica todos os commits que podem ser acessados por *qualquer* uma das duas referências, mas não por ambas. Veja o histórico de confirmação de exemplo em Histórico de exemplo para seleção de intervalo. Se você quiser ver o que está no mestre ou no experimento , mas não nas referências comuns, execute:

```
$ git log master...experimente FE


D
C
```

Novamente, isso fornece a saída de log normal , mas mostra apenas as informações de confirmação para essas quatro confirmações, aparecendo na ordem tradicional de data de confirmação.

Uma opção comum para usar com o comando log neste caso é --left-right, que mostra em que lado do intervalo cada confirmação está. Isso ajuda a tornar a saída mais útil:

Machine Translated by Google

```
$ git log --esquerda-direita mestre...experiência < F

< E
>D
>C
```

Com essas ferramentas, você pode informar ao Git com muito mais facilidade qual commit ou commits você deseja inspecionar.

Encenação Interativa

Nesta seção, você verá alguns comandos Git interativos que podem ajudá-lo a criar seus commits para incluir apenas certas combinações e partes de arquivos. Essas ferramentas são úteis se você modificar vários arquivos extensivamente e decidir que deseja que essas alterações sejam particionadas em vários commits focados, em vez de um grande commit confuso. Dessa forma, você pode garantir que seus commits sejam conjuntos de alterações logicamente separados e possam ser revisados facilmente pelos desenvolvedores que trabalham com você.

Se você executar git add com a opção -i ou --interactive , o Git entrará em um modo de shell interativo, exibindo algo como isto:

```
$ git add -i
             staged          caminho não
1: 2:        inalterado          preparado
             inalterado          +0/-1 TODO +1/-1
3:           inalterado          index.html +5/-1 lib/simplegit.rb

*** Comandos *** 1:
[s]status 5: [p]atch     2: [u] atualize     3: [r] inverter     4: [a] adicionar não
                         6: [d] iff          7: [q] desligar     rastreado 8: [h] ajuda
E agora>
```

Você pode ver que este comando mostra uma visão muito diferente da sua área de preparação do que você provavelmente está acostumado — basicamente, a mesma informação que você obtém com git status , mas um pouco mais sucinta e informativa. Ele lista as alterações que você testou à esquerda e as alterações não selecionadas à direita.

Depois disso, vem uma seção de “Comandos”, que permite que você faça várias coisas, como preparar e remover arquivos, preparar partes de arquivos, adicionar arquivos não rastreados e exibir diferenças do que foi testado.

Arquivos de preparação e remoção de preparação

Se você digitar u ou 2 (para atualização) no prompt What now> , será solicitado a informar quais arquivos deseja testar:

```
E agora> vc
                                caminho não encenado
1:           encenado inalterado  +0/-1 TODO
```

```
2:          inalterado          +1/-1 index.html +5/-1
3:                              lib/simplegit.rb
Atualizar>>
```

Para testar os arquivos TODO e index.html , você pode digitar os números:

```
Atualização >> 1,2
                estágio         caminho não
* 1:          inalterado          preparado
* 2:          inalterado          +0/-1 TODO +1/-1
3:            inalterado inalterado   index.html +5/-1 lib/simplegit.rb
Atualizar>>
```

O * ao lado de cada arquivo significa que o arquivo foi selecionado para ser testado. Se você pressionar Enter depois de não digitar nada no prompt Update>> , o Git pega qualquer coisa selecionada e prepara para você:

```
Atualizar >>
atualizado 2 caminhos

*** Comandos *** 1:
[s]status 5: [p]atch      2: [u] atualize      3: [r] inverter      4: [a] adicionar não
                          6: [d] iff           7: [q] desligar      rastreado 8: [h] ajuda
O que agora é
                                caminho não
1:                                preparado nada
2:                                TODO nada index.html
3:          encenado +0/-1 +1/-1 inalterado +5/-1 lib/simplegit.rb
```

Agora você pode ver que os arquivos TODO e index.html foram testados e o arquivo simplegit.rb ainda não foi testado. Se você quiser remover o arquivo TODO neste ponto, use a opção r ou 3 (para reverter):

```
*** Comandos *** 1:
[s]status 5: [p]atch      2: [u] atualize      3: [r] inverter      4: [a] adicionar não
                          6: [d] iff           7: [q] desligar      rastreado 8: [h] ajuda
E agora> r
                encenado        caminho não
1: 2:             +0/-1           preparado nada
3:                +1/-1           TODO nada index.html
            inalterado                +5/-1 lib/simplegit.rb
Reverter>> 1

                                caminho não
* 1:                              preparado nada
2:                                TODO nada index.html
encenado +0/-1 +1/-1 3: inalterado  +5/-1 lib/simplegit.rb
Reverter>> [enter]
reverteu um caminho
```

Machine Translated by Google

Olhando para o seu status Git novamente, você pode ver que removeu o arquivo TODO :

```
*** Comandos ***
1: [s]status 5:          2: [u] atualize          3: [r] inverter          4: [a] adicionar não
[p]atch                  6: [d] iff               7: [q] desligar          rastreado 8: [h] ajuda
O que agora>
                  é                caminho não
1:          encenado                 preparado
2: 3:                                +0/-1 TODO nada
            inalterado +1/-1 inalterado index.html +5/-1 lib/simplegit.rb
```

Para ver o diff do que você preparou, você pode usar o comando d ou 6 (para diff). Ele mostra uma lista de seus arquivos preparados e você pode selecionar aqueles para os quais gostaria de ver o diff preparado. Isso é como especificar git diff --cached na linha de comando:

```
*** Comandos ***
1: [s]status 5:          2: [u] atualize          3: [r] inverter          4: [a] adicionar não
[p]atch What now>        6: [d] iff               7: [q] desligar          rastreado 8: [h] ajuda
d staged +1/-1
                  Review           caminho não
1:                 diff>>            preparado nada index.html
1 diff --git a/
index.html b/index.html index
4d07108 ..4335f49 100644
--- a/index.html ++
+ b/index.html @@
-16,7 +16,7 @@ Data Finder

<p id="out">...</p>

-<div id="footer">contato: support@github.com</div> +<div
id="footer">contato: email.support@github.com</div>

<tipo de script="texto/javascript">
```

Com esses comandos básicos, você pode usar o modo de adição interativo para lidar com sua área de preparação com um pouco mais de facilidade.

Patches de preparação

Também é possível para o Git preparar certas *partes* dos arquivos e não o resto. Por exemplo, se você fizer duas alterações em seu arquivo simplegit.rb e quiser testar uma delas e não a outra, fazer isso é muito fácil no Git. No mesmo prompt interativo explicado na seção anterior, digite p ou 5 (para patch). O Git perguntará quais arquivos você gostaria de testar parcialmente; então, para cada seção dos arquivos selecionados, ele exibirá pedaços do diff do arquivo e perguntará se você gostaria de organizá-los, um por
um:

Machine Translated by Google

```
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


culpa def (caminho)
Encenar este pedaço [y,n,a,d,/,j,J,g,e,?]?
```

Você tem muitas opções neste momento. Digitando ? mostra uma lista do que você pode fazer:

```
Encenar este pedaço [y,n,a,d,/,j,J,g,e,?]? ? y - preparar
este pedaço n - não preparar este pedaço a - preparar
este e todos os pedaços restantes no arquivo d - não
preparar este pedaço nem nenhum dos pedaços restantes no arquivo g -
selecionar um pedaço para ir para / - procurar um pedaço correspondente ao regex fornecido j
- deixar este pedaço indeciso, ver o próximo pedaço indeciso J - deixar este pedaço indeciso,
ver o próximo pedaço k - deixar este pedaço indeciso, ver o pedaço indeciso anterior K - deixar
este pedaço indeciso, ver o pedaço anterior s - dividir o pedaço atual em pedaços menores e -
editar manualmente o pedaço atual ? - ajuda para impressão
```

Geralmente, você digitará s ou n se quiser testar cada pedaço, mas colocar todos eles em determinados arquivos ou pular uma decisão de pedaço para mais tarde também pode ser útil. Se você preparar uma parte do arquivo e deixar outra parte não selecionada, sua saída de status ficará assim:

```
E agora> 1
                                caminho não
1: 2:       encenado                preparado
                                 +0/-1 TODO nada
3:                inalterado +1/-1 +1/-1 index.html +4/-0 lib/simplegit.rb
```

O status do arquivo simplegit.rb é interessante. Isso mostra que algumas falas são encenadas e algumas não são encenadas. Você preparou parcialmente este arquivo. Neste ponto, você pode sair do script de adição interativo e executar git commit para confirmar os arquivos parcialmente preparados.

Você também não precisa estar no modo de adição interativo para fazer a preparação de arquivo parcial - você pode iniciar o

Machine Translated by Google

mesmo script usando git add -p ou git add --patch na linha de comando.

Além disso, você pode usar o modo patch para redefinir parcialmente os arquivos com o comando git reset --patch , para verificar partes de arquivos com o comando git checkout --patch e para armazenar partes de arquivos com o comando git stash save --patch . Entraremos em mais detalhes sobre cada um deles à medida que chegarmos a usos mais avançados desses comandos.

Armazenamento e Limpeza

Frequentemente, quando você está trabalhando em parte do seu projeto, as coisas estão confusas e você quer trocar de ramo um pouco para trabalhar em outra coisa. O problema é que você não quer fazer um commit de trabalho incompleto apenas para poder voltar a esse ponto mais tarde. A resposta para esse problema é o comando git stash .

O stashing pega o estado sujo do seu diretório de trabalho - ou seja, seus arquivos rastreados modificados e alterações preparadas - e o salva em uma pilha de alterações inacabadas que você pode reaplicar a qualquer momento (mesmo em uma ramificação diferente).

ÿ

Migrando para git stash push

No final de outubro de 2017, houve uma extensa discussão na lista de discussão do Git, em que o comando git stash save está sendo descontinuado em favor da alternativa existente git stash push. A principal razão para isso é que git stash push introduz a opção de armazenar *pathspecs selecionados,* algo que git stash save não suporta.

git stash save não vai desaparecer tão cedo, então não se preocupe com ele desaparecer de repente. Mas você pode querer começar a migrar para a alternativa push para a nova funcionalidade.

Escondendo seu trabalho

Para demonstrar o stashing, você entrará em seu projeto e começará a trabalhar em alguns arquivos e possivelmente preparará uma das alterações. Se você executar git status, poderá ver seu estado sujo:

```
$ git status
Mudanças a serem
confirmadas: (use "git reset HEAD <arquivo>..." para retirar o estágio)

modificado: index.html

Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será
submetido) (use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: lib/simplegit.rb
```

Agora você quer mudar de branch, mas não quer comprometer o que está trabalhando ainda,

Machine Translated by Google

então você vai esconder as alterações. Para enviar um novo stash para sua pilha, execute git stash ou git stash push:

```
$ git stash
Diretório de trabalho salvo e estado do índice \
"WIP no mestre: 049d078 Criar arquivo de índice"
HEAD está agora em 049d078 Criar arquivo de índice
(Para restaurá-los, digite "git stash apply")
```

Agora você pode ver que seu diretório de trabalho está limpo:

```
$ git status
# No branch master
nada para confirmar, diretório de trabalho limpo
```

Neste ponto, você pode trocar de ramo e trabalhar em outro lugar; suas alterações são armazenadas em sua pilha. Para ver quais stashes você armazenou, você pode usar git stash list:

```
$ git stash list
stash@{0}: WIP no master: 049d078 Criar arquivo de índice
stash@{1}: WIP no master: c264051 Reverter "Add file_size" stash@{2}:
WIP no master: 21d80a5 Adicionar número ao log
```

Neste caso, dois esconderijos foram salvos anteriormente, então você tem acesso a três trabalhos escondidos diferentes. Você pode reaplicar aquele que acabou de armazenar usando o comando mostrado na saída de ajuda do comando stash original: git stash apply. Se você deseja aplicar um dos stashes mais antigos, pode especificá-lo nomeando-o, assim: git stash apply stash@{2}. Se você não especificar um stash, o Git assume o stash mais recente e tenta aplicá-lo:

```
$ git stash apply No
branch master
Alterações não preparadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido)
(use "git checkout -- <arquivo>..." para descartar as alterações em diretório de trabalho)

modificado: index.html modificado:
lib/simplegit.rb

nenhuma alteração adicionada ao commit (use "git add" e/ou "git commit -a")
```

Você pode ver que o Git modifica novamente os arquivos que você reverteu quando salvou o stash. Nesse caso, você tinha um diretório de trabalho limpo quando tentou aplicar o stash e tentou aplicá-lo no mesmo branch do qual o salvou. Ter um diretório de trabalho limpo e aplicá-lo na mesma ramificação não é necessário para aplicar um stash com êxito. Você pode salvar um stash em uma ramificação, alternar para outra ramificação posteriormente e tentar reaplicar as alterações. Você também pode ter arquivos modificados e não confirmados em seu diretório de trabalho ao aplicar um stash — o Git oferece conflitos de mesclagem, se houver

Machine Translated by Google

não se aplica mais de forma limpa.

As alterações em seus arquivos foram reaplicadas, mas o arquivo que você testou antes não foi testado novamente. Para fazer isso, você deve executar o comando git stash apply com uma opção --index para instruir o comando a tentar reaplicar as alterações preparadas. Se você tivesse executado isso, teria voltado à sua posição original:

```
$ git stash apply --index No
branch master Mudanças a
serem confirmadas: (use "git
reset HEAD <arquivo>..." para retirar o estágio)

modificado: index.html

Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será
submetido) (use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: lib/simplegit.rb
```

A opção de aplicar apenas tenta aplicar o trabalho armazenado - você continua a tê-lo em sua pilha. Para removê-lo, você pode executar git stash drop com o nome do stash a ser removido:

```
$ git stash list
stash@{0}: WIP no master: 049d078 Criar arquivo de índice
stash@{1}: WIP no master: c264051 Reverter "Add file_size"
stash@{2}: WIP no master: 21d80a5 Adicionar número ao log $ git
stash drop stash@{0}
Stash perdido @{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Você também pode executar git stash pop para aplicar o stash e, em seguida, soltá-lo imediatamente de sua pilha.

Armazenamento de criativos

Existem algumas variantes de estoque que também podem ser úteis. A primeira opção bastante popular é a opção --keep-index para o comando git stash . Isso diz ao Git para não apenas incluir todo o conteúdo preparado no stash que está sendo criado, mas simultaneamente deixá-lo no índice.

```
$ git status -s M
index.html
M lib/simplegit.rb

$ git stash --keep-index
Diretório de trabalho salvo e estado de índice WIP no mestre: 1b65b17 adicionou o arquivo de índice
HEAD está agora em 1b65b17 adicionado o arquivo de índice

$ git status -s
```

Machine Translated by Google

```
M index.html
```

Outra coisa comum que você pode querer fazer com o stash é armazenar os arquivos não rastreados, bem como os rastreados. Por padrão, o git stash armazenará apenas arquivos *rastreados* modificados e preparados . Se você especificar --include-untracked ou -u, o Git incluirá arquivos não rastreados no stash que está sendo criado. No entanto, incluir arquivos não rastreados no stash ainda não incluirá arquivos explicitamente *ignorados ;* para incluir adicionalmente arquivos ignorados, use --all (ou apenas -a).

```
$ git status -s M
index.html M lib/
simplegit.rb ?? novo-
arquivo.txt


$ git stash -u
Diretório de trabalho salvo e estado de índice WIP no mestre: 1b65b17 adicionou o arquivo de índice
HEAU está agora em 1b65b17 adicionado o arquivo de índice


$ git status -s $
```

Por fim, se você especificar o sinalizador --patch , o Git não ocultará tudo o que for modificado, mas, em vez disso, solicitará interativamente quais alterações você gostaria de ocultar e quais gostaria de manter em seu diretório de trabalho.

```
$ 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(árvore = 'mestre')
+            command("git show #{treeish}") end
+

fim
teste
Esconda este pedaço [y,n,q,a,d,/,e,?]? y

Diretório de trabalho salvo e estado de índice WIP no mestre: 1b65b17 adicionou o arquivo de índice
```

Criando uma ramificação a partir de um Stash

Se você esconder algum trabalho, deixe-o lá por um tempo e continue no ramo de onde você

Machine Translated by Google

escondeu o trabalho, você pode ter problemas para reaplicar o trabalho. Se o aplicativo tentar modificar um arquivo que você modificou desde então, você terá um conflito de mesclagem e terá que tentar resolvê-lo. Se você deseja uma maneira mais fácil de testar as alterações armazenadas novamente, pode executar git stash branch <new branchname>, que cria uma nova ramificação para você com o nome da ramificação selecionada, verifica o commit em que você estava quando ocultou seu trabalho, reaplica seu trabalho lá e, em seguida, descarta o stash se for aplicado com sucesso:

```
$ git stash branch testchanges M
index.html M lib/simplegit.rb Alterado
para um novo branch 'testchanges'

No branch testchanges
Alterações a serem
confirmadas: (use "git reset HEAD <arquivo>..." para remover o estágio)

modificado: index.html


Alterações não testadas para
confirmação: (use "git add <arquivo>..." para atualizar o que será submetido)
(use "git checkout -- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: lib/simplegit.rb


Refs/stash@{0} descartados (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Este é um bom atalho para recuperar facilmente o trabalho armazenado e trabalhar nele em uma nova ramificação.

Limpando seu diretório de trabalho

Finalmente, você pode não querer esconder alguns trabalhos ou arquivos em seu diretório de trabalho, mas simplesmente se livrar deles; é para isso que serve o comando git clean .

Alguns motivos comuns para limpar seu diretório de trabalho podem ser remover lixo gerado por fusões ou ferramentas externas ou remover artefatos de compilação para executar uma compilação limpa.

Você deve ter muito cuidado com este comando, pois ele foi projetado para remover arquivos de seu diretório de trabalho que não são rastreados. Se você mudar de ideia, muitas vezes não há como recuperar o conteúdo desses arquivos. Uma opção mais segura é executar git stash --all para remover tudo, mas salvá-lo em um stash.

Supondo que você queira remover arquivos cruft ou limpar seu diretório de trabalho, você pode fazer isso com git clean. Para remover todos os arquivos não rastreados em seu diretório de trabalho, você pode executar git clean -f -d, que remove todos os arquivos e também quaisquer subdiretórios que ficam vazios como resultado. O -f significa 'forçar' ou “realmente fazer isso” e é necessário se a variável de configuração do Git clean.requireForce não for definida explicitamente como false.

Se você quiser ver o que ele faria, você pode executar o comando com a opção --dry-run (ou -n) , que significa “faça uma simulação e me diga o que você *teria* removido”.

Machine Translated by Google

```
$ git clean -d -n
Removeria test.o
Iria remover tmp/
```

Por padrão, o comando git clean removerá apenas arquivos não rastreados que não são ignorados. Qualquer arquivo que corresponda a um padrão em seu .gitignore ou outros arquivos ignorados não serão removidos. Se você deseja remover esses arquivos também, como remover todos os arquivos .o gerados de uma compilação para poder fazer uma compilação totalmente limpa, pode adicionar um -x ao comando clean.

```
$ git status -s M
lib/simplegit.rb ??
construir.TMP
?? tmp/

$ git clean -n -d
Removeria build.TMP
Iria remover tmp/

$ git clean -n -d -x
Removeria build.TMP
Iria remover test.o
Iria remover tmp/
```

Se você não sabe o que o comando git clean fará, sempre execute-o com um -n primeiro para verificar novamente antes de alterar o -n para a -f e fazê-lo de verdade. A outra maneira de ter cuidado com o processo é executá-lo com o sinalizador -i ou “interativo”.

Isso executará o comando clean em um modo interativo.

```
$ git clean -x -i
Removeria os seguintes itens:
build.TMP test.o *** Commands ***
1: clean each
                    2: filtrar por padrão        3: selecione por números        4: pergunte
          5: sair
6: ajuda
E agora>
```

Dessa forma, você pode percorrer cada arquivo individualmente ou especificar padrões para exclusão interativamente.

ÿ

Há uma situação peculiar em que você pode precisar ser mais enérgico ao pedir ao Git para limpar seu diretório de trabalho. Se você estiver em um diretório de trabalho no qual copiou ou clonou outros repositórios Git (talvez como submódulos), mesmo git clean -fd se recusará a excluir esses diretórios. Em casos como esse, você precisa adicionar uma segunda opção -f para dar ênfase.

Machine Translated by Google

Assinando seu trabalho

Git é criptograficamente seguro, mas não é infalível. Se você estiver recebendo trabalho de outras pessoas na Internet e quiser verificar se os commits são realmente de uma fonte confiável, o Git tem algumas maneiras de assinar e verificar o trabalho usando o GPG.

GPG Introdução

Primeiro de tudo, se você quiser assinar qualquer coisa, você precisa configurar o GPG e instalar sua chave pessoal.

```
$ gpg --list-keys /Users/
schacon/.gnupg/pubring.gpg
---------------------------------

pub 2048R/0A46826A 2014-06-04 uid
sub 2048R/874529A9 2014-06-04 Scott Chacon (chave de assinatura Git) <schacon@gmail.com>
```

Se você não tiver uma chave instalada, poderá gerar uma com gpg --gen-key.

```
$ gpg --gen-key
```

Depois de ter uma chave privada para assinar, você pode configurar o Git para usá-la para assinar coisas, definindo a configuração user.signingkey .

```
$ git config --global user.signingkey 0A46826A!
```

Agora o Git usará sua chave por padrão para assinar tags e commits se você quiser.

Marcas de assinatura

Se você tiver uma chave privada GPG configurada, agora poderá usá-la para assinar novas tags. Tudo o que você precisa fazer é usar -s em vez de -a:

```
$ git tag -s v1.5 -m 'minha tag 1.5 assinada'

Você precisa de uma frase secreta para desbloquear a chave
secreta do usuário: "Ben Straub <ben@straub.cc>" Chave RSA
de 2048 bits, ID 800430EB, criada em 2014-05-04
```

Se você executar git show nessa tag, poderá ver sua assinatura GPG anexada a ela:

```
$ git show v1.5 tag
v1.5 Tagger: Ben
Straub <ben@straub.cc>
```

Machine Translated by Google

```
Data: sábado, 3 de maio 20:29:41 2014 -0700

minha tag 1.5 assinada
-----BEGIN PGP SIGNATURE----- Versão:
GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b hM1/
PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----TERMINAR ASSINATURA PGP-----

commit ca82a6dff817ec66f44342007202690a93763949 Autor: Scott
Chacon <schacon@gee-mail.com> Data: Seg Mar 17 21:52:11 2008
-0700

Alterar número da versão
```

Verificando tags

Para verificar uma tag assinada, use git tag -v <tag-name>. Este comando usa GPG para verificar a assinatura. Você precisa da chave pública do signatário em seu chaveiro para que isso funcione corretamente:

```
$ git tag -v v1.4.2.1 objeto
883653babd8ee7ea23e6a5c392bb739348b1eb61 type commit tag
v1.4.2.1 tagger Junio C Hamano <junkio@cox.net> 1158138501
-0700


GIT 1.4.2.1

Pequenas correções desde 1.4.2, incluindo git-mv e git-http com alternativas. gpg: Assinatura feita na
quarta-feira, 13 de setembro 02:08:25 2006 PDT usando o ID de chave DSA F3119B9A gpg: Boa assinatura
de "Junio C Hamano <junkio@cox.net>" também conhecido como "[imagem jpeg de tamanho 1513]" gpg:
D9A4 F311 9B9A                    Chave primária impressão digital: 3565 2A26 2040 E066 C9A7 4A7D C0C6
```

Se você não tiver a chave pública do signatário, obterá algo assim:

```
gpg: Assinatura feita na quarta-feira, 13 de setembro 02:08:25 2006 PDT usando ID de chave DSA F3119B9A
gpg: Não é possível verificar a assinatura: erro de chave pública não encontrada: não foi possível verificar a
tag 'v1.4.2.1'
```

Machine Translated by Google

Confirmações de assinatura

Nas versões mais recentes do Git (v1.7.9 e superior), agora você também pode assinar commits individuais. Se você estiver interessado em assinar os commits diretamente em vez de apenas as tags, tudo o que você precisa fazer é adicionar um -S ao seu comando git commit .

```
$ git commit -a -S -m 'Commit assinado'

Você precisa de uma senha para desbloquear a chave secreta do
usuário: "Scott Chacon (chave de assinatura do Git) <schacon@gmail.com>"
Chave RSA de 2048 bits, ID 0A46826A, criada em 2014-06-04

[mestre 5c3386c] Confirmação assinada
4 arquivos alterados, 4 inserções (+), 24 exclusões (-) reescrever
Rakefile (100%) modo de criação 100644 lib/git.rb
```

Para ver e verificar essas assinaturas, há também uma opção --show-signature para git log.

```
$ git log --show-signature -1 commit
5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Assinatura feita na quarta-feira, 4 de junho 19:49:17 2014 PDT usando o ID de chave RSA
0A46826A gpg: Boa assinatura de "Scott Chacon (chave de assinatura do Git) <schacon@gmail.com>"
Autor: Scott Chacon <schacon@gmail.com> Data:
quarta-feira, 4 de junho 19:49:17 2014 -0700

confirmação assinada
```

Além disso, você pode configurar o git log para verificar todas as assinaturas encontradas e listá-las em sua saída com o %G? formatar.

```
$ git log --pretty="format:%h %G? %aN %s"

5c3386c G Scott Chacon Commit assinado
ca82a6d N Scott Chacon Alterar o número da versão 085bb3b
N Scott Chacon Remover código de teste desnecessário a11bef0 N
Scott Chacon Commit inicial
```

Aqui podemos ver que apenas o commit mais recente é assinado e válido e os commits anteriores não.

No Git 1.8.3 e posterior, git merge e git pull podem ser instruídos a inspecionar e rejeitar ao mesclar um commit que não carrega uma assinatura GPG confiável com o comando --verify-signatures .

Se você usar esta opção ao mesclar uma ramificação e ela contiver commits que não sejam assinados e válidos, a mesclagem não funcionará.

```
$ git merge --verify-signatures non-verify
```

Machine Translated by Google

```
fatal: Commit ab06180 não possui uma assinatura GPG.
```

Se a mesclagem contiver apenas commits assinados válidos, o comando merge mostrará todas as assinaturas verificadas e, em seguida, prosseguirá com a mesclagem.

```
$ git merge --verify-signatures-branch assinado Commit
13ad65e tem uma boa assinatura GPG por Scott Chacon (chave de assinatura do Git)
<schacon@gmail.com> Atualizando 5c3386c..13ad65e Avanço rápido README | 2 ++ 1 arquivo
alterado, 2 inserções (+)
```

Você também pode usar a opção -S com o comando git merge para assinar a própria confirmação de mesclagem resultante. O exemplo a seguir verifica se cada confirmação na ramificação a ser mesclada está assinada e, além disso, assina a confirmação de mesclagem resultante.

```
$ git merge --verify-signatures -S assinado-branch Commit
13ad65e tem uma boa assinatura GPG por Scott Chacon (chave de assinatura do Git)
<schacon@gmail.com>

Você precisa de uma senha para desbloquear a chave secreta do
usuário: "Scott Chacon (chave de assinatura do Git) <schacon@gmail.com>"
Chave RSA de 2048 bits, ID 0A46826A, criada em 2014-06-04

Fusão feita pela estratégia 'recursiva'.
LEIA-ME | 2 ++ 1
arquivo alterado, 2 inserções (+)
```

Todos devem assinar

Assinar tags e commits é ótimo, mas se você decidir usar isso em seu fluxo de trabalho normal, precisará garantir que todos em sua equipe entendam como fazer isso. Se você não fizer isso, acabará gastando muito tempo ajudando as pessoas a descobrir como reescrever seus commits com versões assinadas.
Certifique-se de entender o GPG e os benefícios de assinar coisas antes de adotá-lo como parte de seu fluxo de trabalho padrão.

Procurando

Com praticamente qualquer base de código de tamanho, muitas vezes você precisará descobrir onde uma função é chamada ou definida ou exibir o histórico de um método. O Git fornece algumas ferramentas úteis para examinar o código e os commits armazenados em seu banco de dados de forma rápida e fácil. Nós vamos passar por alguns deles.

Git GrepName

O Git vem com um comando chamado grep que permite que você pesquise facilmente em qualquer árvore comprometida,

Machine Translated by Google

o diretório de trabalho ou até mesmo o índice de uma string ou expressão regular. Para os exemplos a seguir, pesquisaremos o código-fonte do próprio Git.

Por padrão, o git grep examinará os arquivos em seu diretório de trabalho. Como primeira variação, você pode usar qualquer uma das opções -n ou --line-number para imprimir os números das linhas onde o Git encontrou correspondências:

```
$ 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 *resultado) compat/
gmtime.c:16: ret = gmtime_r(timep, resultado); compat/mingw.c:826:struct tm *gmtime_r(const time_t *timep, struct tm *resultado) compat/
mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct tm *resultado); data.c:482:
data.c:545:
                                if (gmtime_r(&now, &now_tm))
                                if (gmtime_r(&time, tm))
{ date.c:758: /* gmtime_r() em match_digit() pode ter derrotado */ 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
```

Além da pesquisa básica mostrada acima, o git grep oferece suporte a uma infinidade de outras opções interessantes.

Por exemplo, em vez de imprimir todas as correspondências, você pode pedir ao git grep para resumir a saída, mostrando apenas quais arquivos continham a string de pesquisa e quantas correspondências havia em cada arquivo com a opção -c ou --count :

```
$ git grep --count gmtime_r
compat/gmtime.c:4 compat/
mingw.c:1 compat/mingw.h:1
date.c:3 git-compat-util.h:2
```

Se estiver interessado no *contexto* de uma string de pesquisa, você pode exibir o método ou a função de inclusão para cada string correspondente com as opções -p ou --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: date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
                                if (gmtime_r(&now, &now_tm))
*tm_gmt) if (gmtime_r(&time, tm)) { date.c: date.c=int parse_date_basic(const char
*date, timestamp_t *timestamp, int *offset) date.c:


                        /* gmtime_r() em match_digit() pode tê-lo derrotado */
```

Como você pode ver, a rotina gmtime_r é chamada tanto do match_multi_number quanto do match_digit

Machine Translated by Google

funções no arquivo date.c (a terceira correspondência exibida representa apenas a string que aparece em um comentário).

Você também pode pesquisar combinações complexas de strings com o sinalizador --and , que garante que várias correspondências ocorram na mesma linha de texto. Por exemplo, vamos procurar por quaisquer linhas que definam uma constante cujo nome contenha *uma* das substrings “LINK” ou “BUF_MAX”, especificamente em uma versão mais antiga da base de código Git representada pela tag v1.8.0 (nós lançaremos a opções --break e --heading que ajudam a dividir a saída em um formato mais legível):

```
$ git grep --break --heading \ -n -e
'#define' --and \( -e LINK -e BUF_MAX \) v1.8.0 v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)


v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m)                    (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c 30:/
* #define ZLIB_BUF_MAX ((uInt)-1) */ 31:#define
ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

O comando git grep tem algumas vantagens sobre os comandos de pesquisa normais, como grep e ack.
A primeira é que é muito rápido, a segunda é que você pode pesquisar em qualquer árvore no Git, não apenas no diretório de trabalho. Como vimos no exemplo acima, procuramos termos em uma versão mais antiga do código-fonte do Git, não na versão que foi verificada no momento.

Pesquisa de Log do Git

Talvez você não esteja procurando *onde* um termo existe, mas *quando* ele existiu ou foi introduzido. O comando git log possui várias ferramentas poderosas para localizar commits específicos pelo conteúdo de suas mensagens ou até mesmo pelo conteúdo do diff que eles introduzem.

Se, por exemplo, quisermos descobrir quando a constante ZLIB_BUF_MAX foi introduzida originalmente, podemos usar a opção -S (coloquialmente chamada de opção “picareta” do Git) para dizer ao Git para nos mostrar apenas os commits que alteraram o número de ocorrências dessa string.

```
$ git log -S ZLIB_BUF_MAX --oneline
e01503b zlib: permite alimentar mais de 4 GB de uma só vez
```

Machine Translated by Google

```
ef49a7a zlib: zlib só pode processar 4 GB por vez
```

Se olharmos o diff desses commits, podemos ver que em ef49a7a a constante foi introduzida e em e01503b ela foi modificada.

Se precisar ser mais específico, você pode fornecer uma expressão regular para pesquisar com a opção -G .

Pesquisa de registro de linha

Outra pesquisa de log bastante avançada que é incrivelmente útil é a pesquisa de histórico de linha. Basta executar git log com a opção -L e ele mostrará o histórico de uma função ou linha de código em sua base de código.

Por exemplo, se quisermos ver todas as alterações feitas na função git_deflate_bound no arquivo zlib.c , podemos executar git log -L :git_deflate_bound:zlib.c. Isso tentará descobrir quais são os limites dessa função e, em seguida, examinará o histórico e nos mostrará todas as alterações feitas na função como uma série de patches desde quando a função foi criada pela primeira vez.

```
$ git log -L :git_deflate_bound:zlib.c commit
ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Autor: June C Hamano <gitster@pobox.com>
Data: Sex Jun 10 11:52:15 2011 -0700

zlib: zlib só pode processar 4 GB por vez

diff --git a/zlib.cb/zlib.c --- a/
zlib.c +++ b/zlib.c @@ -85,5
+130,5 @@ -unsigned long
git_deflate_bound(z_streamp
strm, unsigned tamanho longo) +longo sem sinal git_deflate_bound(git_zstream
*strm, tamanho longo sem sinal) {

-           return deflateBound(strm, tamanho);
+           return deflateBound(&strm->z, tamanho);
}


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Autor: Junio C Hamano <gitster@pobox.com> Data:
Sex Jun 10 11:18:17 2011 -0700

zlib: envolva deflateBound () também

diff --git a/zlib.cb/zlib.c --- a/
zlib.c +++ b/zlib.c @@ -81,0
+85,5 @@ +unsigned long
git_deflate_bound(z_streamp
strm, unsigned tamanho longo) +{
```

Machine Translated by Google

```
+        return deflateBound(strm, tamanho);
+}
+
```

Se o Git não conseguir descobrir como corresponder a uma função ou método em sua linguagem de programação, você também pode fornecer uma expressão regular (ou *regex).* Por exemplo, isso teria feito a mesma coisa que o exemplo acima: git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c. Você também pode fornecer um intervalo de linhas ou um único número de linha e obterá o mesmo tipo de saída.

Reescrevendo a História

Muitas vezes, ao trabalhar com o Git, você pode querer revisar seu histórico de commits local. Uma das grandes vantagens do Git é que ele permite que você tome decisões no último momento possível. Você pode decidir quais arquivos vão para quais commits logo antes de fazer commit com a área de staging, você pode decidir que não queria estar trabalhando em algo ainda com git stash, e você pode reescrever commits que já aconteceram para que pareçam aconteceu de uma forma diferente. Isso pode envolver alterar a ordem dos commits, alterar mensagens ou modificar arquivos em um commit, agrupar ou separar commits ou remover commits completamente - tudo antes de você compartilhar seu trabalho com outras pessoas.

Nesta seção, você verá como realizar essas tarefas para que possa fazer com que seu histórico de commits tenha a aparência desejada antes de compartilhá-lo com outras pessoas.

ÿ

Não force seu trabalho até que esteja satisfeito com ele

Uma das regras fundamentais do Git é que, como muito do trabalho é local dentro do seu clone, você tem muita liberdade para reescrever sua história *localmente.* No entanto, uma vez que você empurra seu trabalho, é uma história totalmente diferente e você deve considerar o trabalho forçado como final, a menos que tenha um bom motivo para alterá-lo. Resumindo, você deve evitar empurrar seu trabalho até que esteja satisfeito com ele e pronto para compartilhá-lo com o resto do mundo.

Alterando o último commit

Alterar seu commit mais recente é provavelmente a reescrita mais comum do histórico que você fará. Muitas vezes, você desejará fazer duas coisas básicas em seu último commit: simplesmente alterar a mensagem do commit ou alterar o conteúdo real do commit adicionando, removendo e modificando arquivos.

Se você simplesmente deseja modificar sua última mensagem de commit, é fácil:

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

O comando acima carrega a mensagem de confirmação anterior em uma sessão do editor, onde você pode fazer alterações na mensagem, salvar essas alterações e sair. Quando você salva e fecha o editor, o editor grava um novo commit contendo essa mensagem de commit atualizada e o torna seu novo último commit.

Se, por outro lado, você quiser alterar o *conteúdo* real do seu último commit, o processo funciona

Machine Translated by Google

basicamente da mesma maneira - primeiro faça as alterações que você acha que esqueceu, organize essas alterações e o subseqüente git commit --amend *substitui* o último commit pelo seu novo commit aprimorado.

Você precisa ter cuidado com esta técnica porque a alteração altera o SHA-1 do commit. É como um rebase muito pequeno - não altere seu último commit se você já o enviou.

ÿ

Um commit alterado pode (ou não) precisar de uma mensagem de commit alterado

Ao alterar um commit, você tem a oportunidade de alterar a mensagem do commit e o conteúdo do commit. Se você alterar substancialmente o conteúdo do commit, quase certamente deve atualizar a mensagem do commit para refletir o conteúdo alterado.

Por outro lado, se suas alterações forem adequadamente triviais (corrigindo um erro de digitação bobo ou adicionando um arquivo que você esqueceu de testar) de modo que a mensagem de confirmação anterior esteja correta, você pode simplesmente fazer as alterações, organizá-las e evitar o editor desnecessário sessão inteiramente com:

```
$ git commit --amend --no-edit
```

Mudando Múltiplas Mensagens de Commit

Para modificar um commit que está mais atrás em seu histórico, você deve mudar para ferramentas mais complexas. O Git não tem uma ferramenta de histórico de modificação, mas você pode usar a ferramenta de rebase para rebasear uma série de commits no HEAD no qual eles foram originalmente baseados, em vez de movê-los para outro. Com a ferramenta interativa de rebase, você pode parar após cada confirmação que deseja modificar e alterar a mensagem, adicionar arquivos ou fazer o que desejar. Você pode executar o rebase interativamente adicionando a opção -i ao git rebase. Você deve indicar a que distância deseja reescrever os commits, informando ao comando em qual commit deseja rebasear.

Por exemplo, se você deseja alterar as últimas três mensagens de commit ou qualquer uma das mensagens de commit desse grupo, forneça como argumento para git rebase -i o pai do último commit que deseja editar, que é HEAD~2 ^ ou CABEÇA~3. Pode ser mais fácil lembrar do ~3 porque você está tentando editar os últimos três commits, mas tenha em mente que você está designando quatro commits atrás, o pai do último commit que deseja editar:

```
$ git rebase -i HEAD~3
```

Lembre-se novamente que este é um comando de rebase — todo commit no intervalo HEAD~3..HEAD com uma mensagem alterada *e todos os seus descendentes* serão reescritos. Não inclua nenhum commit que você já tenha enviado para um servidor central — isso irá confundir outros desenvolvedores ao fornecer uma versão alternativa da mesma alteração.

Executar este comando fornece uma lista de commits em seu editor de texto que se parece com isto:

```
escolha f7f3f6d Mude um pouco meu nome
escolha 310154e Atualize a formatação README e adicione culpa
```

Machine Translated by Google

```
escolha a5f4a0d Adicionar arquivo cat

# Rebase 710f0f8..a5f4a0d em 710f0f8
#
# Comandos:
# p, pick <commit> = use commit # r,
reword <commit> = use commit, mas edite a mensagem de commit # e, edit
<commit> = use commit, mas pare para corrigir # s, squash <commit> = use
commit, mas funde-se com o commit anterior # f, fixup <commit> = como "squash",
mas descarta a mensagem de log deste commit # x, exec <command> = executa o comando (o
resto da linha) usando o shell # b, break = pare aqui (continue rebase mais tarde com 'git rebase --
continue') # d, drop <commit> = remove commit # l, label <label> = label HEAD atual com um nome
# t, reset <label> = reset HEAD para a label # m, merge [-C <commit> | -c <commit>] <label> [#
<oneline>] # .



                crie um merge commit usando a mensagem do merge commit original (ou
# .             o oneline, se nenhum merge commit original foi especificado). Use -c
# .             <commit> para reformular a mensagem de confirmação.
#
# Estas linhas podem ser reordenadas; eles são executados de cima para baixo. # # Se você
remover uma linha aqui ESSE COMMIT SERÁ PERDIDO. #



# No entanto, se você remover tudo, o rebase será abortado. # # Observe que os
commits vazios são comentados
```

É importante observar que esses commits são listados na ordem oposta à que você normalmente os vê usando o comando log . Se você executar um log, verá algo assim:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD a5f4a0d
Adicionar cat-file 310154e Atualize a formatação do README
e adicione a culpa f7f3f6d Mude meu nome um pouco
```

Observe a ordem inversa. O rebase interativo fornece um script que será executado. Ele começará no commit que você especificar na linha de comando (HEAD~3) e repetirá as alterações introduzidas em cada um desses commits de cima para baixo. Ele lista o mais antigo no topo, em vez do mais novo, porque é o primeiro a ser reproduzido.

Você precisa editar o script para que pare no commit que deseja editar. Para fazer isso, mude a palavra “pick” para a palavra “edit” para cada um dos commits que você deseja que o script pare depois. Por exemplo, para modificar apenas a terceira mensagem de confirmação, altere o arquivo para que fique assim:

```
editar f7f3f6d Alterar um pouco meu nome
pick 310154e Atualizar a formatação README e adicionar culpa
pick a5f4a0d Adicionar cat-file
```

Machine Translated by Google

Quando você salva e sai do editor, o Git volta para o último commit dessa lista e o coloca na linha de comando com a seguinte mensagem:

```
$ git rebase -i HEAD~3
Parado em f7f3f6d... Mude um pouco meu nome
Você pode alterar o commit agora, com

        git commit --amend

Quando estiver satisfeito com as alterações, execute

        git rebase --continuar
```

Estas instruções dizem exatamente o que fazer. Tipo:

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

Altere a mensagem de confirmação e saia do editor. Então corra:

```
$ git rebase --continuar
```

Este comando aplicará os outros dois commits automaticamente e pronto. Se você alterar pick para editar em mais linhas, poderá repetir essas etapas para cada commit que alterar para editar. A cada vez, o Git irá parar, deixar você corrigir o commit e continuar quando terminar.

Reordenando Commits

Você também pode usar rebases interativos para reordenar ou remover completamente os commits. Se você deseja remover o commit “Add cat-file” e alterar a ordem em que os outros dois commits são introduzidos, você pode alterar o script de rebase a partir deste:

```
pick f7f3f6d Mude um pouco meu nome
pick 310154e Atualize a formatação README e adicione a
culpa pick a5f4a0d Add cat-file
```

para isso:

```
pick 310154e Atualize a formatação README e adicione a
culpa pick f7f3f6d Mude meu nome um pouco
```

Quando você salva e sai do editor, o Git rebobina sua ramificação para o pai desses commits, aplica 310154e e depois f7f3f6d e então para. Você efetivamente altera a ordem desses commits e remove o commit “Add cat-file” completamente.

Machine Translated by Google

Squashing Commits

Também é possível pegar uma série de commits e esmagá-los em um único commit com a ferramenta interativa de rebase. O script coloca instruções úteis na mensagem de rebase:

```
#
# Comandos:
# p, pick <commit> = use commit # r,
reword <commit> = use commit, mas edite a mensagem de commit # e, edit <commit>
= use commit, mas pare para corrigir # s, squash <commit> = use commit, mas funde-
se com o commit anterior # f, fixup <commit> = como "squash", mas descarta a
mensagem de log deste commit # x, exec <command> = executa o comando (o resto da linha) usando
o shell # b, break = pare aqui (continue rebase mais tarde com 'git rebase --continue') # d, drop
<commit> = remove commit # l, label <label> = label HEAD atual com um nome # t, reset <label> =
reset HEAD para a label # m, merge [-C <commit> | -c <commit>] <label> [# <oneline>] # .




                    crie um merge commit usando a mensagem do merge commit original (ou
# .                 o oneline, se nenhum merge commit original foi especificado). Use -c
# .                 <commit> para reformular a mensagem de confirmação.
#
# Estas linhas podem ser reordenadas; eles são executados de cima para baixo. # # Se você
remover uma linha aqui ESSE COMMIT SERÁ PERDIDO. #



# No entanto, se você remover tudo, o rebase será abortado. # # Observe que os
commits vazios são comentados
```

Se, ao invés de “pick” ou “edit”, você especificar “squash”, o Git aplicará tanto a alteração quanto a alteração diretamente anterior a ela e fará com que você mescle as mensagens do commit. Então, se você quiser fazer um único commit a partir desses três commits, faça o script ficar assim:

```
escolha f7f3f6d Mude meu nome um pouco
squash 310154e Atualize a formatação README e adicione a culpa
squash a5f4a0d Adicionar arquivo cat
```

Quando você salva e sai do editor, o Git aplica todas as três alterações e, em seguida, coloca você de volta no editor para mesclar as três mensagens de confirmação:

```
# Esta é uma combinação de 3 commits.
# A mensagem do primeiro commit é:
Mude um pouco meu nome

# Esta é a segunda mensagem do commit:

Atualize a formatação README e adicione culpa
```

Machine Translated by Google

```
# Esta é a terceira mensagem de confirmação:

Adicionar arquivo cat
```

Quando você salva isso, você tem um único commit que introduz as mudanças de todos os três commits anteriores.

Dividindo um Commit

Dividir um commit desfaz um commit e, em seguida, faz estágios e commits parciais quantas vezes quiser. Por exemplo, suponha que você queira dividir o commit do meio de seus três commits. Em vez de "Atualizar a formatação do README e adicionar culpa", você deseja dividi-lo em dois commits: "Atualizar a formatação do README" para o primeiro e "Adicionar culpa" para o segundo. Você pode fazer isso no script rebase -i alterando a instrução no commit que deseja dividir para "editar":

```
escolha f7f3f6d Mude um pouco meu nome
edite 310154e Atualize a formatação README e adicione a culpa pick
a5f4a0d Adicionar arquivo cat
```

Então, quando o script leva você para a linha de comando, você redefine esse commit, pega as alterações que foram redefinidas e cria vários commits a partir delas. Quando você salva e sai do editor, o Git retrocede para o pai do primeiro commit em sua lista, aplica o primeiro commit (f7f3f6d), aplica o segundo (310154e) e leva você ao console. Lá, você pode fazer uma redefinição mista desse commit com git reset HEAD^, que efetivamente desfaz esse commit e deixa os arquivos modificados fora do estágio. Agora você pode testar e confirmar arquivos até ter vários commits e executar git rebase --continue quando terminar:

```
$ git reset HEAD^ $ git
add README $ git
commit -m 'Atualizar a formatação do README' $ git add
lib/simplegit.rb $ git commit -m 'Adicionar culpa' $ git rebase
--continue
```

O Git aplica o último commit (a5f4a0d) no script, e seu histórico fica assim:

```
$ git log -4 --pretty=format:"%h %s" 1c002dd
Adicionar arquivo cat 9b29157 Adicionar culpa

35cfb2b Atualizar a formatação README
f7f3f6d Mudar um pouco meu nome
```

Isso altera os SHA-1s dos três commits mais recentes em sua lista, portanto, certifique-se de que nenhum commit alterado apareça na lista que você já enviou para um repositório compartilhado. Observe que o último

Machine Translated by Google

commit (f7f3f6d) na lista permanece inalterado. Apesar deste commit ser mostrado no script, porque foi marcado como "pick" e foi aplicado antes de qualquer mudança de rebase, o Git deixa o commit inalterado.

Excluindo um commit

Se você quiser se livrar de um commit, você pode excluí-lo usando o script rebase -i . Na lista de commits, coloque a palavra "drop" antes do commit que deseja deletar (ou simplesmente delete essa linha do script de rebase):

```
pick 461cb2a Este commit está OK
drop 5aecc10 Este commit está quebrado
```

Devido à maneira como o Git cria objetos de commit, excluir ou alterar um commit causará a reescrita de todos os commits que o seguem. Quanto mais você recuar no histórico do repositório, mais confirmações precisarão ser recriadas. Isso pode causar muitos conflitos de mesclagem se você tiver muitos commits posteriormente na sequência que dependem daquele que você acabou de deletar.

Se você chegar no meio de um rebase como este e decidir que não é uma boa ideia, você sempre pode parar. Digite git rebase --abort e seu repositório retornará ao estado em que estava antes de iniciar o rebase.

Se você terminar um rebase e decidir que não é o que deseja, pode usar git reflog para recuperar uma versão anterior de sua ramificação. Consulte Recuperação de dados para obter mais informações sobre o comando reflog .

ÿ Drew DeVault fez um guia prático com exercícios para aprender a usar git rebase. Você pode encontrá-lo em: https://git-rebase.io/

A opção nuclear: ramo-filtro

Existe outra opção de reescrita de histórico que você pode usar se precisar reescrever um número maior de commits de alguma forma programável — por exemplo, alterando seu endereço de e-mail globalmente ou removendo um arquivo de cada commit. O comando é filter-branch e pode reescrever grandes trechos do seu histórico, então você provavelmente não deve usá-lo, a menos que seu projeto ainda não seja público e outras pessoas não tenham baseado o trabalho nos commits que você está prestes a reescrever . No entanto, pode ser muito útil. Você aprenderá alguns dos usos comuns para ter uma ideia de algumas das coisas de que ele é capaz.

ÿ git filter-branch tem muitas armadilhas e não é mais a maneira recomendada de reescrever o histórico. Em vez disso, considere usar git-filter-repo, que é um script Python que faz um trabalho melhor para a maioria dos aplicativos em que você normalmente usaria filter-branch. Sua documentação e código-fonte podem ser encontrados em

https://github.com/newren/git-filter-repo.

Removendo um arquivo de cada commit

Isso ocorre com bastante frequência. Alguém acidentalmente comete um enorme arquivo binário com um git add impensado, e você deseja removê-lo de todos os lugares. Talvez você acidentalmente tenha cometido um arquivo que

Machine Translated by Google

continha uma senha e você deseja tornar seu projeto de código aberto. filter-branch é a ferramenta que você provavelmente deseja usar para limpar todo o seu histórico. Para remover um arquivo chamado passwords.txt de todo o seu histórico, você pode usar a opção --tree-filter para filter-branch:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD Rewrite
6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' foi reescrito
```

A opção --tree-filter executa o comando especificado após cada check-out do projeto e, em seguida, confirma novamente os resultados. Nesse caso, você remove um arquivo chamado passwords.txt de cada instantâneo, quer ele exista ou não. Se você deseja remover todos os arquivos de backup do editor comprometidos acidentalmente, pode executar algo como git filter-branch --tree-filter 'rm -f *~' HEAD.

Você poderá observar o Git reescrevendo árvores e commits e, em seguida, mover o ponteiro do branch no final. Geralmente, é uma boa ideia fazer isso em um branch de teste e, em seguida, reinicializar seu branch master depois de determinar que o resultado é o que você realmente deseja. Para executar filter-branch em todos os seus branches, você pode passar --all para o comando.

Tornando um subdiretório a nova raiz

Suponha que você tenha feito uma importação de outro sistema de controle de origem e tenha subdiretórios que não fazem sentido (tronco, tags e assim por diante). Se você quiser fazer com que o subdiretório trunk seja a nova raiz do projeto para cada commit, filter-branch pode ajudá-lo a fazer isso também:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Reescrever 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' foi reescrito
```

Agora, sua nova raiz do projeto é o que estava no subdiretório trunk todas as vezes. O Git também removerá automaticamente os commits que não afetaram o subdiretório.

Alteração de endereços de e-mail globalmente

Outro caso comum é que você esqueceu de executar git config para definir seu nome e endereço de e-mail antes de começar a trabalhar, ou talvez queira abrir o código de um projeto no trabalho e alterar todos os seus endereços de e-mail de trabalho para seu endereço pessoal. Em qualquer caso, você também pode alterar endereços de e-mail em vários commits em um lote com filter-branch . Você precisa ter cuidado para alterar apenas os endereços de e-mail que são seus, então use --commit-filter:

```
$ git filter-branch --commit-filter if [ "$GIT_AUTHOR_EMAIL"
        = "schacon@localhost" ]; então

                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com"; git commit-tree
                "$@";
        outro
                git commit-tree "$@";
```

Machine Translated by Google

```
cabeça de fi'
```

Isso passa e reescreve cada confirmação para ter seu novo endereço. Como os commits contêm os valores SHA-1 de seus pais, esse comando altera todos os SHA-1 de commit em seu histórico, não apenas aqueles que têm o endereço de e-mail correspondente.

Reiniciar Desmistificado

Antes de passar para ferramentas mais especializadas, vamos falar sobre os comandos reset e checkout do Git .
Esses comandos são duas das partes mais confusas do Git quando você os encontra pela primeira vez. Eles fazem tantas coisas que parece impossível realmente entendê-los e empregá-los adequadamente.
Para isso, recomendamos uma metáfora simples.

As Três Árvores

Uma maneira mais fácil de pensar em redefinir e finalizar a compra é por meio da estrutura mental do Git como um gerenciador de conteúdo de três árvores diferentes. Por “árvore” aqui, realmente queremos dizer “coleção de arquivos”, não especificamente a estrutura de dados. Existem alguns casos em que o índice não age exatamente como uma árvore, mas para nossos propósitos é mais fácil pensar dessa forma por enquanto.

O Git como um sistema gerencia e manipula três árvores em sua operação normal:

Árvore	Papel
CABEÇA	Instantâneo do último commit, próximo pai
Índice	Instantâneo do próximo commit proposto
Diretório de trabalho	Caixa de areia

A cabeça

HEAD é o ponteiro para a referência do branch atual, que por sua vez é um ponteiro para o último commit feito naquele branch. Isso significa que HEAD será o pai do próximo commit criado. Geralmente é mais simples pensar em HEAD como o instantâneo do **seu último commit naquele branch.**

Na verdade, é muito fácil ver a aparência desse instantâneo. Aqui está um exemplo de como obter a listagem de diretório real e as somas de verificação SHA-1 para cada arquivo no instantâneo HEAD:

```
$ git cat-file -p HEAD tree
cfda3bf379e4f8dba8717dee55aab78aef7f4daf autor Scott
Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

confirmação inicial

$ git ls-tree -r HEAD 100644
blob a906cb2a4a904a152... README 100644 blob
8f94139338f9404f2... Rakefile
```

Machine Translated by Google

```
040000 árvore 99f1a6d12cb4b6f19... lib
```

Os comandos Git cat-file e ls-tree são comandos de “encanamento” usados para coisas de nível inferior e não muito usados no trabalho do dia-a-dia, mas eles nos ajudam a ver o que está acontecendo aqui.

O índice

O *índice* é seu **próximo commit proposto.** Também nos referimos a esse conceito como a “área de preparação” do Git, pois é isso que o Git vê quando você executa o git commit.

O Git preenche esse índice com uma lista de todo o conteúdo do arquivo cujo check-out foi feito pela última vez em seu diretório de trabalho e sua aparência quando foi feito o check-out originalmente. Em seguida, você substitui alguns desses arquivos por novas versões deles e git commit converte isso na árvore para um novo commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0
README 100644 8f94139338f9404f26296befa88755fc2598c289
0 Rakefile 100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Novamente, aqui estamos usando git ls-files, que é mais um comando de bastidores que mostra a aparência atual do seu índice.

O índice não é tecnicamente uma estrutura de árvore — na verdade, ele é implementado como um manifesto nivelado — mas, para nossos propósitos, é próximo o suficiente.

O diretório de trabalho

Finalmente, você tem seu *diretório de trabalho* (também conhecido como “árvore de trabalho”). As outras duas árvores armazenam seu conteúdo de forma eficiente, mas inconveniente, dentro da pasta .git .
O diretório de trabalho os descompacta em arquivos reais, o que torna muito mais fácil para você editá-los. Pense no diretório de trabalho como uma **caixa de proteção,** onde você pode experimentar as alterações antes de enviá-las para sua área de preparação (índice) e depois para o histórico.

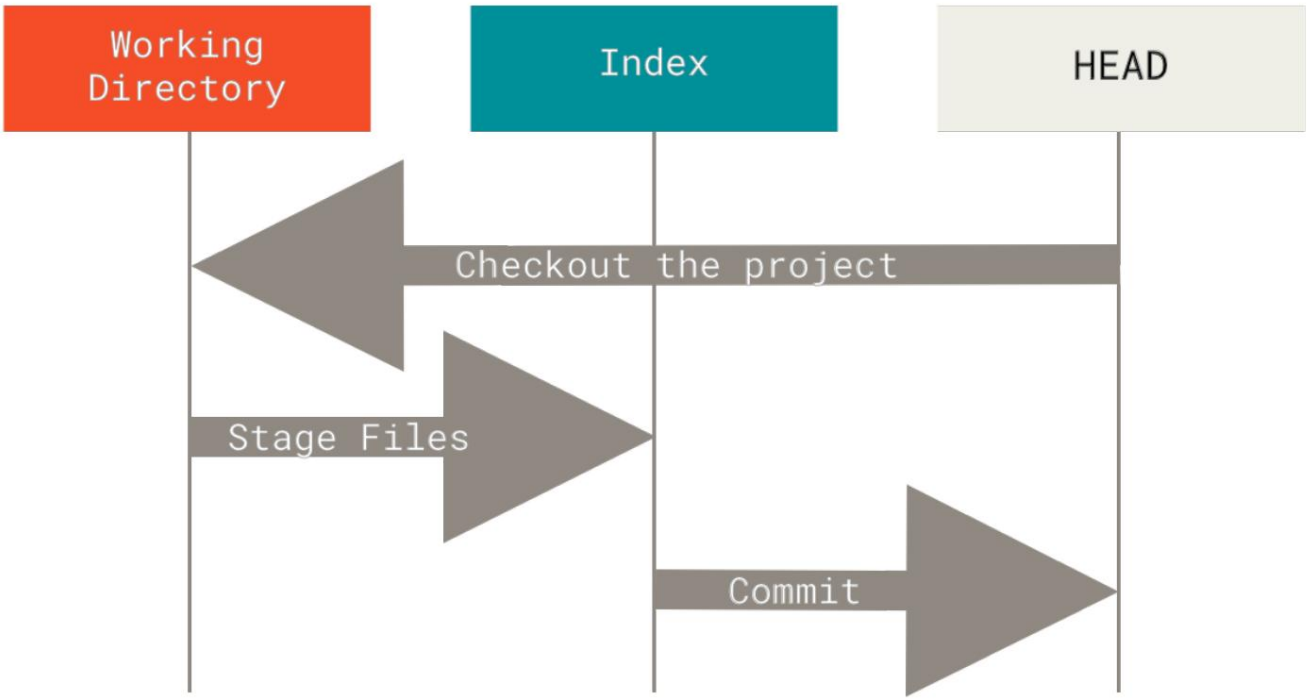
```
$ árvore
.
 ÿÿÿ README
 ÿÿÿ Rakefile
 ÿÿÿ lib
        ÿÿÿ simplegit.rb

1 diretório, 3 arquivos
```

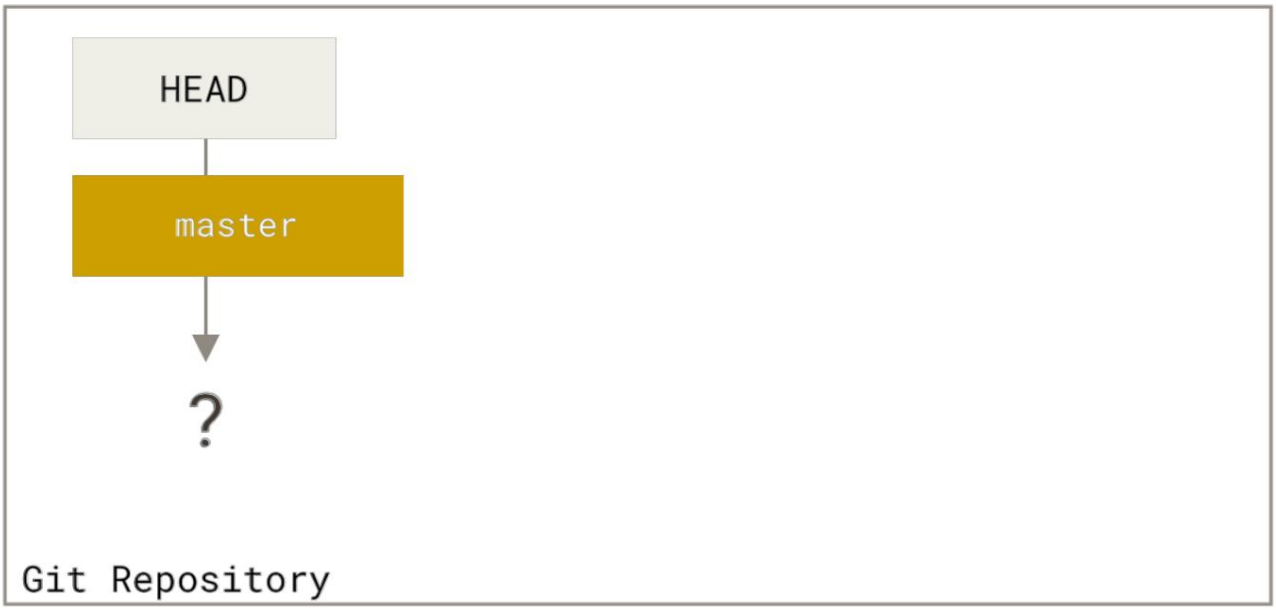
O fluxo de trabalho

O fluxo de trabalho típico do Git é registrar instantâneos de seu projeto em estados sucessivamente melhores, manipulando essas três árvores.

Machine Translated by Google



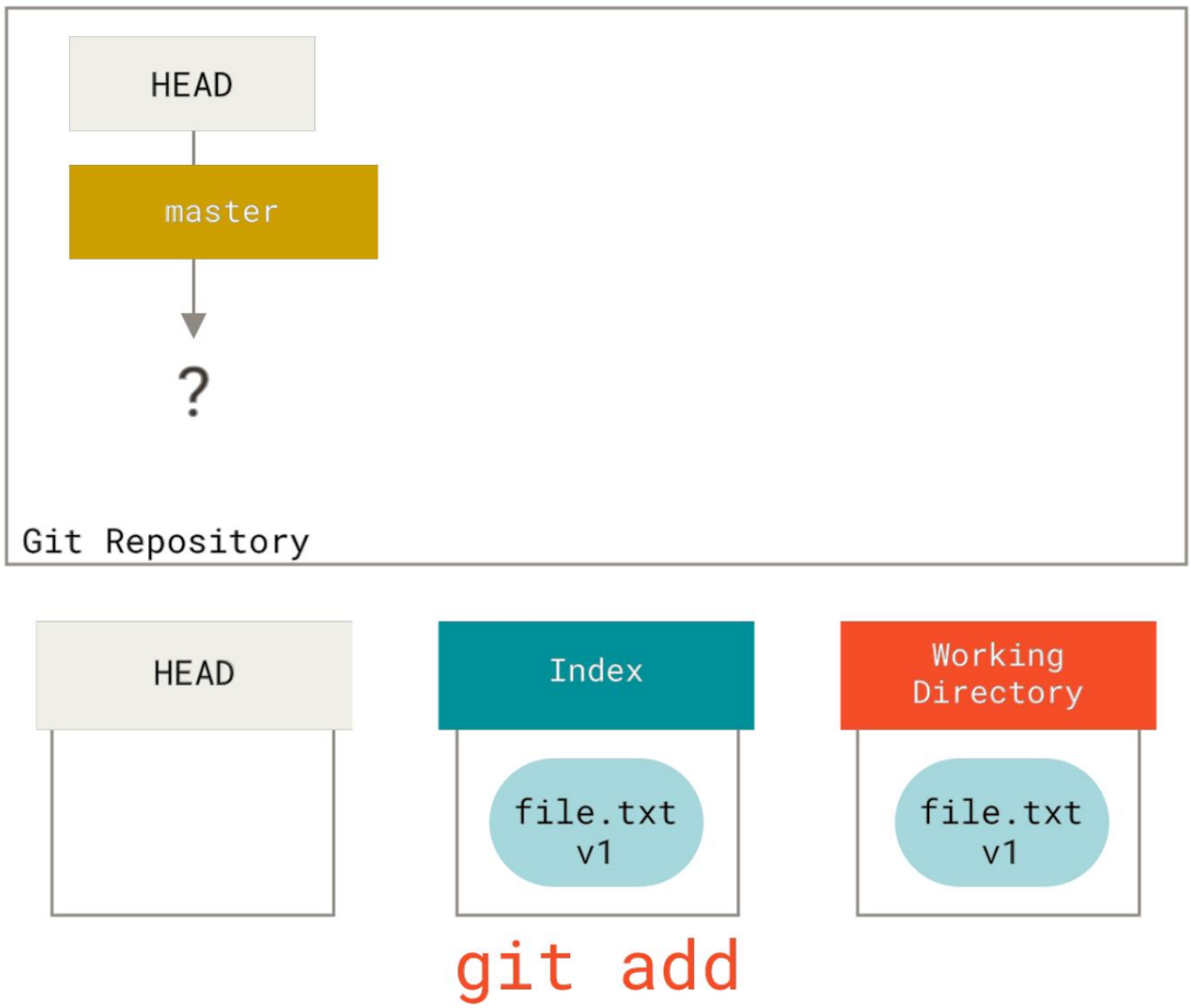
Vamos visualizar este processo: digamos que você entre em um novo diretório com um único arquivo. Chamaremos isso **de v1** do arquivo e o indicaremos em azul. Agora executamos o git init, que criará um repositório Git com uma referência HEAD que aponta para o ramo mestre não nascido .



Machine Translated by Google

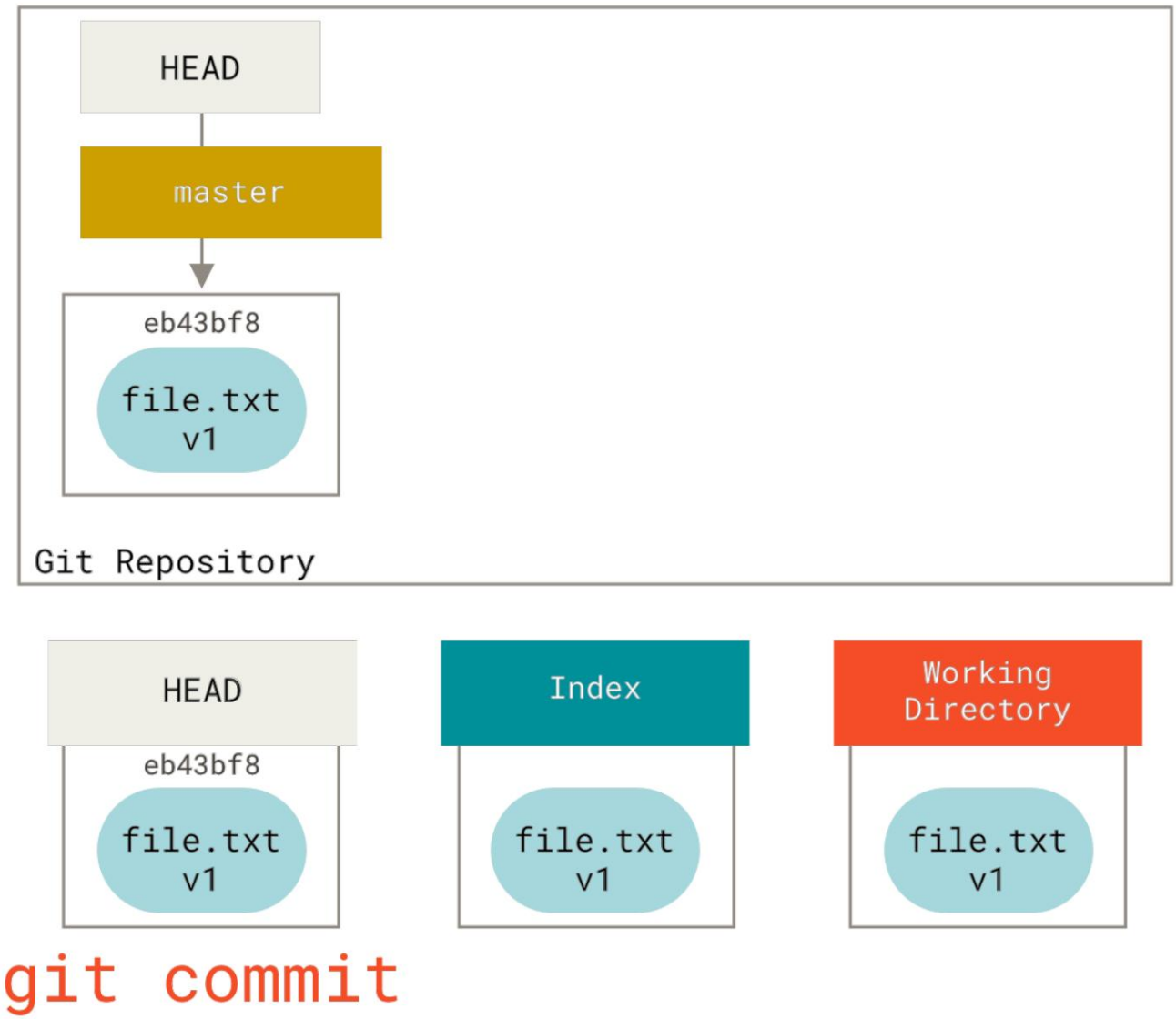
Neste ponto, apenas a árvore de diretórios de trabalho possui algum conteúdo.

Agora queremos confirmar este arquivo, então usamos git add para pegar o conteúdo no diretório de trabalho e copiá-lo para o índice.



Em seguida, executamos git commit, que pega o conteúdo do índice e o salva como um instantâneo permanente, cria um objeto commit que aponta para esse instantâneo e atualiza o master para apontar para esse commit.

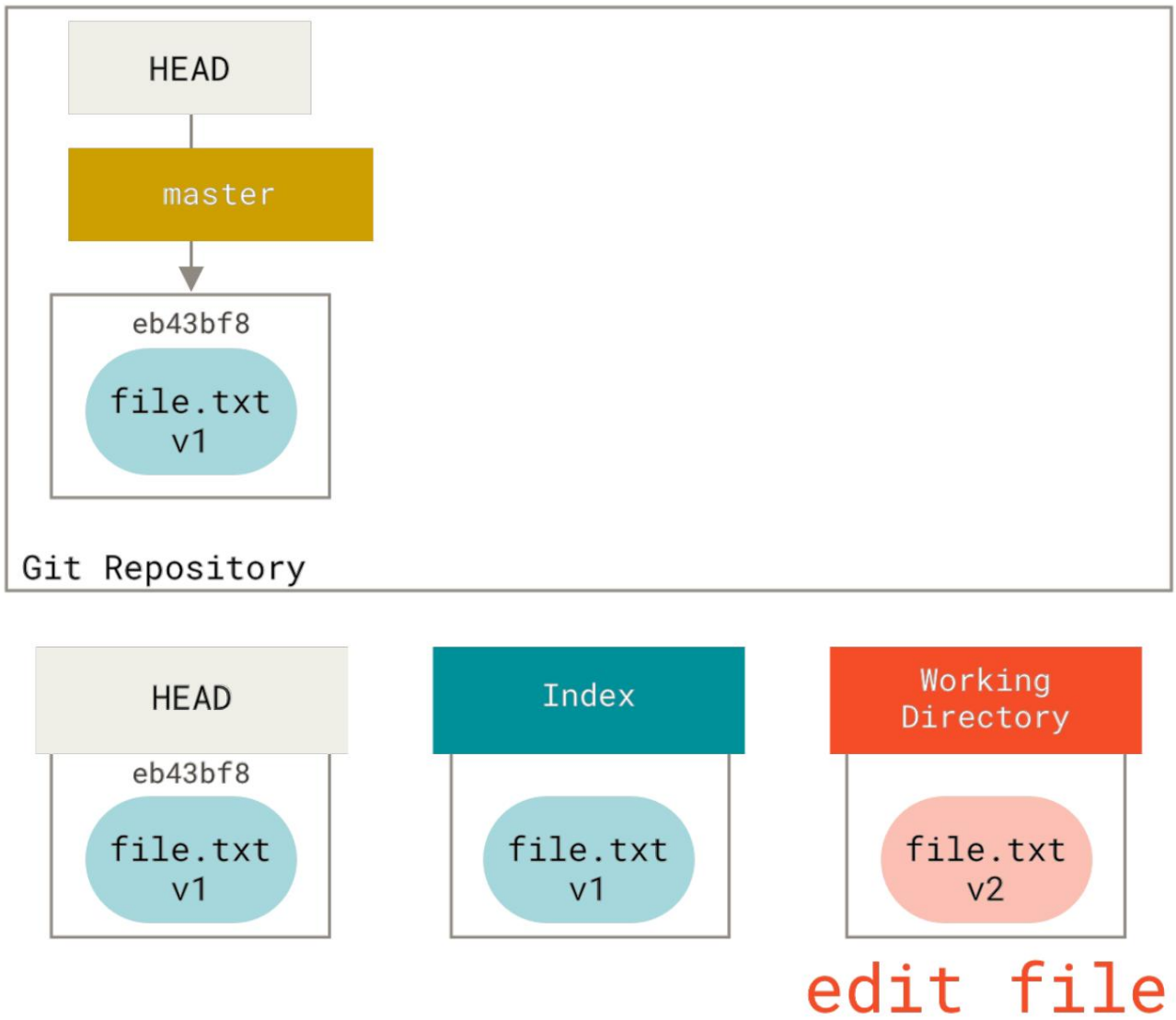
Machine Translated by Google



Se executarmos git status, não veremos alterações, porque todas as três árvores são iguais.

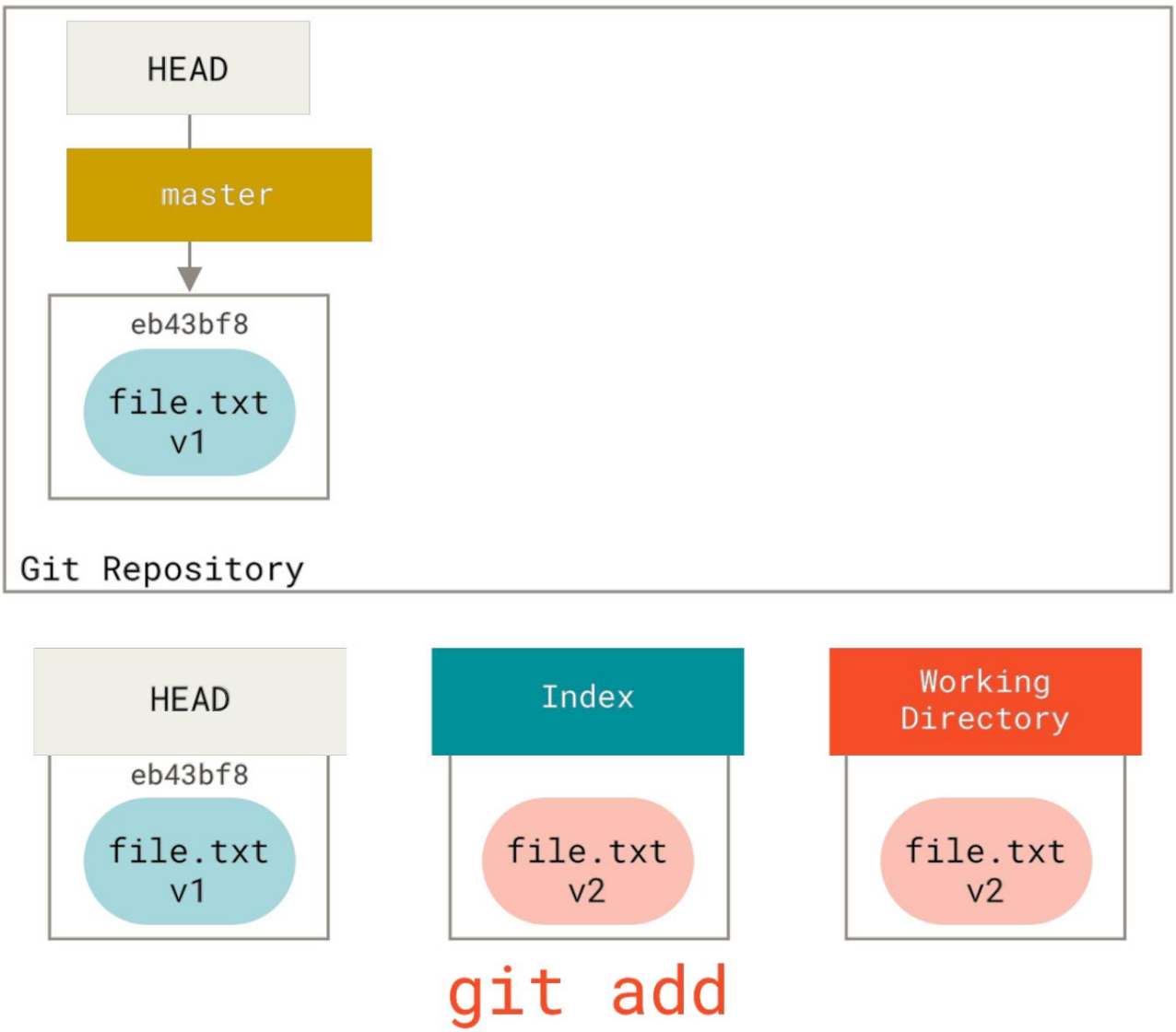
Agora queremos fazer uma alteração nesse arquivo e confirmá-lo. Passaremos pelo mesmo processo; primeiro, alteramos o arquivo em nosso diretório de trabalho. Vamos chamar isso **de v2** do arquivo e indicá-lo em vermelho.

Machine Translated by Google



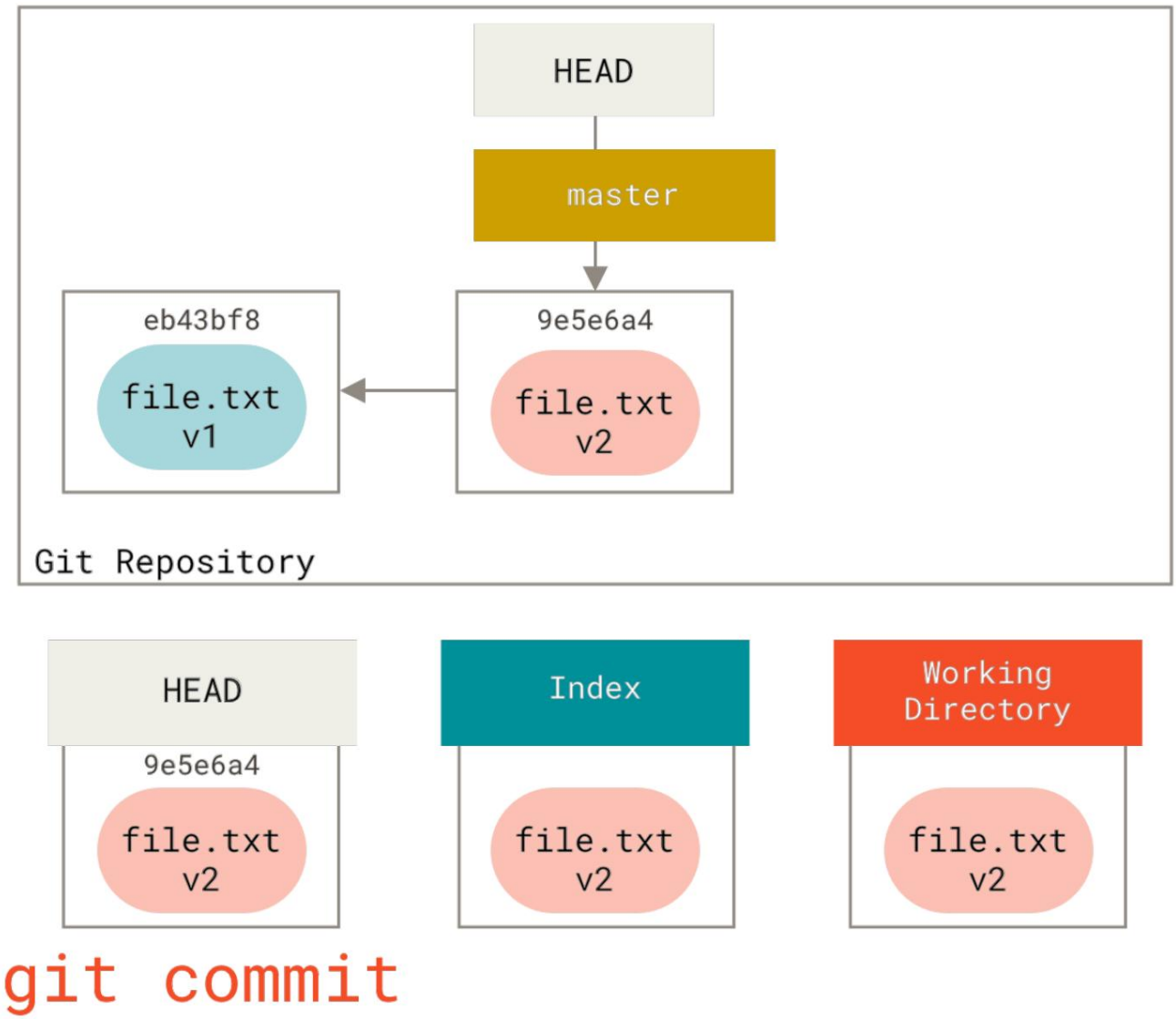
Se executarmos git status agora, veremos o arquivo em vermelho como “Changes not staged for commit”, porque essa entrada difere entre o índice e o diretório de trabalho. Em seguida, executamos git add nele para colocá-lo em nosso índice.

Machine Translated by Google



Neste ponto, se executarmos git status, veremos o arquivo em verde em “Changes to be commited” porque o índice e o HEAD diferem — ou seja, nosso próximo commit proposto agora é diferente do nosso último commit. Por fim, executamos git commit para finalizar o commit.

Machine Translated by Google



Agora git status não nos dará nenhuma saída, porque todas as três árvores são as mesmas novamente.

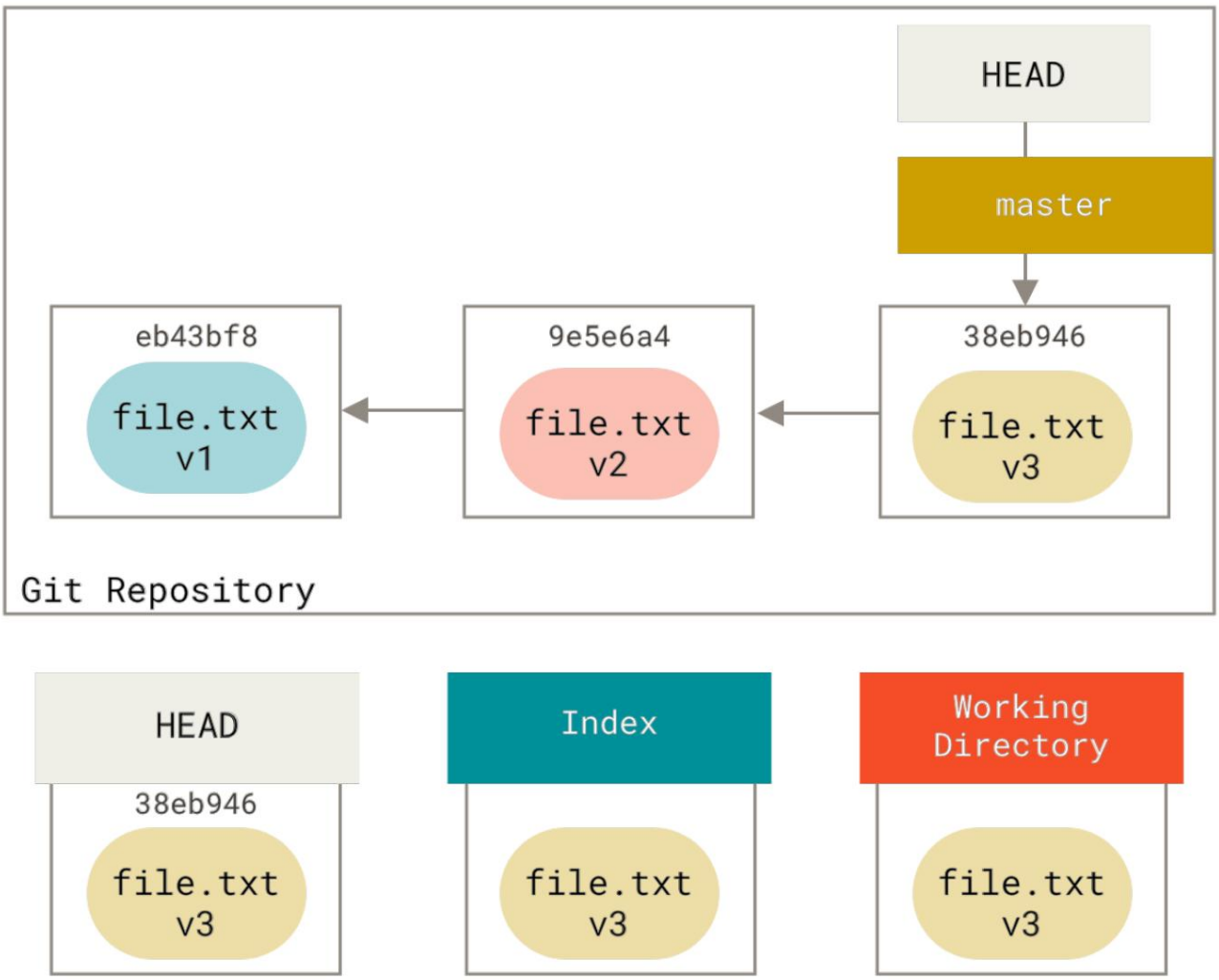
A troca de filiais ou a clonagem passam por um processo semelhante. Quando você faz checkout de um branch, ele muda **o HEAD** para apontar para o novo branch ref, preenche seu **índice** com o instantâneo daquele commit e então copia o conteúdo do **índice** em seu **diretório de trabalho.**

O papel da reinicialização

O comando reset faz mais sentido quando visto neste contexto.

Para fins desses exemplos, digamos que modificamos o arquivo.txt novamente e o confirmamos pela terceira vez. Então agora nossa história se parece com isso:

Machine Translated by Google

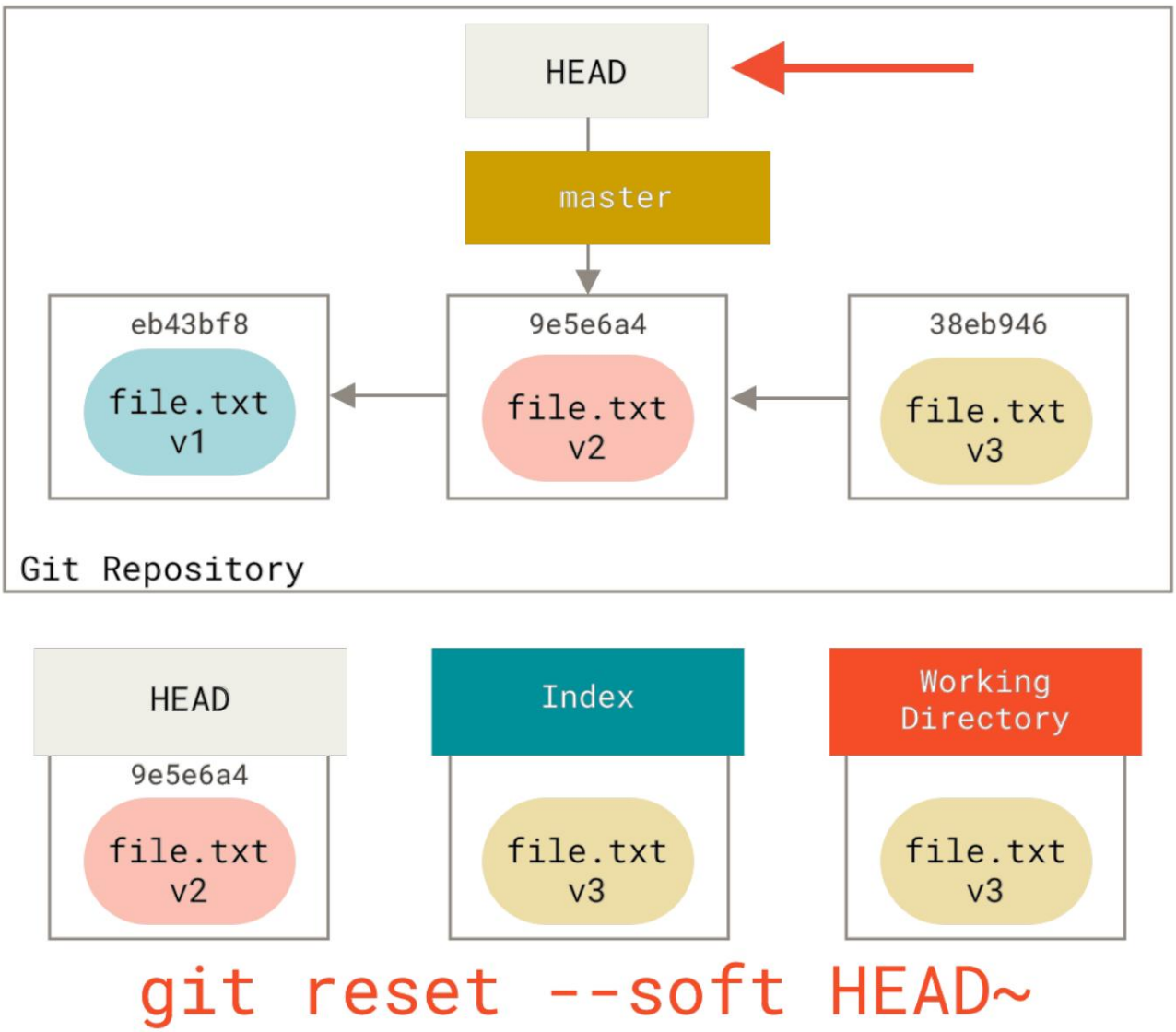


Vamos agora examinar exatamente o que reset faz quando você o chama. Ele manipula diretamente essas três árvores de forma simples e previsível. Faz até três operações básicas.

Passo 1: Mover HEAD

A primeira coisa que o reset fará é mover o que o HEAD aponta. Isso não é o mesmo que alterar o próprio HEAD (que é o que o checkout faz); reset move a ramificação para a qual HEAD está apontando. Isso significa que, se HEAD estiver definido como o branch master (ou seja, você está atualmente no branch master), a execução de git reset 9e5e6a4 começará fazendo o ponto master para 9e5e6a4.

Machine Translated by Google



Não importa qual forma de redefinição com um commit você invoca, esta é a primeira coisa que ele sempre tentará fazer. Com reset --soft, ele simplesmente parará por aí.

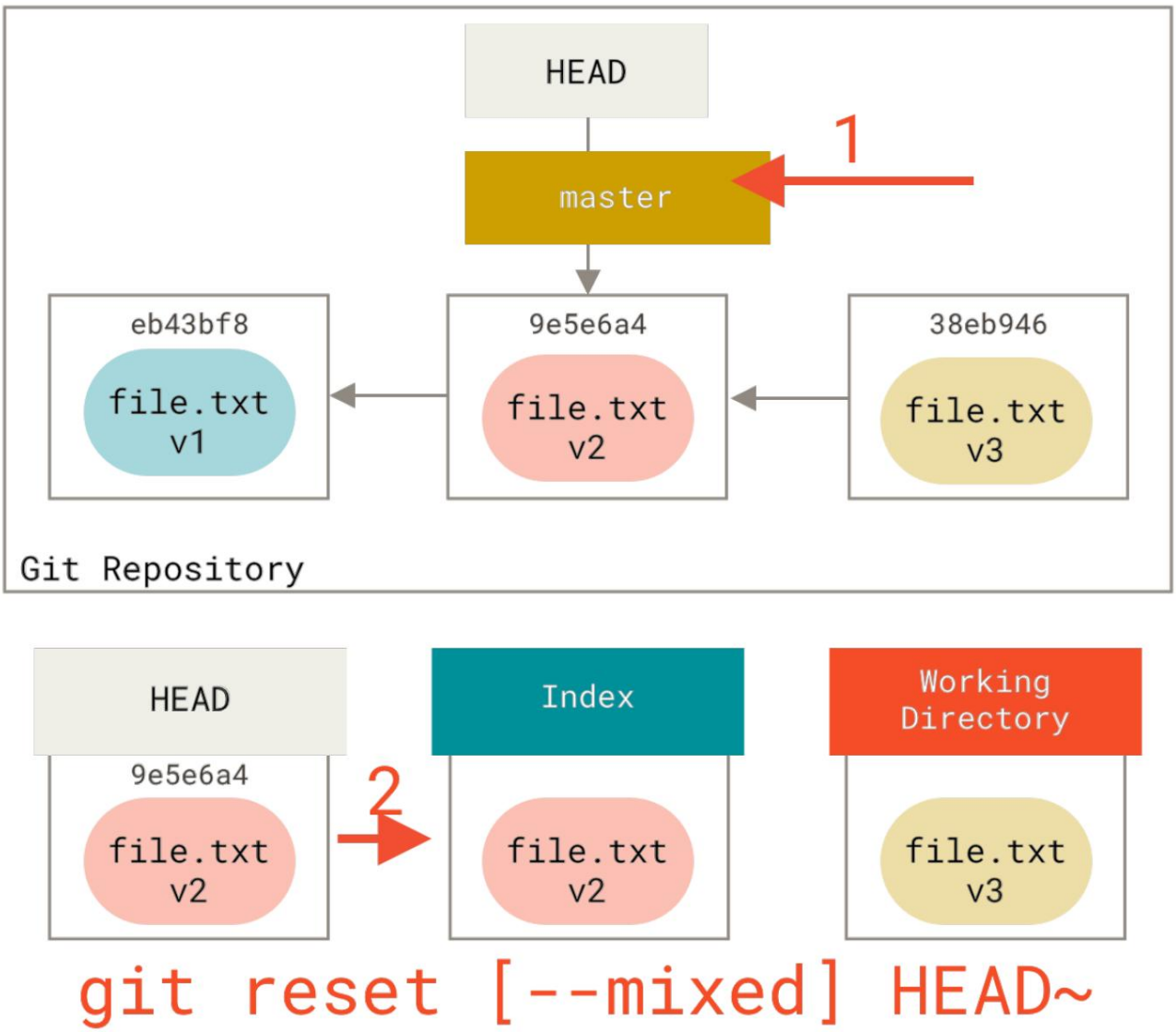
Agora, dedique um segundo para olhar para esse diagrama e perceber o que aconteceu: ele basicamente desfez o último comando git commit . Quando você executa o git commit, o Git cria um novo commit e move o branch para o qual o HEAD aponta para ele. Ao redefinir para HEAD~ (o pai de HEAD), você está movendo a ramificação de volta para onde estava, sem alterar o índice ou o diretório de trabalho. Agora você pode atualizar o índice e executar git commit novamente para realizar o que git commit --amend teria feito (consulte Alterando o último commit).

Passo 2: Atualizando o índice (--mixed)

Observe que, se você executar git status agora, verá em verde a diferença entre o índice e o que é o novo HEAD.

A próxima coisa que a redefinição fará é atualizar o índice com o conteúdo de qualquer instantâneo que o HEAD agora apontar.

Machine Translated by Google



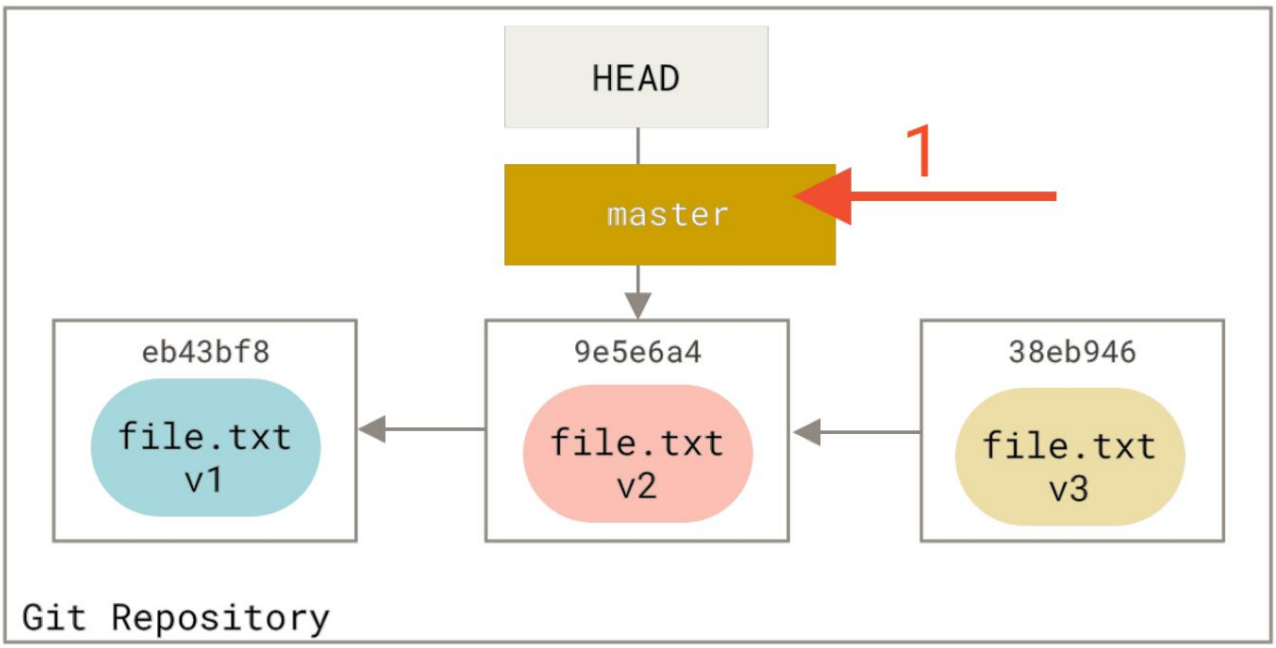
Se você especificar a opção --mixed , a redefinição será interrompida neste ponto. Este também é o padrão, portanto, se você não especificar nenhuma opção (apenas git reset HEAD~ neste caso), é aqui que o comando irá parar.

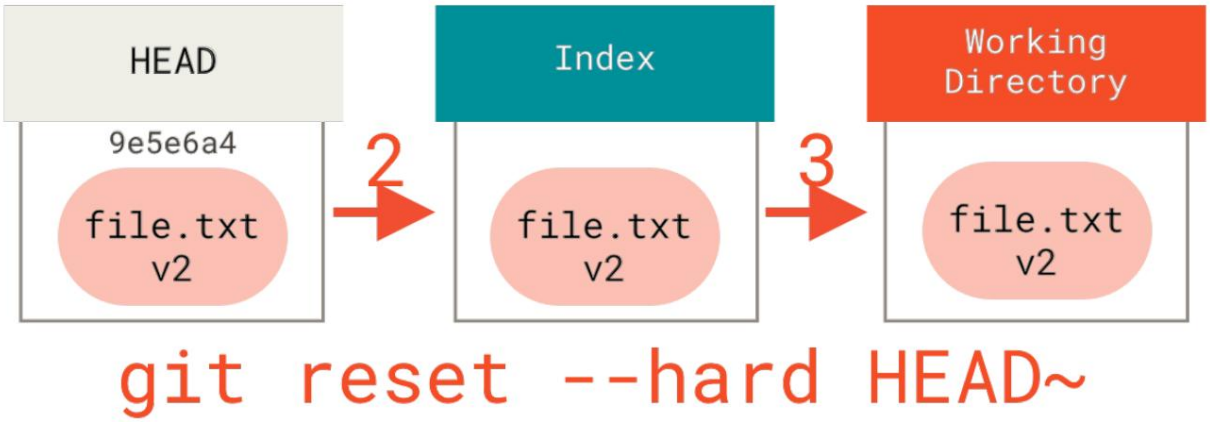
Agora, dedique mais um segundo para olhar esse diagrama e perceba o que aconteceu: ele ainda desfez seu último commit, mas também *desfez* tudo. Você voltou para antes de executar todos os seus comandos git add e git commit .

Passo 3: Atualizando o Diretório de Trabalho (--hard)

A terceira coisa que a redefinição fará é tornar o diretório de trabalho parecido com o index. Se você usar a opção --hard , continuará neste estágio.

Machine Translated by Google





Então, vamos pensar sobre o que acabou de acontecer. Você desfez seu último commit, os comandos git add e git commit **e** todo o trabalho que você fez em seu diretório de trabalho.

É importante observar que esse sinalizador (--hard) é a única maneira de tornar o comando reset perigoso e um dos poucos casos em que o Git realmente destrói os dados. Qualquer outra invocação de redefinição pode ser facilmente desfeita, mas a opção --hard não pode, pois sobrescreve arquivos à força no diretório de trabalho. Neste caso particular, ainda temos a versão **v3** do nosso arquivo em um commit em nosso Git DB, e poderíamos recuperá-lo olhando nosso reflog, mas se não o tivéssemos feito o commit, o Git ainda teria sobrescrito o arquivo e seria irrecuperável.

Recapitular

O comando reset substitui essas três árvores em uma ordem específica, parando quando você diz para:

1. Mova os pontos de ramificação HEAD para *(pare aqui se --soft).*
2. Faça o índice parecer HEAD *(pare aqui, a menos que --hard).*
3. Faça com que o diretório de trabalho se pareça com o índice.

Machine Translated by Google

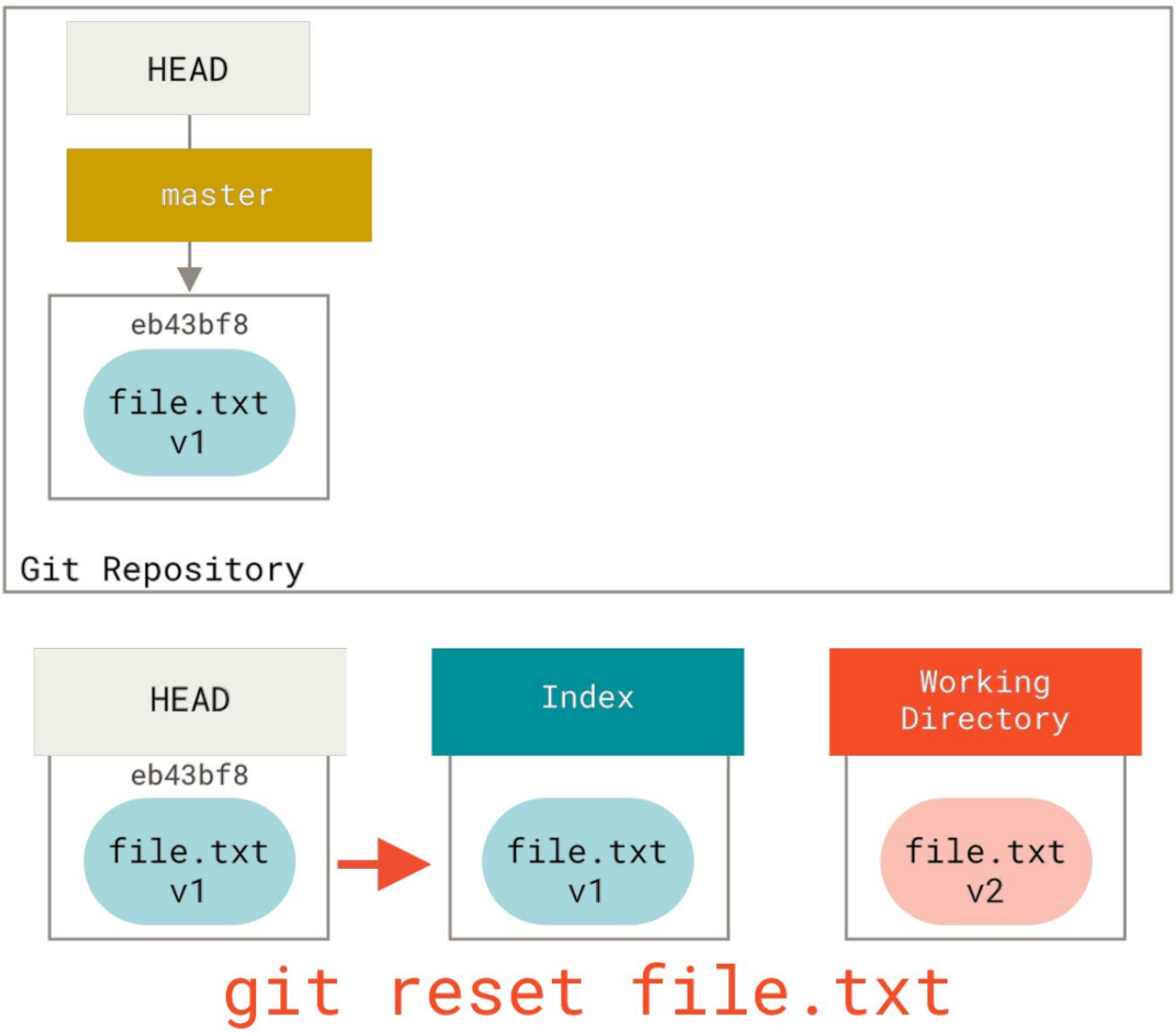
Redefinir com um caminho

Isso abrange o comportamento de redefinição em sua forma básica, mas você também pode fornecer um caminho para agir. Se você especificar um caminho, a redefinição ignorará a etapa 1 e limitará o restante de suas ações a um arquivo ou conjunto de arquivos específico. Isso realmente faz sentido - HEAD é apenas um ponteiro, e você não pode apontar para parte de um commit e parte de outro. Mas o índice e o diretório de trabalho *podem* ser parcialmente atualizados, portanto, a redefinição prossegue com as etapas 2 e 3.

Portanto, suponha que executamos git reset file.txt. Este formulário (já que você não especificou um commit SHA-1 ou ramificação, e você não especificou --soft ou --hard) é um atalho para git reset --mixed HEAD file.txt, que irá:

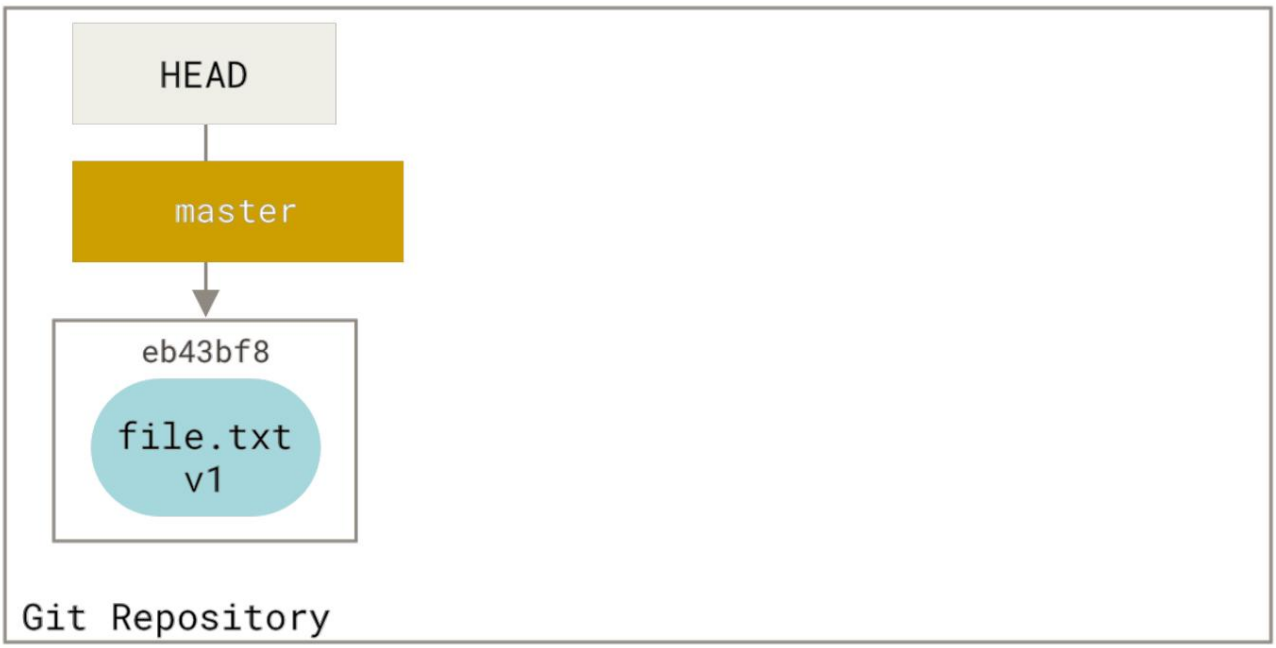
1. Mova os pontos HEAD da ramificação para *(ignorados).*
2. Faça o índice parecer HEAD *(pare aqui).*

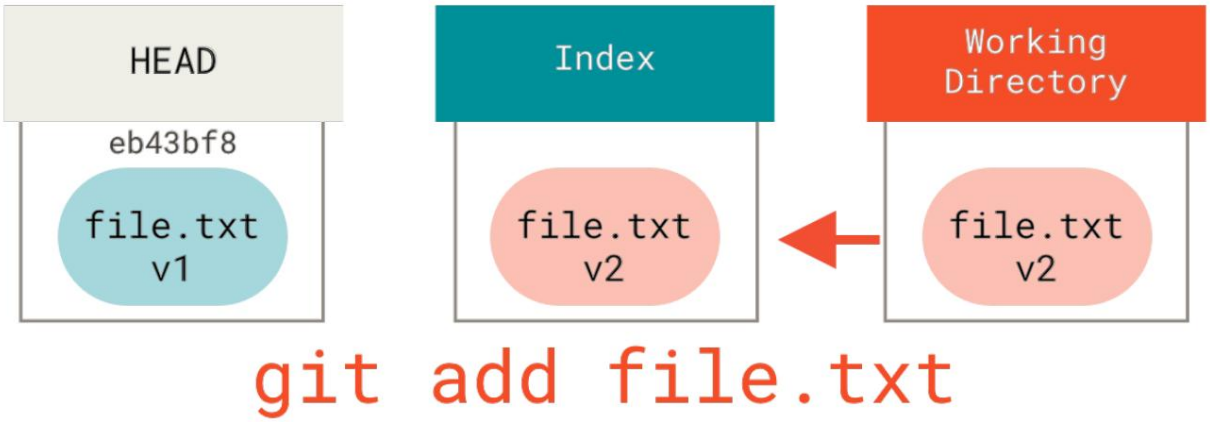
Então, basicamente, apenas copia file.txt de HEAD para o índice.



Isso tem o efeito prático de *desinstalar* o arquivo. Se olharmos para o diagrama desse comando e pensarmos sobre o que git add faz, eles são exatamente opostos.

Machine Translated by Google

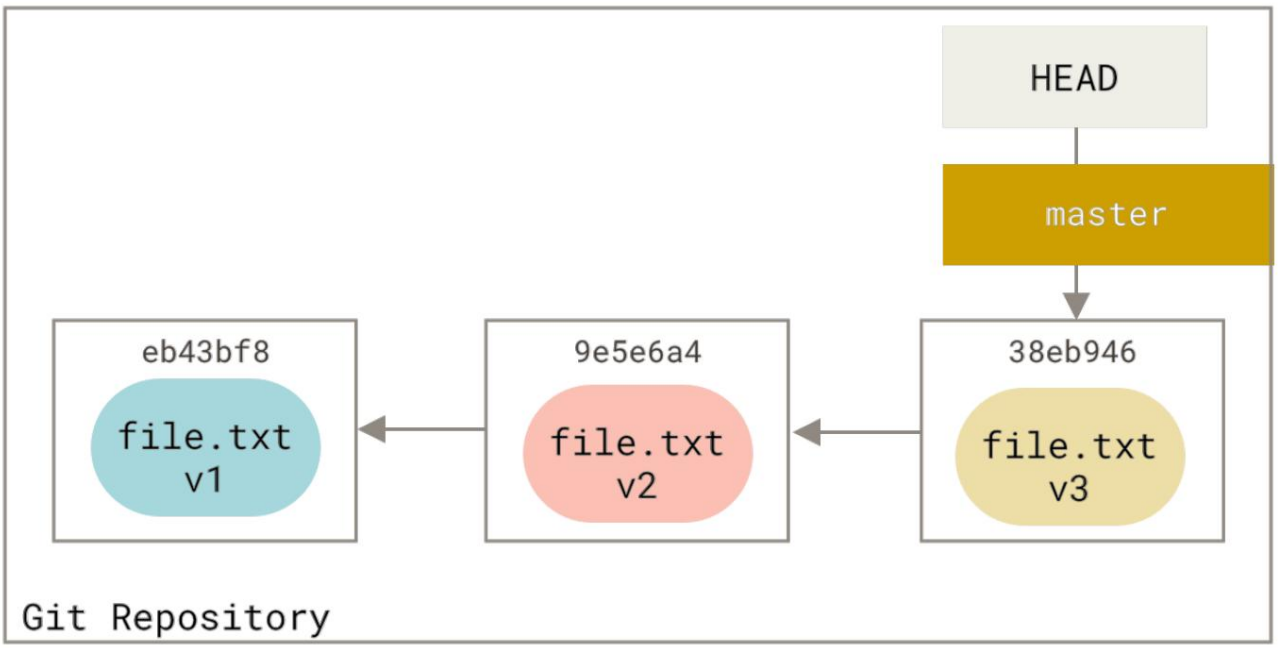




É por isso que a saída do comando git status sugere que você o execute para remover o estágio de um arquivo (consulte Desinstalando um arquivo preparado para saber mais sobre isso).

Poderíamos facilmente não deixar o Git presumir que quisemos dizer “puxar os dados do HEAD” especificando um commit específico para extrair essa versão do arquivo. Nós apenas executaríamos algo como git reset eb43bf file.txt.

Machine Translated by Google



```
git reset eb43 -- file.txt
```

Isso efetivamente faz a mesma coisa como se tivéssemos revertido o conteúdo do arquivo para **v1** no diretório de trabalho, executado git add nele e, em seguida, revertido para **v3** novamente (sem realmente passar por todas essas etapas). Se executarmos git commit agora, ele registrará uma alteração que reverte esse arquivo de volta para **v1,** mesmo que nunca o tenhamos em nosso diretório de trabalho novamente.

Também é interessante notar que, assim como o git add, o comando reset aceitará uma opção --patch para retirar o conteúdo do estágio de fragmento por fragmento. Assim, você pode remover seletivamente ou reverter o conteúdo.

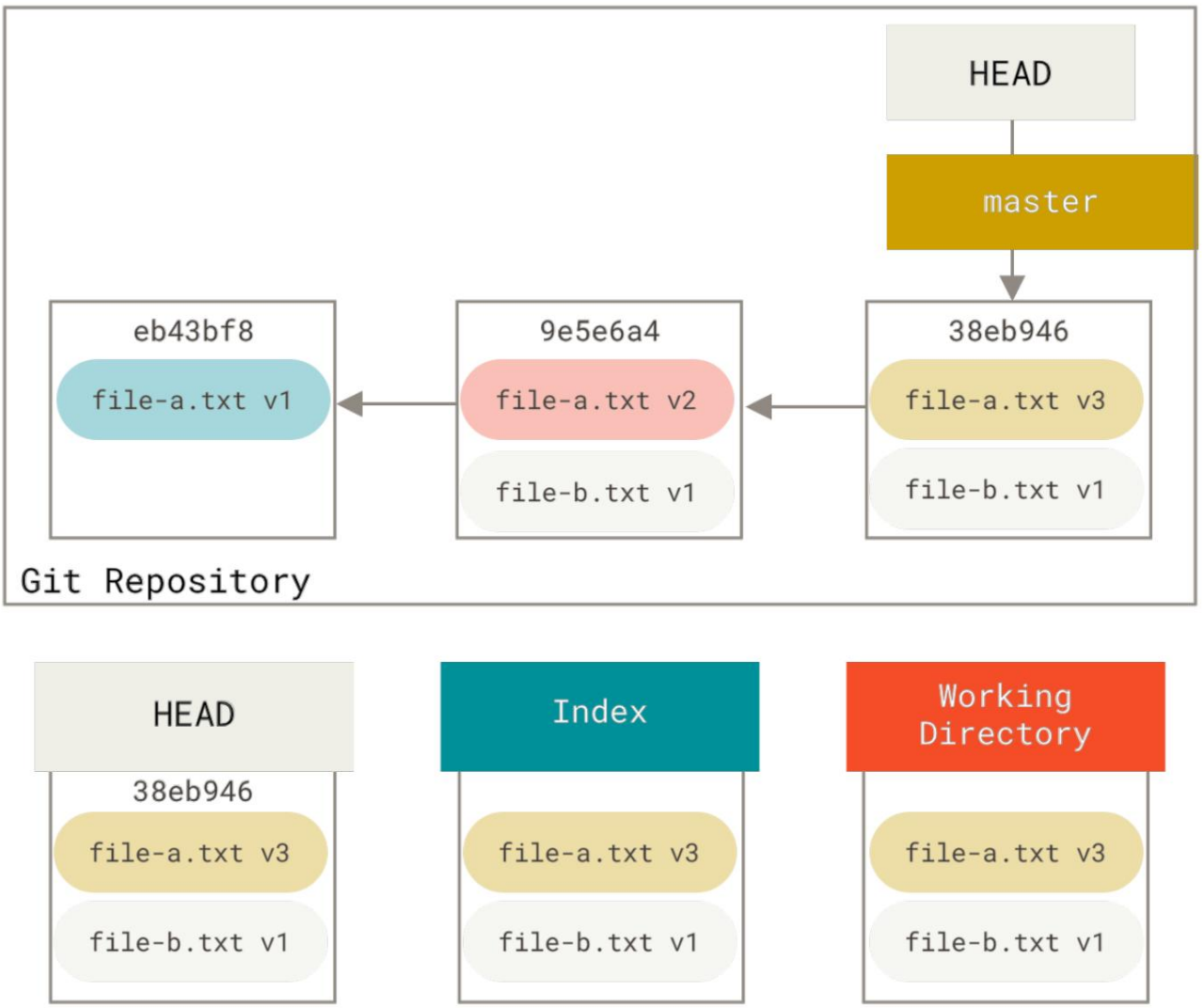
Squashing

Vejamos como fazer algo interessante com esse poder recém-descoberto — esmagar commits.

Digamos que você tenha uma série de commits com mensagens como “oops.”, “WIP” e “forgot this file”. Você pode usar o reset para esmagá-los de forma rápida e fácil em um único commit que faz você parecer realmente inteligente. Squashing Commits mostra outra maneira de fazer isso, mas neste exemplo é mais simples usar reset.

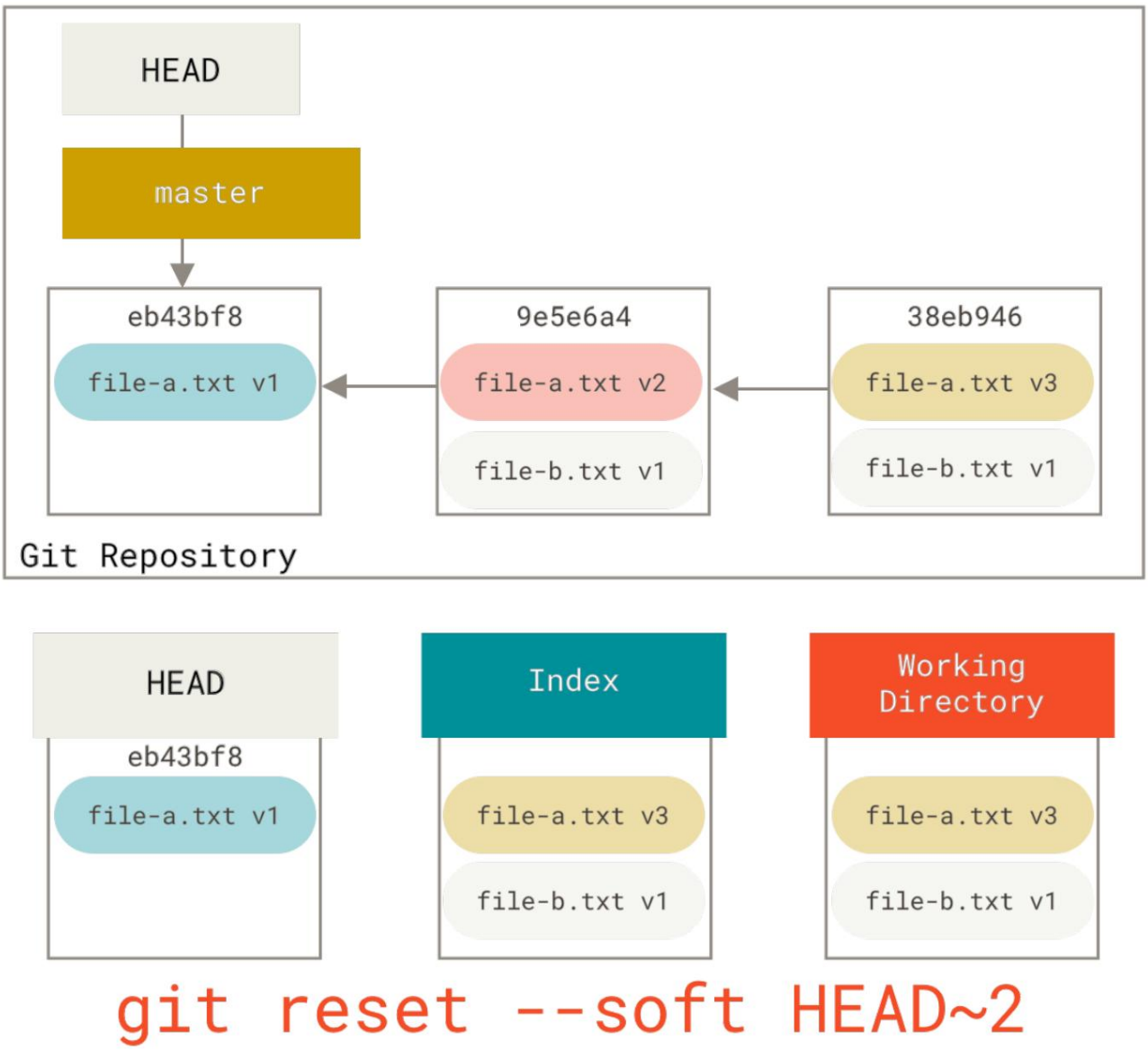
Digamos que você tenha um projeto onde o primeiro commit tem um arquivo, o segundo commit adicionou um novo arquivo e mudou o primeiro, e o terceiro commit mudou o primeiro arquivo novamente. O segundo commit foi um trabalho em andamento e você deseja esmagá-lo.

Machine Translated by Google



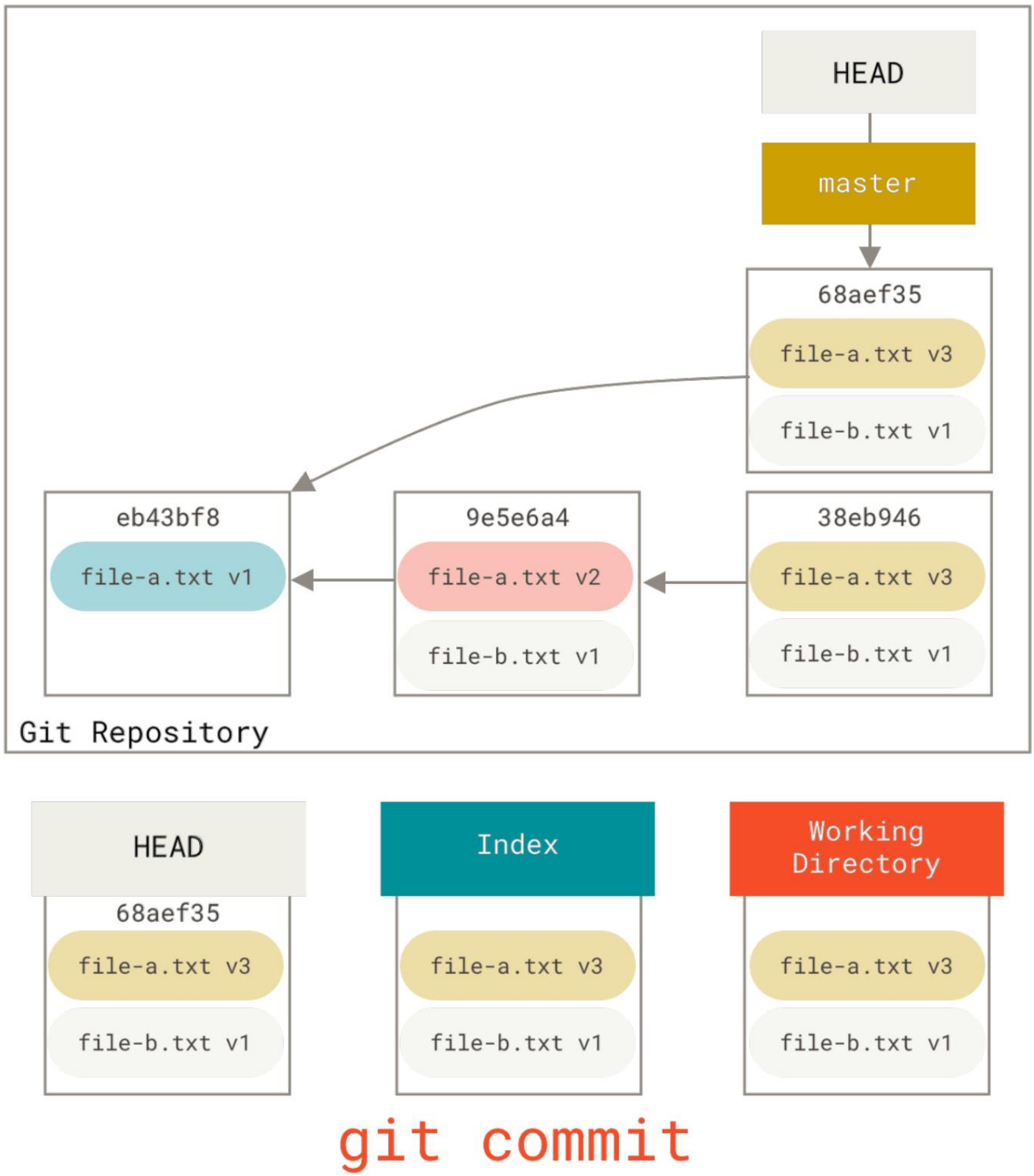
Você pode executar git reset --soft HEAD~2 para mover a ramificação HEAD de volta para um commit mais antigo (o commit mais recente que você deseja manter):

Machine Translated by Google



E então simplesmente execute git commit novamente:

Machine Translated by Google



Agora você pode ver que seu histórico alcançável, o histórico que você enviaria, agora parece que você tinha um commit com arquivo-a.txt v1, então um segundo que modificou o arquivo-a.txt para v3 e adicionou o arquivo b.txt. O commit com a versão v2 do arquivo não está mais no histórico.

Confira

Finalmente, você pode se perguntar qual é a diferença entre checkout e reset . Como reset, checkout manipula as três árvores, e é um pouco diferente dependendo se você dá ao comando um caminho de arquivo ou não.

Machine Translated by Google

sem caminhos

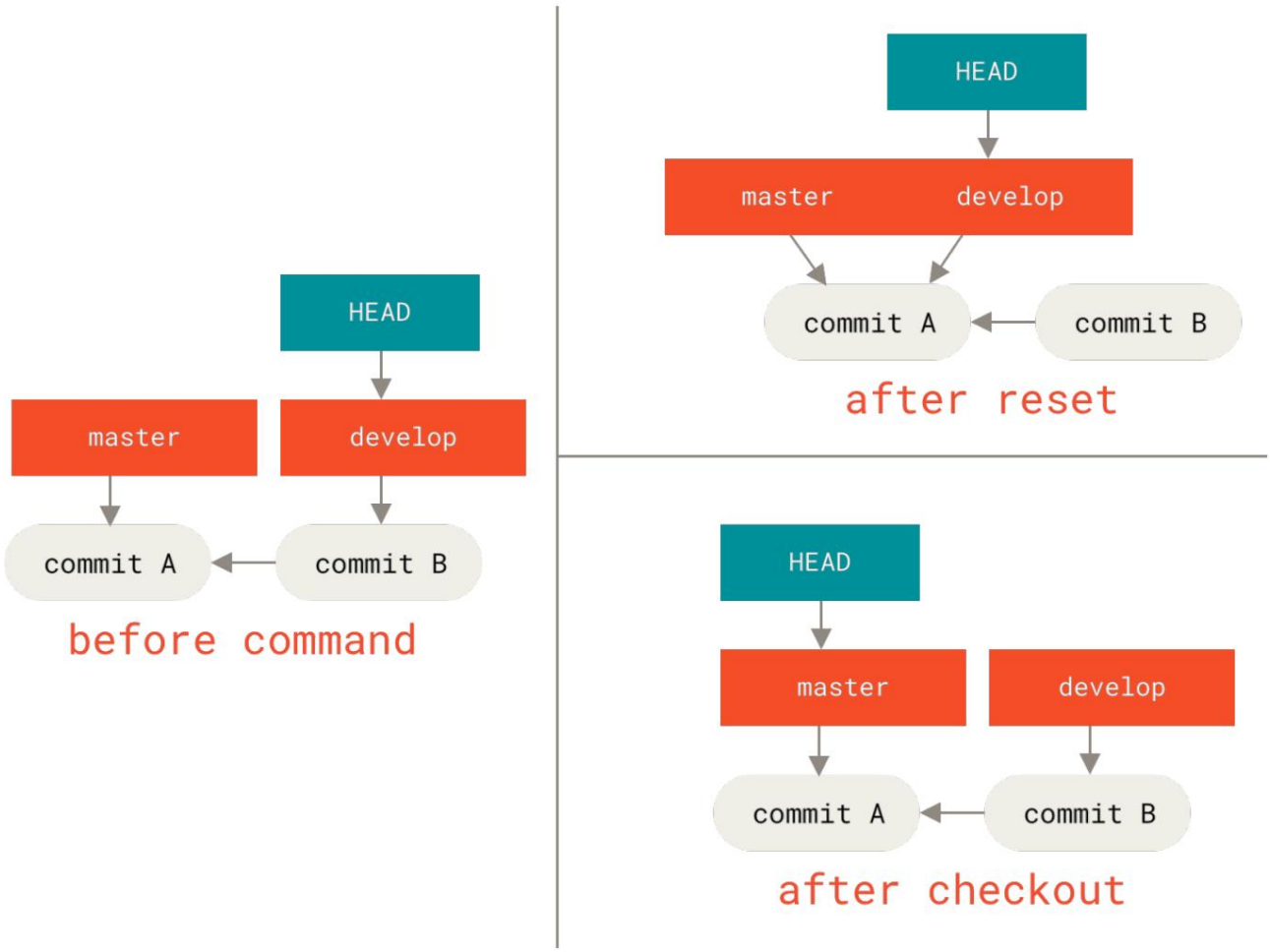
Executar git checkout [branch] é muito semelhante a executar git reset --hard [branch] no sentido de atualizar todas as três árvores para que você se pareça com [branch], mas há duas diferenças importantes.

Primeiro, ao contrário de reset --hard, o checkout é seguro no diretório de trabalho; ele verificará se não está eliminando os arquivos com alterações. Na verdade, é um pouco mais inteligente do que isso - ele tenta fazer uma mesclagem trivial no diretório de trabalho, para que todos os arquivos que você *não* alterou sejam atualizados. reset --hard, por outro lado, simplesmente substituirá tudo sem verificar.

A segunda diferença importante é como o checkout atualiza o HEAD. Considerando que reset moverá o branch para o qual HEAD aponta, checkout moverá o próprio HEAD para apontar para outro branch.

Por exemplo, digamos que temos ramificações master e develop que apontam para diferentes commits, e estamos atualmente em develop (então HEAD aponta para isso). Se executarmos git reset master, o próprio develop apontará para o mesmo commit que o master faz. Se, em vez disso, executarmos git checkout master, o develop não se move, o próprio HEAD sim. HEAD agora apontará para o mestre.

Então, em ambos os casos, estamos movendo HEAD para apontar para o commit A, mas *como* fazemos isso é muito diferente. redefinir moverá os pontos HEAD da ramificação, o checkout moverá o próprio HEAD.



Com Caminhos

A outra maneira de executar o checkout é com um caminho de arquivo, que, como reset, não move HEAD. É como o arquivo git reset [branch] na medida em que atualiza o índice com esse arquivo naquele commit, mas também

Machine Translated by Google

substitui o arquivo no diretório de trabalho. Seria exatamente como git reset --hard [branch] file (se reset permitir que você execute isso) - não é seguro para o diretório de trabalho e não move o HEAD.

Além disso, como git reset e git add, o checkout aceitará uma opção --patch para permitir que você reverta seletivamente o conteúdo do arquivo pedaço por pedaço.

Resumo

Espero que agora você entenda e se sinta mais confortável com o comando reset , mas provavelmente ainda está um pouco confuso sobre como exatamente ele difere do checkout e não poderia se lembrar de todas as regras das diferentes invocações.

Aqui está uma folha de dicas para quais comandos afetam quais árvores. A coluna “HEAD” lê “REF” se esse comando move a referência (ramificação) para a qual o HEAD aponta, e “HEAD” se move o próprio HEAD. Preste atenção especial ao 'WD Safe?' coluna — se disser **NÃO,** pense um pouco antes de executar esse comando.

	CABEÇA	**Índice**	**Workdir**	**WD seguro?**
Nível de compromisso				
redefinir --soft [commit]	REF	NÃO	NÃO	SIM
redefinir [comprometer]	REF	SIM	NÃO	SIM
reset --hard [commit]	REF	SIM	SIM	**NÃO**
check-out <commit>	CABEÇA	SIM	SIM	SIM
Nível de arquivo				
redefinir [confirmar] <caminhos>	NÃO	SIM	NÃO	SIM
checkout [commit] <caminhos>	NÃO	SIM	SIM	**NÃO**

Fusão Avançada

A fusão no Git geralmente é bastante fácil. Como o Git torna fácil mesclar outro branch várias vezes, isso significa que você pode ter um branch de vida muito longa, mas pode mantê-lo atualizado à medida que avança, resolvendo pequenos conflitos com frequência, em vez de ser surpreendido por um enorme conflito no final da série.

No entanto, às vezes ocorrem conflitos complicados. Ao contrário de alguns outros sistemas de controle de versão, o Git não tenta ser muito inteligente na resolução de conflitos de mesclagem. A filosofia do Git é ser inteligente ao determinar quando uma resolução de mesclagem é inequívoca, mas se houver um conflito, ele não tenta ser inteligente para resolvê-lo automaticamente. Portanto, se você esperar muito para mesclar duas ramificações que divergem rapidamente, poderá ter alguns problemas.

Nesta seção, veremos quais podem ser alguns desses problemas e quais ferramentas o Git fornece para ajudar a lidar com essas situações mais complicadas. Também abordaremos alguns dos diferentes tipos de mesclagem não padrão que você pode fazer, bem como veremos como desistir das mesclagens que você fez.

Machine Translated by Google

Mesclar Conflitos

Embora tenhamos abordado algumas noções básicas sobre como resolver conflitos de mesclagem em Conflitos de mesclagem básicos, para conflitos mais complexos, o Git fornece algumas ferramentas para ajudá-lo a descobrir o que está acontecendo e como lidar melhor com o conflito.

Em primeiro lugar, se possível, tente garantir que seu diretório de trabalho esteja limpo antes de fazer uma mesclagem que possa ter conflitos. Se você tiver um trabalho em andamento, confirme-o em uma ramificação temporária ou armazene-o. Isso permite que você possa desfazer **qualquer coisa** que tentar aqui. Se você tiver alterações não salvas em seu diretório de trabalho ao tentar uma mesclagem, algumas dessas dicas podem ajudá-lo a preservar esse trabalho.

Vamos percorrer um exemplo muito simples. Temos um arquivo Ruby super simples que imprime 'hello world'.

```
#! /usr/bin/env rubi

def olá
coloca 'olá mundo' fim


olá()
```

Em nosso repositório, criamos uma nova ramificação chamada whitespace e passamos a alterar todas as terminações de linha do Unix para as terminações de linha do DOS, basicamente alterando cada linha do arquivo, mas apenas com espaço em branco. Então trocamos a linha “hello world” por “hello mundo”.

```
$ git checkout -b espaço em branco
Mudou para um novo branch 'whitespace'

$ unix2dos hello.rb
unix2dos: convertendo arquivo hello.rb para formato DOS ...
$ git commit -am 'Converter hello.rb para DOS' [whitespace
 3270f76] Converter hello.rb para DOS 1 arquivo alterado, 7
inserções (+) , 7 deleções(-)

$ vim hello.rb $
git diff -b diff --
git a/hello.rb b/hello.rb index
 ac51efd..e85207e 100755
--- a/olá.rb +++
b/olá.rb
@@ -1,7 +1,7
@@ #! /usr/bin/env rubi

def olá -
 coloca 'olá mundo' +
coloca 'olá mundo'^M end
```

Machine Translated by Google

```
olá()

$ git commit -am 'Usar espanhol em vez de
inglês' [whitespace 6d338d2] Usar espanhol em vez de
inglês 1 arquivo alterado, 1 inserção(+), 1 exclusão(-)
```

Agora voltamos para nossa ramificação master e adicionamos alguma documentação para a função.

```
$ git checkout mestre
Comutado para ramificação 'mestre'

$ vim hello.rb $
git diff diff --git
a/hello.rb b/hello.rb index
ac51efd..36c06c8 100755
--- a/olá.rb +++
b/olá.rb @@
-1,5 +1,6 @@ #! /
usr/bin/env rubi

+# imprime uma saudação
def hello puts 'hello world'
end


$ git commit -am 'Adicionar comentário documentando a
função' [master bec6336] Adicionar comentário documentando
a função 1 arquivo alterado, 1 inserção(+)
```

Agora tentamos mesclar em nossa ramificação de espaço em branco e obteremos conflitos por causa das alterações de espaço em branco.

```
$ git merge whitespace
Mesclagem automática
hello.rb CONFLICT (conteúdo): Conflito de mesclagem em
hello.rb Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

Abortando uma fusão

Agora temos algumas opções. Primeiro, vamos cobrir como sair dessa situação. Se você talvez não estivesse esperando conflitos e não quer lidar com a situação ainda, você pode simplesmente sair da mesclagem com git merge --abort.

```
$ git status -sb ##
mestre
UU olá.rb
```

Machine Translated by Google

```
$ git go --abort

$ git status -sb
## mestre
```

A opção git merge --abort tenta reverter para o seu estado antes de executar a mesclagem. Os únicos casos em que pode não ser capaz de fazer isso perfeitamente seriam se você tivesse alterações não consolidadas e não confirmadas em seu diretório de trabalho quando o executou, caso contrário, deve funcionar bem.

Se, por algum motivo, você quiser apenas recomeçar, também pode executar git reset --hard HEAD e seu repositório voltará ao último estado confirmado. Lembre-se de que qualquer trabalho não confirmado será perdido, portanto, certifique-se de não querer nenhuma de suas alterações.

Ignorando espaços em branco

Neste caso específico, os conflitos estão relacionados a espaços em branco. Sabemos disso porque o caso é simples, mas também é muito fácil dizer em casos reais ao olhar para o conflito porque cada linha é removida de um lado e adicionada novamente do outro. Por padrão, o Git vê todas essas linhas como sendo alteradas, portanto não pode mesclar os arquivos.

No entanto, a estratégia de mesclagem padrão pode receber argumentos, e alguns deles tratam de ignorar adequadamente as alterações de espaço em branco. Se você perceber que tem muitos problemas de espaço em branco em uma mesclagem, basta anulá-la e fazê-la novamente, desta vez com -Xignore- all-space ou -Xignore-space-change. A primeira opção ignora **completamente** os espaços em branco ao comparar linhas, a segunda trata sequências de um ou mais caracteres de espaço em branco como equivalentes.

```
$ git merge -Xignore-space-change whitespace
Mesclagem automática hello.rb Mesclagem feita
pela estratégia 'recursiva'. ola.rb | 2 +- 1 arquivo
alterado, 1 inserção(+), 1 exclusão(-)
```

Como neste caso, as alterações reais do arquivo não foram conflitantes, uma vez que ignoramos as alterações dos espaços em branco, tudo se funde perfeitamente.

Isso é um salva-vidas se você tiver alguém em sua equipe que gosta de reformatar ocasionalmente tudo, desde espaços até guias ou vice-versa.

Recombinação manual de arquivos

Embora o Git lide muito bem com o pré-processamento de espaço em branco, existem outros tipos de alterações que talvez o Git não possa manipular automaticamente, mas são correções de script. Por exemplo, vamos fingir que o Git não conseguiu lidar com a alteração do espaço em branco e precisamos fazer isso manualmente.

O que realmente precisamos fazer é executar o arquivo que estamos tentando mesclar por meio de um programa dos2unix antes de tentar a mesclagem real do arquivo. Então, como faríamos isso?

Primeiro, entramos no estado de conflito de mesclagem. Em seguida, queremos obter cópias da minha versão do arquivo, a versão deles (da ramificação em que estamos mesclando) e a versão comum (de onde ambos os lados

Machine Translated by Google

ramificada). Em seguida, queremos corrigir o lado deles ou o nosso e tentar novamente a mesclagem apenas para este único arquivo.

Obter as três versões de arquivo é realmente muito fácil. O Git armazena todas essas versões no índice em “estágios”, cada um com números associados a eles. O estágio 1 é o ancestral comum, o estágio 2 é a sua versão e o estágio 3 é do MERGE_HEAD, a versão na qual você está mesclando (“deles”).

Você pode extrair uma cópia de cada uma dessas versões do arquivo em conflito com o comando git show e uma sintaxe especial.

```
$ git show :1:hello.rb > hello.common.rb $
git show :2:hello.rb > hello.ours.rb $ git show
:3:hello.rb > hello.theirs.rb
```

Se você deseja obter um núcleo um pouco mais rígido, também pode usar o comando ls-files -u plumbing para obter os SHA-1s reais dos blobs do Git para cada um desses arquivos.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1 hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2 hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3 alô.rb
```

O :1:hello.rb é apenas um atalho para procurar aquele blob SHA-1.

Agora que temos o conteúdo de todos os três estágios em nosso diretório de trabalho, podemos corrigi-los manualmente para corrigir o problema de espaço em branco e mesclar novamente o arquivo com o pouco conhecido comando git merge-file, que faz exatamente isso .

```
$ dos2unix hello.theirs.rb
dos2unix: convertendo o arquivo hello.theirs.rb para o formato Unix ...

$ 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 rubi


+# imprime uma saudação
def hello - puts 'hello world'
+ puts 'hello mundo' end
```

Machine Translated by Google

```
olá()
```

Neste ponto, mesclamos bem o arquivo. Na verdade, isso funciona melhor do que ignorar a opção de mudança de espaço, porque na verdade corrige as mudanças de espaço em branco antes da mesclagem, em vez de simplesmente ignorá-las. Na mesclagem ignorar-espaço-alterar , na verdade acabamos com algumas linhas com terminações de linha do DOS, misturando as coisas.

Se você quiser ter uma ideia antes de finalizar este commit sobre o que realmente foi alterado entre um lado ou outro, você pode pedir ao git diff para comparar o que está em seu diretório de trabalho que você está prestes a confirmar como resultado da mesclagem para qualquer uma dessas etapas. Vamos passar por todos eles.

Para comparar seu resultado com o que você tinha em sua ramificação antes da mesclagem, em outras palavras, para ver o que a mesclagem introduziu, você pode executar git diff --ours:

```
$ git diff --ours *
Caminho não mesclado
hello.rb diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb +++ b/hello.rb @@
-2,7 +2,7 @@


# imprime uma saudação
def hello - puts 'hello
world' + puts 'hello mundo'
end


olá()
```

Aqui podemos ver facilmente que o que aconteceu em nossa ramificação, o que estamos realmente introduzindo neste arquivo com esta mesclagem, está alterando aquela única linha.

Se quisermos ver como o resultado da mesclagem difere do que estava do lado deles, você pode executar git diff --theirs. Neste e no exemplo a seguir, temos que usar -b para remover o espaço em branco porque estamos comparando-o com o que está no Git, não com nosso arquivo hello.theirs.rb limpo.

```
$ git diff --theirs -b *
Caminho não mesclado
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
rubi


+# imprime uma saudação
def olá
```

Machine Translated by Google

```
coloca 'olá mundo' final
```

Finalmente, você pode ver como o arquivo mudou de ambos os lados com git diff --base.

```
$ git diff --base -b *
Caminho não mesclado
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 rubi



+# imprime uma saudação
def hello - puts 'hello world'
+ puts 'hello mundo' end



olá()
```

Neste ponto, podemos usar o comando git clean para limpar os arquivos extras que criamos para fazer a mesclagem manual, mas que não são mais necessários.

```
$ git clean -f
Removendo hello.common.rb
Removendo hello.ours.rb
Removendo hello.theirs.rb
```

Verificando Conflitos

Talvez não estejamos satisfeitos com a resolução neste ponto por algum motivo, ou talvez editar manualmente um ou ambos os lados ainda não tenha funcionado bem e precisamos de mais contexto.

Vamos mudar um pouco o exemplo. Para este exemplo, temos duas ramificações de vida mais longa, cada uma com alguns commits, mas que criam um conflito de conteúdo legítimo quando mescladas.

```
$ git log --graph --oneline --decorate --all * f1270f7
(HEAD, master) Atualizar README * 9af9d3b Criar
README * 694971d Atualizar frase para 'hola
world' | * e3eb223 (mundo) Adicionar mais testes |
* 7cff591 Criar script de teste inicial | * c3ffff1 Muda
o texto para 'hello mundo' |/ * b7dcc89 Código
inicial de hello world
```

Machine Translated by Google

Agora temos três commits únicos que vivem apenas no ramo master e três outros que vivem no ramo mundo . Se tentarmos mesclar a ramificação mundo , teremos um conflito.

```
$ git merge mundo
Mesclagem automática
hello.rb CONFLICT (conteúdo): Conflito de mesclagem em
hello.rb Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

Gostaríamos de ver qual é o conflito de mesclagem. Se abrirmos o arquivo, veremos algo assim:

```
#! /usr/bin/env rubi

def olá
<<<<<<<
HEAD coloca 'hola world'
=======
puts 'olá mundo'
>>>>>>> fim do mundo



olá()
```

Ambos os lados da mesclagem adicionaram conteúdo a este arquivo, mas alguns dos commits modificaram o arquivo no mesmo local que causou este conflito.

Vamos explorar algumas ferramentas que agora você tem à sua disposição para determinar como esse conflito surgiu. Talvez não seja óbvio como exatamente você deve corrigir esse conflito. Você precisa de mais contexto.

Uma ferramenta útil é git checkout com a opção --conflict . Isso fará o check-out do arquivo novamente e substituirá os marcadores de conflito de mesclagem. Isso pode ser útil se você quiser redefinir os marcadores e tentar resolvê-los novamente.

Você pode passar --conflict diff3 ou merge (que é o padrão) . Se você passar diff3, o Git usará uma versão ligeiramente diferente dos marcadores de conflito, não apenas fornecendo as versões “nossa” e “deles”, mas também a versão “base” inline para fornecer mais contexto.

```
$ git checkout --conflict=diff3 hello.rb
```

Depois de executá-lo, o arquivo ficará assim:

```
#! /usr/bin/env rubi

def olá
<<<<<<< nosso
coloca 'hola mundo'
|||||||| base
```

Machine Translated by Google

```
coloca 'olá mundo'
=======
puts 'hello mundo'
>>>>>>> fim deles



olá()
```

Se você gosta desse formato, pode defini-lo como padrão para futuros conflitos de mesclagem, definindo a configuração merge.conflictstyle como diff3.

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

O comando git checkout também pode receber as opções --ours e --theirs , o que pode ser uma maneira muito rápida de escolher apenas um lado ou outro sem mesclar as coisas.

Isso pode ser particularmente útil para conflitos de arquivos binários em que você pode simplesmente escolher um lado ou onde deseja mesclar apenas certos arquivos de outro ramo - você pode fazer a mesclagem e, em seguida, verificar certos arquivos de um lado ou de outro antes de confirmar .

Log de mesclagem

Outra ferramenta útil ao resolver conflitos de merge é o git log. Isso pode ajudá-lo a obter contexto sobre o que pode ter contribuído para os conflitos. Revisar um pouco da história para lembrar por que duas linhas de desenvolvimento estavam tocando a mesma área de código pode ser muito útil às vezes.

Para obter uma lista completa de todos os commits únicos que foram incluídos em qualquer ramificação envolvida nessa mesclagem, podemos usar a sintaxe de “ponto triplo” que aprendemos em Ponto Triplo.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD <
f1270f7 Atualizar README < 9af9d3b Criar README <
694971d Atualizar frase para 'hola world' > e3eb223
Adicionar mais testes > 7cff591 Criar script de teste inicial
> c3ffff1 Alterar texto para ' ola mundo'
```

Essa é uma boa lista do total de seis commits envolvidos, bem como em qual linha de desenvolvimento cada commit estava.

Podemos simplificar ainda mais isso para nos dar um contexto muito mais específico. Se adicionarmos a opção --merge ao git log, ele mostrará apenas os commits em ambos os lados da mesclagem que tocam em um arquivo que está em conflito no momento.

```
$ git log --oneline --left-right --merge < 694971d
Frase de atualização para 'hola world'
```

Machine Translated by Google

```
> c3ffff1 Muda o texto para 'hello mundo'
```

Se você executar isso com a opção -p , obterá apenas as diferenças para o arquivo que acabou em conflito. Isso pode ser **muito** útil para fornecer rapidamente o contexto necessário para ajudar a entender por que algo entra em conflito e como resolvê-lo de maneira mais inteligente.

Formato Diferenciado Combinado

Como o Git prepara todos os resultados de mesclagem bem-sucedidos, quando você executa git diff enquanto está em um estado de mesclagem em conflito, obtém apenas o que ainda está em conflito. Isso pode ser útil para ver o que você ainda precisa resolver.

Quando você executa o git diff diretamente após um conflito de mesclagem, ele fornece informações em um formato de saída diff bastante exclusivo.

```
$ 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
rubi


def olá
++<<<<<<<
HEAD + puts 'hola world'
++=========
+ puts 'olá mundo' +
+>>>>>>> fim do mundo


olá()
```

O formato é chamado de “Combined Diff” e oferece duas colunas de dados ao lado de cada linha. A primeira coluna mostra se essa linha é diferente (adicionada ou removida) entre o ramo “nosso” e o arquivo em seu diretório de trabalho e a segunda coluna faz o mesmo entre o ramo “deles” e sua cópia do diretório de trabalho.

Nesse exemplo, você pode ver que as linhas <<<<<<< e >>>>>>> estão na cópia de trabalho, mas não estavam em nenhum dos lados da mesclagem. Isso faz sentido porque a ferramenta de mesclagem os colocou lá para o nosso contexto, mas devemos removê-los.

Se resolvermos o conflito e executarmos git diff novamente, veremos a mesma coisa, mas é um pouco mais útil.

```
$ vim hello.rb $
git diff diff --cc
hello.rb
```

Machine Translated by Google

```
index 0399cd5,59727f0..0000000
--- a/hello.rb +++ b/hello.rb

@@@ -1,7 -1,7 +1,7
@@@ #! /usr/bin/env rubi

def olá -
coloca 'olá mundo' -
coloca 'olá mundo' ++
coloca 'olá mundo' end


olá()
```

Isso nos mostra que “hola mundo” estava do nosso lado, mas não na cópia de trabalho, que “hello mundo” estava do lado deles, mas não na cópia de trabalho e, finalmente, que “hola mundo” não estava em nenhum dos lados, mas agora está em a cópia de trabalho. Isso pode ser útil para revisar antes de confirmar a resolução.

Você também pode obter isso no git log para qualquer mesclagem para ver como algo foi resolvido após o fato. O Git produzirá este formato se você executar git show em um commit de mesclagem ou se adicionar uma opção --cc a um git log -p (que por padrão mostra apenas patches para commits que não são de mesclagem).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Mesclar: f1270f7 e3eb223 Autor: Scott Chacon
<schacon@gmail.com> Data: sex 19 set 18:14:49
2014 +0200

Mesclar ramificação 'mundo'

Conflitos: 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 rubi



def olá -
coloca 'olá mundo' -
coloca 'olá mundo' ++
coloca 'olá mundo' end


olá()
```

Machine Translated by Google

Desfazendo Fusões

Agora que você sabe como criar um merge commit, provavelmente cometerá alguns erros. Uma das grandes vantagens de trabalhar com o Git é que não há problema em cometer erros, porque é possível (e em muitos casos fácil) corrigi-los.

Os commits de mesclagem não são diferentes. Digamos que você começou a trabalhar em um branch de tópico, acidentalmente o fundiu no master, e agora seu histórico de commits está assim:

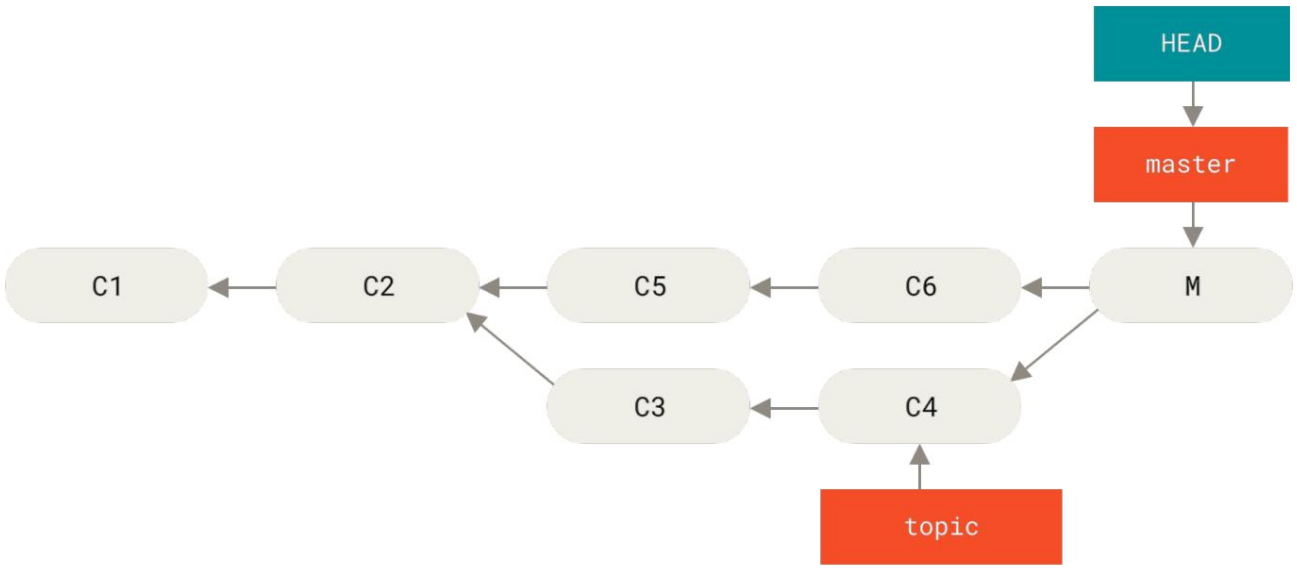


Figura 137. Confirmação de mesclagem acidental

Existem duas maneiras de abordar esse problema, dependendo de qual é o resultado desejado.

Corrija as referências

Se a confirmação de mesclagem indesejada existir apenas em seu repositório local, a solução mais fácil e melhor é mover as ramificações para que apontem para onde você deseja. Na maioria dos casos, se você seguir o errôneo git merge com git reset --hard HEAD~, isso redefinirá os ponteiros de ramificação para que fiquem assim:

Machine Translated by Google

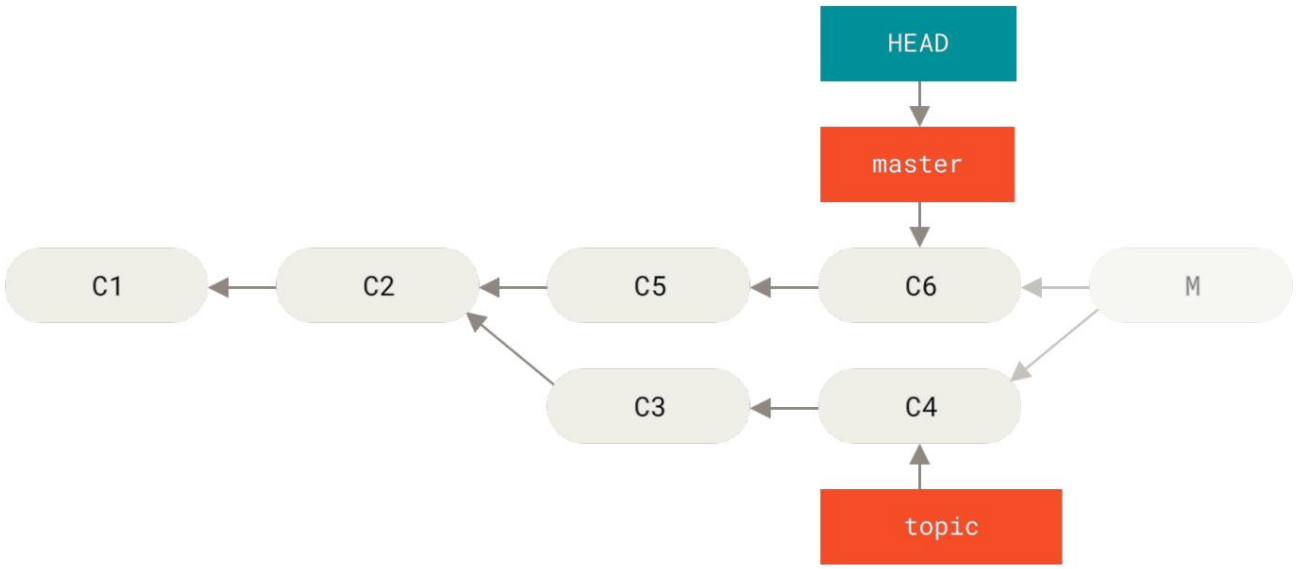


Figura 138. Histórico após git reset --hard HEAD~

Cobrimos a redefinição em Redefinir desmistificado, então não deve ser muito difícil descobrir o que está acontecendo aqui. Aqui está uma atualização rápida: reset --hard geralmente passa por três etapas:

1. Mova os pontos de ramificação HEAD para. Neste caso, queremos mover o master para onde estava antes do merge commit (C6).
2. Faça o índice parecer HEAD.
3. Faça com que o diretório de trabalho se pareça com o índice.

A desvantagem dessa abordagem é que ela está reescrevendo o histórico, o que pode ser problemático com um repositório compartilhado. Confira The Perils of Rebaseing para saber mais sobre o que pode acontecer; a versão curta é que, se outras pessoas tiverem os commits que você está reescrevendo, provavelmente você deve evitar redefinir. Essa abordagem também não funcionará se qualquer outro commit tiver sido criado desde a fusão; mover os árbitros efetivamente perderia essas mudanças.

Reverter o commit

Se mover os ponteiros de ramificação não funcionar para você, o Git oferece a opção de fazer um novo commit que desfaz todas as alterações de um já existente. O Git chama essa operação de “reversão” e, nesse cenário específico, você a chamaria assim:

```
$ git revert -m 1 HEAD
[master b1d8379] Reverter "Mesclar ramificação 'tópico'"
```

O sinalizador -m 1 indica qual pai é a “linha principal” e deve ser mantido. Quando você invoca um merge em HEAD (git merge topic), o novo commit tem dois pais: o primeiro é HEAD (C6), e o segundo é a ponta do branch sendo merge em (C4). Nesse caso, queremos desfazer todas as alterações introduzidas pela mesclagem no pai nº 2 (C4), mantendo todo o conteúdo do pai nº 1 (C6).

O histórico com o commit de reversão se parece com isso:

Machine Translated by Google

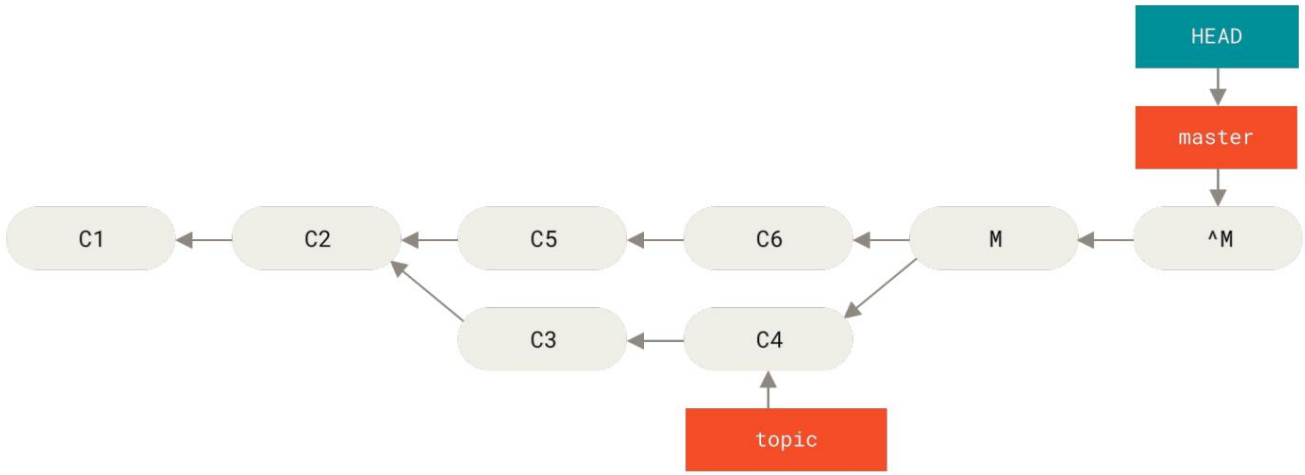


Figura 139. Histórico após git revert -m 1

O novo commit ^M tem exatamente o mesmo conteúdo do C6, então a partir daqui é como se a fusão nunca tivesse acontecido, exceto que os commits agora não mesclados ainda estão no histórico do HEAD . O Git ficará confuso se você tentar mesclar o tópico no mestre novamente:

```
$ git merge topic
Já atualizado.
```

Não há nada no tópico que ainda não esteja acessível a partir do mestre. O que é pior, se você adicionar trabalho ao tópico e mesclar novamente, o Git só trará as alterações *desde* a mesclagem revertida:

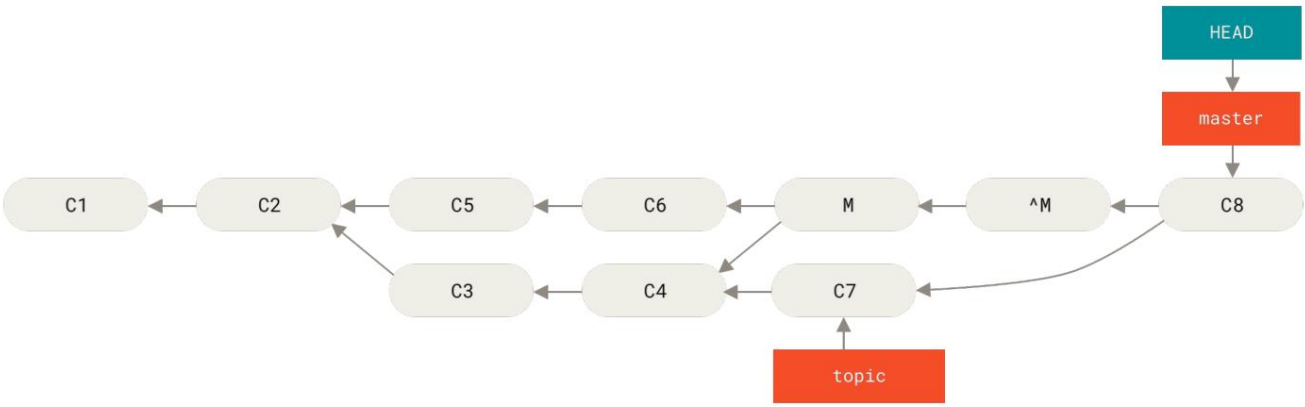


Figura 140. Histórico com uma mesclagem incorreta

A melhor maneira de contornar isso é cancelar a reversão da mesclagem original, já que agora você deseja trazer as alterações que foram revertidas e, **em seguida,** criar uma nova confirmação de mesclagem:

```
$ git revert ^M
[master 09f0126] Reverter "Reverter "Merge branch 'topic'""
$ git merge topic
```

Machine Translated by Google

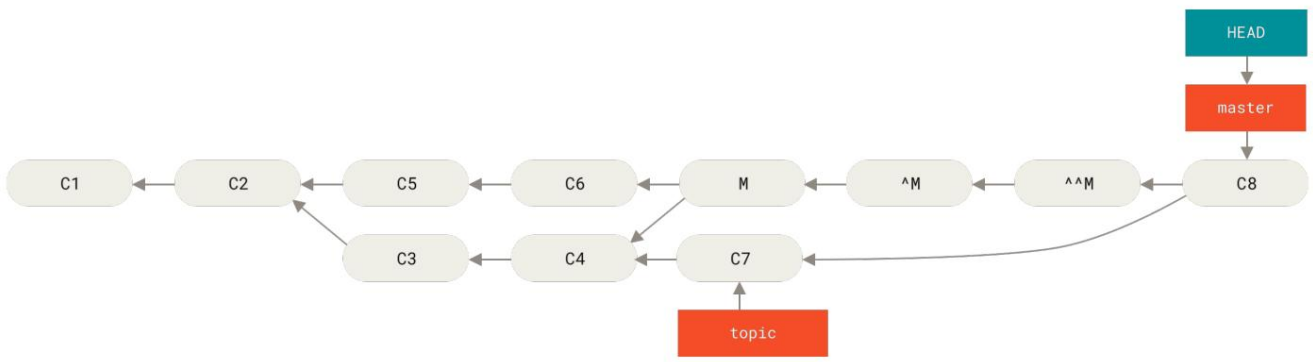


Figura 141. Histórico após mesclar novamente uma mesclagem revertida

Neste exemplo, M e ^M se cancelam. ^^M efetivamente se funde nas mudanças de C3 e C4, e C8 se funde com as mudanças de C7, então agora o tópico está totalmente mesclado.

Outros tipos de fusões

Até agora, cobrimos a mesclagem normal de duas ramificações, normalmente tratada com o que é chamado de estratégia "recursiva" de mesclagem. No entanto, existem outras maneiras de mesclar ramificações. Vamos cobrir alguns deles rapidamente.

Preferência nossa ou deles

Em primeiro lugar, há outra coisa útil que podemos fazer com o modo "recursivo" normal de mesclagem.
Já vimos as opções ignore-all-space e ignore-space-change que são passadas com um -X , mas também podemos dizer ao Git para favorecer um lado ou outro quando ele vê um conflito.

Por padrão, quando o Git vê um conflito entre duas ramificações sendo mescladas, ele adiciona marcadores de conflito de mesclagem em seu código e marca o arquivo como conflitante e permite que você o resolva. Se você preferir que o Git simplesmente escolha um lado específico e ignore o outro lado em vez de permitir que você resolva manualmente o conflito, você pode passar o comando de mesclagem para -Xours ou -Xtheirs.

Se o Git vir isso, ele não adicionará marcadores de conflito. Quaisquer diferenças que possam ser mescladas serão mescladas. Quaisquer diferenças que entrem em conflito, ele simplesmente escolherá o lado que você especificar no todo, incluindo arquivos binários.

Se voltarmos ao exemplo "hello world" que estávamos usando antes, podemos ver que mesclar em nossa ramificação causa conflitos.

```
$ git merge mundo
Mesclagem automática
hello.rb CONFLICT (conteúdo): Conflito de mesclagem em
hello.rb Resolvido 'hello.rb' usando a resolução anterior.
Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

No entanto, se o executarmos com -Xours ou -Xtheirs, isso não acontecerá.

```
$ git merge -Xours mundo
Mesclagem automática hello.rb
```

Machine Translated by Google

```
Fusão feita pela estratégia 'recursiva'. ola.rb
| 2 +- teste.sh | 2 ++ 2 arquivos alterados, 3
inserções (+), 1 exclusão (-) modo de
criação 100644 test.sh
```

Nesse caso, em vez de obter marcadores de conflito no arquivo com “hello mundo” de um lado e “hola world” do outro, ele simplesmente selecionará “hola world”. No entanto, todas as outras alterações não conflitantes nessa ramificação são mescladas com sucesso.

Essa opção também pode ser passada para o comando git merge-file que vimos anteriormente executando algo como git merge-file --ours para mesclagens de arquivos individuais.

Se você quer fazer algo assim, mas não tem o Git nem mesmo para tentar mesclar as alterações do outro lado, existe uma opção mais draconiana, que é a *estratégia de mesclagem “nossa”.* Isso é diferente da *opção de mesclagem recursiva “nossa”.*

Isso basicamente fará uma mesclagem falsa. Ele gravará um novo commit de mesclagem com ambos os branches como pais, mas nem mesmo olhará para o branch em que você está mesclando. Ele simplesmente registrará como resultado do merge o código exato em seu branch atual.

```
$ git merge -s ours mundo
Mesclagem feita pela estratégia 'ours'.
$ git diff HEAD HEAD~ $
```

Você pode ver que não há diferença entre o branch em que estávamos e o resultado da mesclagem.

Isso geralmente pode ser útil para basicamente enganar o Git fazendo-o pensar que uma ramificação já foi mesclada ao fazer uma mesclagem mais tarde. Por exemplo, digamos que você ramificou uma ramificação de lançamento e fez algum trabalho nela que deseja mesclar novamente em sua ramificação principal em algum momento. Enquanto isso, algumas correções de bug no master precisam ser portadas para o seu branch de lançamento . Você pode mesclar o branch bugfix com o branch release e também mesclar o mesmo branch com o branch master (mesmo que a correção já esteja lá), então quando você mesclar o branch release novamente, não haverá conflitos do bugfix.

Fusão de subárvore

A ideia da subárvore merge é que você tem dois projetos, e um dos projetos é mapeado para um subdiretório do outro. Quando você especifica uma mesclagem de subárvore, o Git geralmente é inteligente o suficiente para descobrir que uma é uma subárvore da outra e mesclar apropriadamente.

Veremos um exemplo de adição de um projeto separado em um projeto existente e, em seguida, mesclando o código do segundo em um subdiretório do primeiro.

Primeiro, adicionaremos o aplicativo Rack ao nosso projeto. Adicionaremos o projeto Rack como uma referência remota em nosso próprio projeto e, em seguida, verificaremos em sua própria ramificação:

```
$ 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: Comprimindo objetos: 100% (1465/1465), concluído.
remoto: Total 3184 (delta 1952), reutilizado 2770 (delta 1675)



Recebendo objetos: 100% (3184/3184), 677,42 KiB | 4 KiB/s, pronto.
Resolvendo deltas: 100% (1952/1952), concluído.
De https://github.com/rack/rack * [nova
ramificação] build -> rack_remote/ [nova ramificação] master -> rack_remote/
master * [nova ramificação] rack-0.4 -> rack_remote/rack-0.4 * [nova ramificação] rack-0.9 -> rack_remote/rack-0.9
rack_remote/master Branch rack_branch configurado para acompanhar remoto.
rack_remote/rack_branch.



Mudou para uma nova filial "rack_branch"
```

Agora temos a raiz do projeto Rack em nossa ramificação rack_branch e nosso próprio projeto na ramificação master . Se você verificar um e depois o outro, verá que eles têm projetos diferentes raízes:

```
$ ls
AUTORES                PROBLEMAS CONHECIDOS Rakefile    contribuir    lib
COPIANDO               lixeira              exemplo     teste
README $ git checkout mestre
Comutado para ramificação "mestre"
$ ls
Leia-me
```

Este é um conceito meio estranho. Nem todas as ramificações em seu repositório precisam ser ramificações do mesmo projeto. Não é comum, porque raramente é útil, mas é bastante fácil ter branches contendo históricos completamente diferentes.

Neste caso, queremos colocar o projeto Rack em nosso projeto mestre como um subdiretório. Podemos fazer isso no Git com git read-tree. Você aprenderá mais sobre read-tree e seus amigos no Git Internals, mas, por enquanto, saiba que ele lê a árvore raiz de uma ramificação em sua área de preparação atual e diretório de trabalho. Acabamos de voltar para sua ramificação master e colocamos a ramificação rack_branch no subdiretório rack de nossa ramificação master de nosso projeto principal:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Quando fazemos o commit, parece que temos todos os arquivos do Rack naquele subdiretório – como se os copiássemos de um tarball. O que é interessante é que podemos mesclar com bastante facilidade as alterações de um dos ramos para o outro. Portanto, se o projeto do Rack for atualizado, podemos obter as alterações upstream alternando para essa ramificação e obtendo:

Machine Translated by Google

```
$ git checkout rack_branch $ git
pull
```

Em seguida, podemos mesclar essas alterações de volta em nossa ramificação principal . Para obter as alterações e pré-preencher a mensagem de confirmação, use a opção --squash , bem como a opção -Xsubtree da estratégia de mesclagem recursiva . A estratégia recursiva é o padrão aqui, mas a incluímos para maior clareza.

```
$ git checkout master $
git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit - não atualizando o HEAD
A fusão automática correu bem; parou antes de confirmar conforme solicitado
```

Todas as alterações do projeto Rack são mescladas e prontas para serem confirmadas localmente. Você também pode fazer o oposto – fazer alterações no subdiretório rack de sua ramificação master e depois mesclá-las em sua ramificação rack_branch posteriormente para enviá-las aos mantenedores ou enviá-las para o upstream.

Isso nos dá uma maneira de ter um fluxo de trabalho um pouco semelhante ao fluxo de trabalho do submódulo sem usar submódulos (que abordaremos em Submódulos). Podemos manter ramificações com outros projetos relacionados em nosso repositório e subárvores mesclá-los em nosso projeto ocasionalmente. É bom em alguns aspectos, por exemplo, todo o código está comprometido em um único local. No entanto, ele tem outras desvantagens, pois é um pouco mais complexo e mais fácil cometer erros ao reintegrar as alterações ou enviar acidentalmente uma ramificação para um repositório não relacionado.

Outra coisa um pouco estranha é que para obter uma diferença entre o que você tem em seu subdiretório rack e o código em sua ramificação rack_branch – para ver se você precisa mesclá-los – você não pode usar o comando diff normal. Em vez disso, você deve executar git diff-tree com o branch que deseja comparar
para:

```
$ git diff-tree -p rack_branch
```

Ou, para comparar o que está em seu subdiretório rack com o que era o branch master no servidor na última vez que você buscou, você pode executar:

```
$ git diff-tree -p rack_remote/master
```

Voo

A funcionalidade git rerere é um recurso um pouco oculto. O nome significa “reutilizar resolução gravada” e, como o nome indica, permite que você peça ao Git para lembrar como você resolveu um conflito de hunk para que, na próxima vez que ele vir o mesmo conflito, o Git possa resolvê-lo para você automaticamente .

Há vários cenários em que essa funcionalidade pode ser realmente útil. Um dos exemplos mencionados na documentação é quando você deseja garantir que uma ramificação de tópico de longa duração seja mesclada de maneira limpa, mas não deseja ter um monte de mesclagens intermediárias

Machine Translated by Google

commits bagunçando seu histórico de commits. Com o rerere ativado, você pode tentar a mesclagem ocasional, resolver os conflitos e, em seguida, sair da mesclagem. Se você fizer isso continuamente, a mesclagem final deve ser fácil porque o rerere pode fazer tudo automaticamente para você.

Essa mesma tática pode ser usada se você quiser manter uma ramificação rebaseada para não ter que lidar com os mesmos conflitos de rebase toda vez que fizer isso. Ou, se você quiser pegar uma ramificação que você fundiu e corrigiu vários conflitos e, em vez disso, decidir rebaseá-la — provavelmente não terá que fazer todos os mesmos conflitos novamente.

Outra aplicação de rerere é onde você mescla um monte de ramificações de tópicos em evolução em uma cabeça testável ocasionalmente, como o próprio projeto Git costuma fazer. Se os testes falharem, você pode rebobinar as mesclagens e refazê-las sem a ramificação do tópico que fez os testes falharem sem ter que resolver novamente os conflitos.

Para ativar a funcionalidade rerere , basta executar esta configuração:

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

Você também pode ativá-lo criando o diretório .git/rr-cache em um repositório específico, mas a definição de configuração é mais clara e habilita esse recurso globalmente para você.

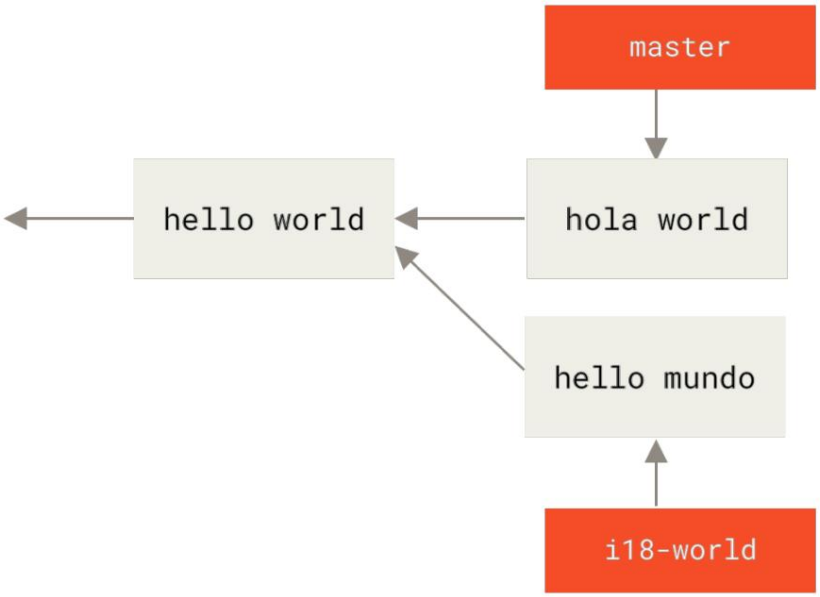
Agora vamos ver um exemplo simples, semelhante ao nosso anterior. Digamos que temos um arquivo chamado hello.rb que se parece com isto:

```
#! /usr/bin/env rubi

def olá
coloca 'olá mundo'
fim
```

Em um ramo mudamos a palavra “hello” para “hola”, depois em outro ramo mudamos o “world” para “mundo”, como antes.

Machine Translated by Google



Quando mesclarmos as duas ramificações, teremos um conflito de mesclagem:

```
$ git merge i18n-world
Mesclagem automática
hello.rb CONFLICT (conteúdo): Conflito de mesclagem
em hello.rb Pré-imagem gravada para 'hello.rb'
Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

Você deve observar a nova linha Recorded preimage for FILE ali. Caso contrário, deve parecer exatamente como um conflito de mesclagem normal. Neste ponto, rerere pode nos dizer algumas coisas. Normalmente, você pode executar git status neste ponto para ver o que está em conflito:

```
$ git status #
On branch master #
Unmerged paths: #
(use "git reset HEAD <arquivo>..." para remover o estágio)
# (use "git add <arquivo>..." para marcar a resolução) # #
ambos modificados:
                                    ola.rb
#
```

No entanto, git rerere também informará o que foi registrado no estado de pré-fusão com git rerere status:

```
$ git run status hello.rb
```

E git rerere diff mostrará o estado atual da resolução - o que você começou a resolver e o que você resolveu.

Machine Translated by Google

```
$ git rerere diff --- a/
hello.rb +++ b/hello.rb

@@ -1,11 +1,11 @@
#! /usr/bin/env rubi

def olá
-<<<<<<<
-coloca 'olá mundo'
-=======
+<<<<<<<
HEAD coloca 'hola world'
->>>>>>>
+=======
+ puts 'hello mundo'
+>>>>>>> i18n-world end
```

Além disso (e isso não está realmente relacionado ao rerere), você pode usar git ls-files -u para ver os arquivos em conflito e as versões anterior, esquerda e direita:

```
$ git ls-files -u 100644
39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 olá.rb
100644 54336ba847c3758ab604876419607e9443848474 3 alô.rb
```

Agora você pode resolvê-lo para ser apenas puts 'hola mundo' e você pode executar git rerere diff novamente para ver o que rerere irá lembrar:

```
$ git rerere diff --- a/
hello.rb +++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env rubi


def olá
-<<<<<<<
-coloca 'olá mundo'
-=======
- coloca 'hola mundo'
->>>>>>>
+ coloca 'olá mundo' fim
```

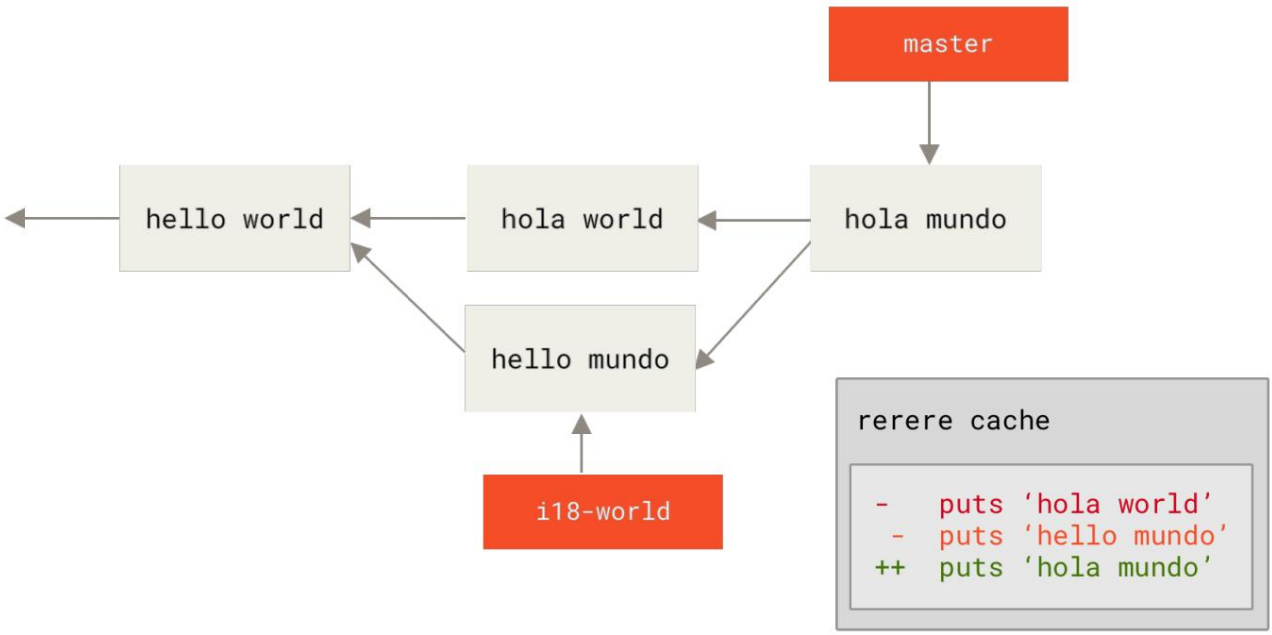
Isso basicamente diz que, quando o Git vê um conflito de fragmentos em um arquivo hello.rb que tem “hello mundo” de um lado e “hola world” do outro, ele o resolverá como “hola mundo”.

Machine Translated by Google

Agora podemos marcá-lo como resolvido e confirmá-lo:

```
$ git add hello.rb $ git
commit Resolução
gravada para 'hello.rb'. [mestre 68e16e5]
Mesclar ramificação 'i18n'
```

Você pode ver que "Resolução gravada para ARQUIVO".



Agora, vamos desfazer essa mesclagem e, em vez disso, rebaseá-la no topo de nossa ramificação master . Podemos mover nosso branch de volta usando git reset como vimos em Reset Demystified.

```
$ git reset --hard HEAD^
HEAD agora está em ad63f15 i18n olá
```

Nossa fusão foi desfeita. Agora vamos fazer o rebase da ramificação do tópico.

```
$ git checkout i18n-world
Mudou para o ramo 'i18n-world'

$ git rebase master
Primeiro, rebobinando a cabeça para reproduzir seu trabalho em cima dela...
Aplicando: i18n uma palavra
Usando informações de índice para reconstruir uma árvore base...
Voltando ao patch base e 3-way merge...
Mesclagem automática
hello.rb CONFLICT (conteúdo): Conflito de mesclagem em
hello.rb Resolvido 'hello.rb' usando a resolução anterior.
```

Machine Translated by Google

```
Falha ao mesclar as alterações.
Patch falhou em 0001 i18n uma palavra
```

Agora, temos o mesmo conflito de mesclagem que esperávamos, mas dê uma olhada no ARQUIVO resolvido usando a linha de resolução anterior. Se olharmos para o arquivo, veremos que ele já foi resolvido, não há marcadores de conflito de mesclagem nele.

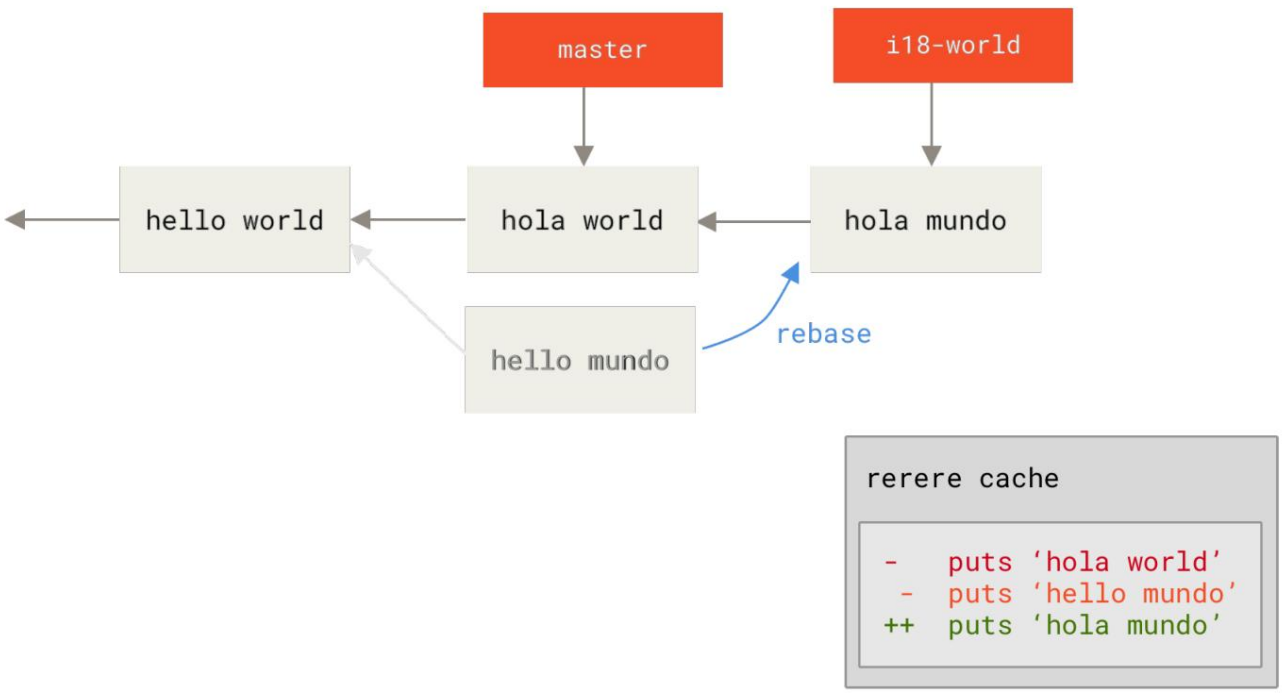
```
#! /usr/bin/env rubi

def olá
coloca 'olá mundo'
fim
```

Além disso, o git diff mostrará como foi resolvido automaticamente:

```
$ 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 rubi


def olá -
coloca 'olá mundo' -
coloca 'olá mundo' ++
coloca 'olá mundo' end
```



Machine Translated by Google

Você também pode recriar o estado do arquivo em conflito com git checkout:

```
$ git checkout --conflict=merge hello.rb $ cat
hello.rb #! /usr/bin/env rubi


def olá
<<<<<<< nosso
coloca 'olá mundo'
=======
puts 'hello mundo'
>>>>>>> fim deles
```

Vimos um exemplo disso em Advanced Merging. Por enquanto, vamos resolvê-lo apenas executando git rerere novamente:

```
$ git rerere
Resolvido 'hello.rb' usando a resolução anterior. $
gato olá.rb #! /usr/bin/env rubi


def olá
coloca 'olá mundo'
fim
```

Resolvemos o arquivo automaticamente usando a resolução em cache do rerere . Agora você pode adicionar e continuar o rebase para completá-lo.

```
$ git add hello.rb $
git rebase --continue
Aplicando: i18n uma palavra
```

Então, se você faz muitas re-merges, ou quer manter um topic branch atualizado com seu branch master sem uma tonelada de merges, ou você faz rebase frequentemente, você pode ativar o rerere para ajudar um pouco na sua vida.

Depurando com Git

Além de ser principalmente para controle de versão, o Git também fornece alguns comandos para ajudá-lo a depurar seus projetos de código-fonte. Como o Git foi projetado para lidar com praticamente qualquer tipo de conteúdo, essas ferramentas são bastante genéricas, mas geralmente podem ajudá-lo a procurar um bug ou culpado quando as coisas dão errado.

Anotação de arquivo

Se você rastrear um bug em seu código e quiser saber quando ele foi introduzido e por quê, a anotação de arquivo geralmente é sua melhor ferramenta. Ele mostra qual commit foi o último a modificar cada linha de qualquer

Machine Translated by Google

arquivo. Então, se você perceber que um método em seu código está com erros, você pode anotar o arquivo com gitculpe para determinar qual commit foi responsável pela introdução dessa linha.

O exemplo a seguir usa gitculpe para determinar qual commit e committer foi responsável pelas linhas no Makefile do kernel Linux de nível superior e, além disso, usa a opção -L para restringir a saída da anotação às linhas 69 a 82 desse arquivo:

```
$ git culpa -L 69,82 Makefile
b8b0618cf6fab (Cheng Renquan 2009-05-26 16:03:07 +0800 69) ifeq ("$(origem V)", "linha de comando")
b8b0618cf6fab (Cheng Renquan 2009-05- 26 16:03:07 +0800 70) KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif ^1da177e4c3f4 (Linus Torvalds
2005-04-16 15:20: 36 -0700 72) ifndef KBUILD_VERBOSE ^1da177e4c3f4 (Linus Torvalds 2005-04-16
15:20:36 -0700 73) KBUILD_VERBOSE = 0 ^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 ^7 -0700)
endif 7 -0700 1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75) 066b7ed955808 (Michal Marek
2014-07-04 14:29:30 +0200 76) ifeq ($(KBUILD_VERBOSE),1) 066bek7ed954808 (M0154808) -07-04
14:29:30 +0200 77) quieto = 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 81) 2 014-07-04 14:29:30 +0200 82) endif
```

Observe que o primeiro campo é o SHA-1 parcial do commit que modificou essa linha pela última vez. Os próximos dois campos são valores extraídos desse commit — o nome do autor e a data de criação desse commit — para que você possa ver facilmente quem modificou essa linha e quando. Depois disso vem o número da linha e o conteúdo do arquivo. Observe também as linhas de commit ^1da177e4c3f4 , onde o prefixo ^ designa as linhas que foram introduzidas no commit inicial do repositório e permaneceram inalteradas desde então. Isso é um pouco confuso, porque agora você viu pelo menos três maneiras diferentes que o Git usa o ^ para modificar um commit SHA-1, mas é isso que significa aqui.

Outra coisa legal sobre o Git é que ele não rastreia renomeações de arquivos explicitamente. Ele registra os instantâneos e tenta descobrir o que foi renomeado implicitamente, após o fato. Uma das características interessantes disso é que você pode pedir para descobrir todos os tipos de movimento de código também. Se você passar -C para culpar o git, o Git analisará o arquivo que você está anotando e tentará descobrir de onde vieram originalmente os trechos de código contidos nele, caso tenham sido copiados de outro lugar. Por exemplo, digamos que você esteja refatorando um arquivo chamado GITServerHandler.m em vários arquivos, um dos quais é GITPackUpload.m. Ao culpar GITPackUpload.m com a opção -C , você pode ver de onde as seções do código vieram originalmente:

```
$ git culpa -C -L 141.153 GITPackUpload.m f344f58d
GITServerHandler.m (Scott 2009-01-04 141) f344f58d
GITServerHandler.m (Scott 2009-01-04 142) - (void) 01-04 143) { 70befddd GITServerHandler.m (Scott
2009-03-22 144) ad11ac80 GITPackUpload.m (Scott 2009-03-24 145) ad11ac80 GITPackUpload.m (Scott
2009-03-24 146)                                                    //NSLog(@"REUNIR COMMI

                                                                   NSString *parentSha;
```

Machine Translated by Google

```
ad11ac80 GITPackUpload.m (Scott 24/03/2009 147) ad11ac80                GITCommit *commit = [g
GITPackUpload.m (Scott 24/03/2009 148)
ad11ac80 GITPackUpload.m ad11ac80 GITPackUpload.m (Scott 24/03/2009 149)  //NSLog(@"REUNIR COMMI
24/03/2009 150) 56ef2caf GITServerHandler.m (Scott 05/01/2009
151) 56ef2caf GITServerHandler.m (Scott 01-2009 2009) 152) 56ef2caf      if(commit)
GITServerHandler.m (Scott 05/01/2009 153)                                { [refDict setOb
```

Isso é realmente útil. Normalmente, você obtém como commit original o commit onde você copiou o código, porque é a primeira vez que você tocou nessas linhas neste arquivo. O Git informa o commit original onde você escreveu essas linhas, mesmo que tenha sido em outro arquivo.

Pesquisa Binária

Anotar um arquivo ajuda se você souber onde está o problema. Se você não sabe o que está quebrando e houve dezenas ou centenas de confirmações desde o último estado em que você sabe que o código funcionou, provavelmente recorrerá ao git bisect para obter ajuda. O comando bisect faz uma pesquisa binária em seu histórico de commits para ajudá-lo a identificar o mais rápido possível qual commit apresentou um problema.

Digamos que você acabou de enviar uma versão de seu código para um ambiente de produção, está recebendo relatórios de bug sobre algo que não estava acontecendo em seu ambiente de desenvolvimento e não consegue imaginar por que o código está fazendo isso. Você volta ao seu código e descobre que pode reproduzir o problema, mas não consegue descobrir o que está errado. Você pode *dividir* o código ao meio para descobrir. Primeiro, você executa git bisect start para fazer as coisas funcionarem e, em seguida, usa git bisect bad para informar ao sistema que o commit atual em que você está está quebrado. Em seguida, você deve informar ao bisect quando foi o último estado bom conhecido, usando git bisect good <good_commit>:

```
$ git bisect start $ git
bisect bad $ git bisect
good v1.0 Bisecting: 6
revisões restantes para testar depois disso
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] Tratamento de erros no repositório
```

O Git descobriu que cerca de 12 commits vieram entre o commit que você marcou como o último commit bom (v1.0) e a versão ruim atual, e verificou o do meio para você. Neste ponto, você pode executar seu teste para ver se o problema existe a partir deste commit. Se tiver, então foi introduzido algum tempo antes deste commit do meio; caso contrário, o problema foi introduzido algum tempo depois do commit do meio. Acontece que não há problema aqui, e você diz ao Git isso digitando git bisect good e continua sua jornada:

```
$ git bisect bom
Bissecção: 3 revisões restantes para testar depois disso
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Proteja isso
```

Agora você está em outro commit, a meio caminho entre aquele que você acabou de testar e seu commit ruim. Você executa seu teste novamente e descobre que este commit está quebrado, então você diz ao Git que com git bisect bad:

Machine Translated by Google

```
$ git bisect ruim
Bissecção: 1 revisões restantes para testar depois disso
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Eliminar tabela de exceções
```

Este commit está correto e agora o Git tem todas as informações necessárias para determinar onde o problema foi introduzido. Ele informa o SHA-1 do primeiro commit incorreto e mostra algumas das informações do commit e quais arquivos foram modificados nesse commit para que você possa descobrir o que aconteceu que pode ter introduzido esse bug:

```
$ git bisect bom
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 é o primeiro commit
ruim b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Autor: PJ Hyett <pjhyett@example.com> Data:
terça-feira, 27 de janeiro 14:48:32 2009 -0800

Proteger esta coisa

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M config
```

Quando terminar, você deve executar git bisect reset para redefinir seu HEAD para onde estava antes de começar, ou você terminará em um estado estranho:

```
$ git bisect redefinir
```

Esta é uma ferramenta poderosa que pode ajudá-lo a verificar centenas de commits para um bug introduzido em minutos. Na verdade, se você tiver um script que sairá 0 se o projeto for bom ou diferente de 0 se o projeto for ruim, você pode automatizar totalmente o git bisect. Primeiro, você novamente informa o escopo da bissetriz, fornecendo os commits bons e ruins conhecidos. Você pode fazer isso listando-os com o comando bisect start , se quiser, listando o commit ruim conhecido primeiro e o commit válido depois:

```
$ git bisect start HEAD v1.0 $ git
bisect run test-error.sh
```

Fazer isso automaticamente executa test-error.sh em cada commit com check-out até que o Git encontre o primeiro commit quebrado. Você também pode executar algo como fazer ou fazer testes ou o que quer que execute testes automatizados para você.

Submódulos

Muitas vezes acontece que, ao trabalhar em um projeto, você precisa usar outro projeto dentro dele.
Talvez seja uma biblioteca desenvolvida por terceiros ou que você esteja desenvolvendo separadamente e usando em vários projetos principais. Um problema comum surge nesses cenários: você deseja tratar os dois projetos como separados e, ainda assim, usar um dentro do outro.

Machine Translated by Google

Aqui está um exemplo. Suponha que você esteja desenvolvendo um site e criando feeds Atom. Em vez de escrever seu próprio código gerador de Atom, você decide usar uma biblioteca. É provável que você tenha que incluir este código de uma biblioteca compartilhada como uma instalação CPAN ou gem Ruby, ou copiar o código-fonte em sua própria árvore de projeto. O problema de incluir a biblioteca é que é difícil personalizar a biblioteca de qualquer maneira e, muitas vezes, mais difícil de implantá-la, porque você precisa garantir que cada cliente tenha essa biblioteca disponível. O problema de copiar o código em seu próprio projeto é que qualquer alteração personalizada que você fizer será difícil de mesclar quando as alterações upstream estiverem disponíveis.

O Git aborda esse problema usando submódulos. Os submódulos permitem que você mantenha um repositório Git como um subdiretório de outro repositório Git. Isso permite clonar outro repositório em seu projeto e manter seus commits separados.

Começando com Submódulos

Vamos percorrer o desenvolvimento de um projeto simples que foi dividido em um projeto principal e alguns subprojetos.

Vamos começar adicionando um repositório Git existente como um submódulo do repositório no qual estamos trabalhando. Para adicionar um novo submódulo, use o comando git submodule add com a URL absoluta ou relativa do projeto que você gostaria de começar a rastrear. Neste exemplo, adicionaremos uma biblioteca chamada “DbConnector”.

```
$ git submodule add https://github.com/chaconinc/DbConnector Cloning into
'DbConnector'... remote: Counting objects: 11, done. remote: Comprimindo
objetos: 100% (10/10), concluído. remoto: Total 11 (delta 0), reutilizado 11 (delta
0)

Desempacotando objetos: 100% (11/11), concluído.
Verificando a conectividade... concluído.
```

Por padrão, os submódulos adicionarão o subprojeto em um diretório com o mesmo nome do repositório, neste caso “DbConnector”. Você pode adicionar um caminho diferente no final do comando se quiser que ele vá para outro lugar.

Se você executar git status neste ponto, notará algumas coisas.

```
$ git status On
branch master Sua
branch está atualizada com 'origin/master'.

Mudanças a serem confirmadas:
(use "git reset HEAD <arquivo>..." para retirar o estágio)

novo arquivo: .gitmodules novo
arquivo: DbConnector
```

Primeiro você deve observar o novo arquivo .gitmodules . Este é um arquivo de configuração que armazena o mapeamento entre a URL do projeto e o subdiretório local para o qual você o puxou:

Machine Translated by Google

```
[submodule "DbConnector"] path
= DbConnector url = https://
github.com/chaconinc/DbConnector
```

Se você tiver vários submódulos, terá várias entradas nesse arquivo. É importante observar que este arquivo é controlado por versão com seus outros arquivos, como seu arquivo .gitignore . É empurrado e puxado com o resto do seu projeto. É assim que outras pessoas que clonam este projeto sabem de onde obter os projetos do submódulo.

ÿ

Como o URL no arquivo .gitmodules é o que outras pessoas tentarão primeiro clonar/obter, certifique-se de usar um URL que eles possam acessar, se possível. Por exemplo, se você usar um URL diferente para fazer push do que outros fariam para pull, use aquele ao qual outras pessoas têm acesso. Você pode sobrescrever este valor localmente com git config submodule.DbConnector.url PRIVATE_URL para seu próprio uso. Quando aplicável, um URL relativo pode ser útil.

A outra listagem na saída git status é a entrada da pasta do projeto. Se você executar git diff nisso, verá algo interessante:

```
$ git diff --cached DbConnector diff --git a/
DbConnector b/DbConnector novo modo de arquivo
160000 index 0000000..c3f01dc

--- /dev/null +++
b/DbConnector @@
-0,0 +1 @@
+Confirmação do subprojeto c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Embora DbConnector seja um subdiretório em seu diretório de trabalho, o Git o vê como um submódulo e não rastreia seu conteúdo quando você não está nesse diretório. Em vez disso, o Git o vê como um commit específico daquele repositório.

Se você quiser uma saída diff um pouco melhor, você pode passar a opção --submodule para git diff.

```
$ git diff --cached --submodule diff --git
a/.gitmodules b/.gitmodules novo modo de arquivo
100644 índice 0000000..71fc376

--- /dev/null +++
b/.gitmodules @@ -0,0
+1,3 @@ +[submodule
"DbConnector"] path = DbConnector
+           url = https://github.com/
+           chaconinc/DbConnector Submodule DbConnector
0000000...c3f01dc (novo submódulo)
```

Machine Translated by Google

Quando você confirma, você vê algo assim:

```
$ git commit -am 'Adicionar módulo DbConnector' [master
 fb9093c] Adicionar módulo DbConnector 2 arquivos
alterados, 4 inserções(+) criar modo 100644 .gitmodules
criar modo 160000 DbConnector
```

Observe o modo 160000 para a entrada DbConnector . Esse é um modo especial no Git que basicamente significa que você está gravando um commit como uma entrada de diretório em vez de um subdiretório ou arquivo.

Por fim, envie essas alterações:

```
$ git push mestre de origem
```

Clonando um projeto com submódulos

Aqui vamos clonar um projeto com um submódulo nele. Quando você clona tal projeto, por padrão você obtém os diretórios que contêm submódulos, mas nenhum dos arquivos dentro deles ainda:

```
$ git clone https://github.com/chaconinc/MainProject Cloning into
'MainProject'... remote: Counting objects: 14, done. remote: Comprimindo
objetos: 100% (13/13), concluído. remoto: Total 14 (delta 1), reutilizado
13 (delta 0)

Desempacotando objetos: 100% (14/14), concluído.
Verificando a conectividade... concluído.
$ cd MainProject $ ls -la total 16 drwxr-
xr-x 9 schacon staff 306 Set 17 15:21 .
drwxr-xr-x 7 schacon staff 238 Set 17
15:21 .. drwxr-xr-x 13 schacon staff 442 Set 17 15:21 .git -rw-r--r--
1 schacon staff 92 Set 17 15:21 .gitmodules drwxr-xr-x 2 schacon
staff 68 Set 17 15:21 DbConnector -rw-r--r-- 1 schacon staff 756 Set 17
15:21 Makefile drwxr-xr-x 3 schacon staff 102 Set 17 15:21 inclui drwxr-xr-x 4
schacon staff 136 Set 17 15:21 scripts drwxr-xr-x 4 schacon staff 136 Set 17
15:21 src $ cd DbConnector/ $ ls $
```

O diretório DbConnector está lá, mas vazio. Você deve executar dois comandos: git submodule init para inicializar seu arquivo de configuração local e git submodule update para buscar todos os dados desse projeto e verificar o commit apropriado listado em seu superprojeto:

Machine Translated by Google

```
$ git submodule init
Submódulo 'DbConnector' (https://github.com/chaconinc/DbConnector) registrado para o caminho 'DbConnector' $ git
 submodule update Clonagem em 'DbConnector'... remote: Counting objects: 11, done. remote: Comprimindo objetos:
100% (10/10), concluído. remoto: Total 11 (delta 0), reutilizado 11 (delta 0)




Desempacotando objetos: 100% (11/11), concluído.
Verificando a conectividade... concluído.
Caminho do submódulo 'DbConnector': verificado 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Agora seu subdiretório DbConnector está no estado exato em que estava quando você fez o commit anteriormente.

No entanto, existe outra maneira de fazer isso, que é um pouco mais simples. Se você passar --recurse-submodules para o comando git clone , ele inicializará e atualizará automaticamente cada submódulo no repositório, incluindo submódulos aninhados se algum dos submódulos no repositório tiver submódulos.

```
$ git clone --recurse-submodules https://github.com/chaconinc/MainProject Clonagem em
'MainProject'... remote: Counting objects: 14, done. remote: Comprimindo objetos: 100% (13/13),
concluído. remoto: Total 14 (delta 1), reutilizado 13 (delta 0)



Desempacotando objetos: 100% (14/14), concluído.
Verificando a conectividade... concluído.
Submódulo 'DbConnector' (https://github.com/chaconinc/DbConnector) registrado para o caminho 'DbConnector'

Clonando em 'DbConnector'... remoto:
Contando objetos: 11, concluído. remote:
Comprimindo objetos: 100% (10/10), concluído. remoto: Total 11
(delta 0), reutilizado 11 (delta 0)
Desempacotando objetos: 100% (11/11), concluído.
Verificando a conectividade... concluído.
Caminho do submódulo 'DbConnector': verificado 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Se você já clonou o projeto e esqueceu --recurse-submodules, pode combinar as etapas git submodule init e git submodule update executando git submodule update --init. Para também inicializar, buscar e verificar quaisquer submódulos aninhados, você pode usar o submódulo git infalível update --init --recursive.

Trabalhando em um projeto com submódulos

Agora temos uma cópia de um projeto com submódulos e iremos colaborar com nossos colegas de equipe tanto no projeto principal quanto no projeto do submódulo.

Machine Translated by Google

Puxando alterações upstream do submódulo remoto

O modelo mais simples de usar submódulos em um projeto seria se você estivesse simplesmente consumindo um subprojeto e desejasse obter atualizações dele de tempos em tempos, mas não estivesse realmente modificando nada em seu checkout. Vamos ver um exemplo simples lá.

Se você deseja verificar se há novos trabalhos em um submódulo, pode entrar no diretório e executar git fetch e git merge no branch upstream para atualizar o código local.

```
$ git fetch De
https://github.com/chaconinc/DbConnector c3f01dc..d0354fc
master -> origin/master $ git merge origin/master Atualizando c3f01dc..d0354fc
Scripts de avanço rápido/connect.sh | src/db.c | 1 + 2 arquivos
alterados, 2 inserções (+)

                                    1 +
```

Agora, se você voltar ao projeto principal e executar git diff --submodule, poderá ver que o submódulo foi atualizado e obter uma lista de commits que foram adicionados a ele. Se você não quiser digitar --submodule toda vez que executar git diff, você pode defini-lo como o formato padrão definindo o valor de configuração diff.submodule como “log”.

```
$ git config --global diff.submodule log $ git diff
Submodule DbConnector c3f01dc..d0354fc: > rotina
db mais eficiente > rotina de conexão melhor
```

Se você confirmar neste ponto, bloqueará o submódulo para ter o novo código quando outras pessoas atualizarem.

Também existe uma maneira mais fácil de fazer isso, se você preferir não buscar e mesclar manualmente no subdiretório. Se você executar git submodule update --remote, o Git entrará em seus submódulos, buscará e atualizará para você.

```
$ git submodule update --remote DbConnector remote:
Counting objects: 4, done. remote: Comprimindo objetos:
100% (2/2), concluído. remoto: Total 4 (delta 2), reutilizado 4
(delta 2)
Desempacotando objetos: 100% (4/4), concluído.
De https://github.com/chaconinc/DbConnector 3f19983..d0354fc
master -> origin/master Caminho do submódulo 'DbConnector': verificado
'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Este comando assumirá, por padrão, que você deseja atualizar o checkout para a ramificação padrão de

Machine Translated by Google

o repositório do submódulo remoto (aquele apontado por HEAD no controle remoto). Você pode, no entanto, definir isso para algo diferente, se desejar. Por exemplo, se você quiser que o submódulo DbConnector rastreie a ramificação "estável" desse repositório, você pode configurá-lo em seu arquivo .gitmodules (para que todos os outros também o rastreiem) ou apenas em seu arquivo .git/config local. Vamos configurá-lo no arquivo .gitmodules :

```
$ git config -f .gitmodules submodule.DbConnector.branch estável

$ git submodule update --remote remote:
Contando objetos: 4, concluído. remote:
Comprimindo objetos: 100% (2/2), concluído. remoto: Total 4
(delta 2), reutilizado 4 (delta 2)
Desempacotando objetos: 100% (4/4), concluído.
De https://github.com/chaconinc/DbConnector 27cf5d3..c87d55d
stable -> origin/stable Caminho do submódulo 'DbConnector':
verificado 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Se você deixar de fora o -f .gitmodules, ele fará a alteração apenas para você, mas provavelmente faz mais sentido rastrear essas informações com o repositório para que todos os outros também o façam.

Quando executarmos o git status neste ponto, o Git nos mostrará que temos "novos commits" no submódulo.

```
$ git status On
branch master Sua
branch está atualizada com 'origin/master'.

Alterações não testadas para confirmação:
(use "git add <arquivo>..." para atualizar o que será submetido) (use "git checkout
-- <arquivo>..." para descartar alterações no diretório de trabalho)

modificado: .gitmodules modificado:
DbConnector (novos commits)

nenhuma alteração adicionada ao commit (use "git add" e/ou "git commit -a")
```

Se você definir a configuração status.submodulesummary, o Git também mostrará um breve resumo das alterações em seus submódulos:

```
$ git config status.submodulesummary 1

$ git status On
branch master Sua
branch está atualizada com 'origin/master'.

Alterações não testadas para confirmação:
(use "git add <arquivo>..." para atualizar o que será submetido) (use "git checkout
-- <arquivo>..." para descartar alterações no diretório de trabalho)
```

Machine Translated by Google

```
modificado: .gitmodules modificado:
DbConnector (novos commits)

Submódulos alterados, mas não atualizados:

* DbConnector c3f01dc...c87d55d (4): > captura linhas
    terminadas não nulas
```

Neste ponto, se você executar git diff, podemos ver que modificamos nosso arquivo .gitmodules e também que há vários commits que baixamos e que estão prontos para serem confirmados em nosso projeto de submódulo.

```
$ 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 = estável




+
Submódulo DbConnector c3f01dc..c87d55d:
    > pegar linhas terminadas não nulas > tratamento de
    erros mais robusto > rotina db mais eficiente > melhor
    rotina de conexão
```

Isso é muito legal, pois podemos ver o log de commits que estamos prestes a fazer em nosso submódulo. Depois de confirmado, você também pode ver essas informações após o fato ao executar git log -p.

```
$ git log -p --submodule commit
0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Autor: Scott Chacon <schacon@gmail.com> Data: quarta-feira, 17
de setembro 16:37:02 2014 +0200

atualizando DbConnector para correções de bugs

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 = estábulo
+
```

Machine Translated by Google

```
Submódulo DbConnector c3f01dc..c87d55d:
    > pegar linhas terminadas não nulas
    > tratamento de erros mais robusto > rotina db
    mais eficiente > melhor rotina de conexão
```

Por padrão, o Git tentará atualizar **todos** os seus submódulos quando você executar git submodule update --remote. Se você tiver muitos deles, talvez queira passar o nome apenas do submódulo que deseja tentar atualizar.

Puxando alterações upstream do projeto remoto

Vamos agora entrar no lugar do seu colaborador, que tem seu próprio clone local do repositório MainProject. Simplesmente executar git pull para obter suas alterações recém-confirmadas não é suficiente:

```
$ git pull De
https://github.com/chaconinc/MainProject fb9093c..0a24cfc master -> origin/
master
Obtendo o submódulo DbConnector De https://
github.com/chaconinc/DbConnector c3f01dc..c87d55d stable -> origin/stable
Atualizando fb9093c..0a24cfc Fast-forward .gitmodules | 2 + - DbConector | 2 + - 2 arquivos alterados,
2 inserções (+), 2 exclusões (-)

$ git status On
branch master Sua branch
está atualizada com 'origin/master'.
Alterações não testadas para confirmação: (use
"git add <arquivo>..." para atualizar o que será submetido) (use "git checkout -- <arquivo>..." para
descartar alterações no diretório de trabalho)

modificado: DbConnector (novos commits)

Submódulos alterados, mas não atualizados:

* DbConnector c87d55d...c3f01dc (4): < captura linhas
    terminadas não nulas < tratamento de erros mais robusto
    < rotina de banco de dados mais eficiente

    < melhor rotina de conexão

nenhuma alteração adicionada ao commit (use "git add" e/ou "git commit -a")
```

Por padrão, o comando git pull busca recursivamente as alterações dos submódulos, como podemos ver na saída do primeiro comando acima. No entanto, ele não **atualiza** os submódulos. Isso é mostrado pela saída do comando git status , que mostra que o submódulo está “modificado” e tem “novo

Machine Translated by Google

comete". Além disso, os colchetes que mostram os novos commits apontam para a esquerda (<), indicando que esses commits são registrados no MainProject, mas não estão presentes no check-out local do DbConnector. Para finalizar a atualização, você precisa executar git submodule update:

```
$ atualização do submódulo git --init --recursive
Caminho do submódulo 'vendor/plugins/demo': verificado
'48679c6302815f6c76f1fe30625d795d9e55fc56'

$ git status
On branch master
Sua branch está atualizada com 'origin/master'. nada
a comprometer, árvore de trabalho limpa
```

Observe que, para garantir, você deve executar git submodule update com o sinalizador --init caso o MainProject confirme que você acabou de adicionar novos submódulos e com o sinalizador --recursive se algum submódulo tiver submódulos aninhados.

Se você deseja automatizar esse processo, pode adicionar o sinalizador --recurse-submodules ao comando git pull (desde o Git 2.14). Isso fará com que o Git execute a atualização do submódulo git logo após o pull, colocando os submódulos no estado correto. Além disso, se você quiser fazer o Git sempre puxar com --recurse -submodules, você pode definir a opção de configuração submodule.recurse como true (isso funciona para git pull desde o Git 2.15). Esta opção fará com que o Git use o sinalizador --recurse-submodules para todos os comandos que o suportam (exceto clone).

Há uma situação especial que pode acontecer ao puxar as atualizações do superprojeto: pode ser que o repositório upstream tenha alterado a URL do submódulo no arquivo .gitmodules em um dos commits que você puxa. Isso pode acontecer, por exemplo, se o projeto do submódulo mudar sua plataforma de hospedagem. Nesse caso, é possível que git pull --recurse-submodules ou git submodule update falhe se o superprojeto fizer referência a um commit de submódulo que não foi encontrado no submódulo remoto configurado localmente em seu repositório. Para remediar esta situação, o comando git submodule sync é necessário:

```
# copie a nova URL para sua configuração
local $ git submodule sync --recursive #
atualize o submódulo da nova URL $ git
submodule update --init --recursive
```

Trabalhando em um submódulo

É bem provável que, se você estiver usando submódulos, esteja fazendo isso porque realmente deseja trabalhar no código no submódulo ao mesmo tempo em que está trabalhando no código no projeto principal (ou em vários submódulos) . Caso contrário, você provavelmente estaria usando um sistema de gerenciamento de dependência mais simples (como Maven ou Rubygems).

Agora vamos ver um exemplo de como fazer alterações no submódulo ao mesmo tempo que no projeto principal e confirmar e publicar essas alterações ao mesmo tempo.

Até agora, quando executamos o comando git submodule update para buscar as alterações do submódulo

Machine Translated by Google

repositórios, o Git obteria as alterações e atualizaria os arquivos no subdiretório, mas deixaria o sub-repositório no que é chamado de estado “HEAD separado”. Isso significa que não há ramificação de trabalho local (como mestre, por exemplo) acompanhando as alterações. Sem alterações de rastreamento de ramificação de trabalho, isso significa que mesmo se você confirmar alterações no submódulo, essas alterações possivelmente serão perdidas na próxima vez que você executar git submodule update. Você precisa executar algumas etapas extras se quiser que as alterações em um submódulo sejam rastreadas.

Para configurar seu submódulo para ser mais fácil de entrar e hackear, você precisa fazer duas coisas. Você precisa entrar em cada submódulo e verificar uma ramificação para trabalhar. Então você precisa dizer ao Git o que fazer se você fez alterações e, em seguida, git submodule update --remote puxa o novo trabalho do upstream. As opções são que você pode mesclá-los em seu trabalho local ou tentar rebasear seu trabalho local sobre as novas alterações.

Em primeiro lugar, vamos ao diretório do submódulo e verificamos uma ramificação.

```
$ cd DbConnector/ $
git checkout estável
Alterado para ramificação 'estável'
```

Vamos tentar atualizar nosso submódulo com a opção “merge”. Para especificá-lo manualmente, podemos apenas adicionar a opção --merge à nossa chamada de atualização . Aqui veremos que houve uma alteração no servidor para este submódulo e ele foi mesclado.

```
$ cd .. $
git submodule update --remote --merge remote:
Contando objetos: 4, concluído. remote: Comprimindo
objetos: 100% (2/2), concluído. remoto: Total 4 (delta 2),
reutilizado 4 (delta 2)
Desempacotando objetos: 100% (4/4), concluído.
De https://github.com/chaconinc/DbConnector
c87d55d..92c7337 estável -> origem/estável Atualizando c87d55d..92c7337
Avanço rápido src/main.c | 1 + 1 arquivo alterado, 1 inserção(+)



Caminho do submódulo 'DbConnector': mesclado em '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Se formos para o diretório DbConnector, teremos as novas alterações já mescladas em nossa ramificação estável local . Agora vamos ver o que acontece quando fazemos nossa própria alteração local na biblioteca e outra pessoa envia outra alteração upstream ao mesmo tempo.

```
$ cd DbConnector/ $
vim src/db.c $ git
commit -am 'Unicode support' [stable f906e16]
Unicode support 1 arquivo alterado, 1
inserção(+)
```

Machine Translated by Google

Agora, se atualizarmos nosso submódulo, podemos ver o que acontece quando fizemos uma alteração local e o upstream também possui uma alteração que precisamos incorporar.

```
$ cd .. $
git submodule update --remote --rebase Primeiro,
rebobinando a cabeça para reproduzir seu trabalho em cima dela...
Aplicando: Caminho do submódulo
de suporte a Unicode 'DbConnector': rebaseado em '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Se você esquecer --rebase ou --merge, o Git apenas atualizará o submódulo para o que estiver no servidor e redefinirá seu projeto para um estado HEAD desanexado.

```
$ atualização do submódulo git --remote
Caminho do submódulo 'DbConnector': verificado '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Se isso acontecer, não se preocupe, você pode simplesmente voltar para o diretório e verificar sua ramificação novamente (que ainda conterá seu trabalho) e mesclar ou rebasear origin/stable (ou qualquer ramificação remota que você desejar) manualmente.

Se você não confirmou suas alterações em seu submódulo e executou uma atualização de submódulo que causaria problemas, o Git buscará as alterações, mas não sobrescreverá o trabalho não salvo em seu diretório de submódulo.

```
$ git submodule update --remote remote:
Contando objetos: 4, concluído. remoto:
Comprimindo objetos: 100% (3/3), concluído. remoto: Total 4
(delta 0), reutilizado 4 (delta 0)
Desempacotando objetos: 100% (4/4), concluído.
De https://github.com/chaconinc/DbConnector -> origin/stable
erro estável: Suas alterações locais nos seguintes arquivos serão
substituídas pelo checkout: scripts/setup.sh Por favor, confirme suas alterações ou armazene-as antes de poder
trocar de branch.

Abortando
Incapaz de verificar 'c75e92a2b3855c9e5b66f915308390d9db204aca' no caminho do submódulo
 'DbConnector'
```

Se você fez alterações que entram em conflito com algo alterado no upstream, o Git o avisará quando executar a atualização.

```
$ git submodule update --remote --merge Mesclagem
automática de scripts/setup.sh CONFLICT (conteúdo):
Conflito de mesclagem em scripts/setup.sh Pré-imagem gravada para
'scripts/setup.sh'
Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
Não é possível mesclar 'c75e92a2b3855c9e5b66f915308390d9db204aca' no caminho do submódulo
```

Machine Translated by Google

```
'DbConnector'
```

Você pode entrar no diretório do submódulo e corrigir o conflito como faria normalmente.

Publicando alterações de submódulo

Agora temos algumas mudanças em nosso diretório de submódulos. Algumas delas foram trazidas do upstream por nossas atualizações e outras foram feitas localmente e ainda não estão disponíveis para mais ninguém, pois ainda não as enviamos.

```
$ git diff
Submódulo DbConnector c87d55d..82d2ad3:
    > Mesclar da origem/estável
    > Atualizar script de configuração
    > Suporte Unicode
    > Remover método desnecessário
    > Adicionar nova opção para conn pooling
```

Se fizermos o commit no projeto principal e fizermos push dele sem fazer push das alterações do submódulo também, outras pessoas que tentarem verificar nossas alterações terão problemas, pois não terão como obter as alterações do submódulo que dependem . Essas alterações só existirão em nossa cópia local.

Para garantir que isso não aconteça, você pode pedir ao Git para verificar se todos os seus submódulos foram enviados corretamente antes de enviar o projeto principal. O comando git push usa o argumento --recurse -submodules que pode ser definido como “check” ou “on-demand”. A opção “check” fará com que o push simplesmente falhe se alguma das alterações do submódulo confirmadas não tiver sido enviada.

```
$ git push --recurse-submodules=check
Os seguintes caminhos de submódulo contêm alterações que não podem ser
encontradas em nenhum controle remoto:
DbConnector

Tente por favor

git push --recurse-submodules=sob demanda

ou cd para o caminho e use

git push

para empurrá-los para um controle remoto.
```

Como você pode ver, ele também nos dá alguns conselhos úteis sobre o que podemos fazer a seguir. A opção simples é entrar em cada submódulo e enviar manualmente para os controles remotos para garantir que eles estejam disponíveis externamente e, em seguida, tentar novamente. Se você deseja que o comportamento de verificação aconteça para todos os pushes, você pode tornar esse comportamento o padrão fazendo git config push.recurseSubmodules check

Machine Translated by Google

A outra opção é usar o valor “sob demanda”, que tentará fazer isso para você.

```
$ git push --recurse-submodules=on-demand Enviando
o submódulo 'DbConnector'
Contando objetos: 9, feito.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (8/8), concluído.
Objetos de escrita: 100% (9/9), 917 bytes | 0 byte/s, pronto.
Total 9 (delta 3), reutilizado 0 (delta 0)
Para https://github.com/chaconinc/DbConnector
c75e92a..82d2ad3 estável -> estável
Contando objetos: 2, feito.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (2/2), concluído.
Objetos de escrita: 100% (2/2), 266 bytes | 0 byte/s, pronto.
Total 2 (delta 1), reutilizado 0 (delta 0)
Para https://github.com/chaconinc/MainProject
3d6d338..9a377d1 mestre -> mestre
```

Como você pode ver, o Git entrou no módulo DbConnector e o empurrou antes de enviar o projeto principal. Se esse push do submódulo falhar por algum motivo, o push do projeto principal também falhará. Você pode tornar esse comportamento o padrão fazendo git config push.recurseSubmodules on-demand.

Mesclando Mudanças de Submódulo

Se você alterar uma referência de submódulo ao mesmo tempo que outra pessoa, poderá ter alguns problemas. Ou seja, se os históricos dos submódulos divergiram e estão comprometidos com ramificações divergentes em um superprojeto, pode dar um pouco de trabalho consertar.

Se um dos commits for um ancestral direto do outro (uma mesclagem de avanço rápido), o Git simplesmente escolherá o último para a mesclagem, de modo que funcione bem.

No entanto, o Git não tentará nem mesmo uma mesclagem trivial para você. Se os commits do submódulo divergirem e precisarem ser mesclados, você obterá algo parecido com isto:

```
$ git pull
remote: Contando objetos: 2, concluído.
remote: Comprimindo objetos: 100% (1/1), concluído. remoto:
Total 2 (delta 1), reutilizado 2 (delta 1)
Desempacotando objetos: 100% (2/2), concluído.
De https://github.com/chaconinc/MainProject 9a377d1..eb974f8
master -> origin/master
Buscando o aviso do submódulo
DbConnector: Falha ao mesclar o submódulo DbConnector (mesclar os seguintes commits não encontrados)
Mesclagem automática
DbConnector CONFLICT (submódulo): Conflito de mesclagem em
DbConnector Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

Machine Translated by Google

Basicamente, o que aconteceu aqui é que o Git descobriu que as duas ramificações registram pontos no histórico do submódulo que são divergentes e precisam ser mesclados. Ele explica isso como “merge following commits not found”, o que é confuso, mas explicaremos o porquê disso daqui a pouco.

Para resolver o problema, você precisa descobrir em que estado o submódulo deve estar. Estranhamente, o Git realmente não fornece muitas informações para ajudar aqui, nem mesmo os SHA-1s dos commits de ambos os lados do histórico. Felizmente, é simples de descobrir. Se você executar git diff , poderá obter os SHA-1s dos commits registrados em ambas as ramificações que você estava tentando mesclar.

```
$ git diff diff
--cc DbConnector index
eb41d76,c771610..0000000 --- a/
DbConnector +++ b/DbConnector
```

Portanto, neste caso, eb41d76 é o commit em nosso submódulo que **tínhamos** e c771610 é o commit que o upstream tinha. Se entrarmos em nosso diretório de submódulo, ele já deve estar em eb41d76 , pois a mesclagem não o teria tocado. Se, por qualquer motivo, não for, você pode simplesmente criar e verificar uma ramificação apontando para ela.

O importante é o SHA-1 do commit do outro lado. É nisso que você terá que se fundir e resolver. Você pode apenas tentar a mesclagem com o SHA-1 diretamente ou pode criar uma ramificação para ela e tentar mesclar isso. Sugerimos o último, mesmo que apenas para criar uma mensagem de confirmação de mesclagem mais agradável.

Então, iremos para o diretório do submódulo, criaremos uma ramificação chamada “try-merge” com base no segundo SHA-1 do git diff e faremos o merge manualmente.

```
$ cd DbConector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

$ git merge try-merge
Mesclagem automática src/
main.c CONFLICT (conteúdo): Conflito de mesclagem em src/
main.c Pré-imagem gravada para 'src/main.c'
Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

Temos um conflito de mesclagem real aqui, portanto, se resolvermos isso e o confirmarmos, podemos simplesmente atualizar o projeto principal com o resultado.

```
$ vim src/main.c ÿ $ git
add src/main.c $ git commit
-am 'merged our changes'
```

Machine Translated by Google

```
Resolução gravada para 'src/main.c'. [master
9fd905e] fundiu nossas alterações

$ cd .. ÿ $ git
diff ÿ 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 ÿ


$ git commit -m "Mesclar as alterações do Tom" ÿ
[master 10d2c60] Mesclar as alterações do Tom
```

ÿ Primeiro resolvemos o conflito.

ÿ Em seguida, voltamos ao diretório principal do projeto.

ÿ Podemos verificar os SHA-1s novamente.

ÿ Resolva a entrada do submódulo em conflito.

ÿ Confirme nossa mesclagem.

Pode ser um pouco confuso, mas não é realmente muito difícil.

Curiosamente, há outro caso que o Git trata. Se existir um commit de mesclagem no diretório do submódulo que contém **os dois** commits em seu histórico, o Git o sugerirá como uma possível solução. Ele vê que em algum ponto do projeto do submódulo, alguém fundiu branches contendo esses dois commits, então talvez você queira aquele.

É por isso que a mensagem de erro anterior era “merge following commits not found”, porque não foi possível fazer **isso.** É confuso porque quem esperaria que ele tentasse **fazer** isso?

Se encontrar um único commit de mesclagem aceitável, você verá algo como isto:

```
$ git merge origin/master
warning: Falha ao mesclar o submódulo DbConnector (não avanço rápido)
Encontrou uma possível resolução de mesclagem para o submódulo:
9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > mesclamos nossas alterações
Se estiver correto, basta adicioná-lo ao índice, por exemplo, usando:


git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
"Conector Db"


que aceitará esta sugestão.
DbConnector de fusão automática
```

Machine Translated by Google

```
CONFLICT (submódulo): Conflito de mesclagem em DbConnector
Falha na mesclagem automática; corrigir conflitos e, em seguida, confirmar o resultado.
```

O comando sugerido que o Git está fornecendo atualizará o índice como se você tivesse executado git add (o que elimina o conflito) e, em seguida, confirme. Você provavelmente não deveria fazer isso. Você pode facilmente entrar no diretório do submódulo, ver qual é a diferença, avançar para este commit, testá-lo adequadamente e, em seguida, confirmá-lo.

```
$ cd DbConnector/ $
git merge 9fd905e
Atualizando eb41d76..9fd905e
Avanço rápido

$ cd .. $
git add DbConnector $ git
commit -am 'Fast forward to a common submodule child'
```

Isso realiza a mesma coisa, mas pelo menos dessa maneira você pode verificar se funciona e se tem o código no diretório do submódulo quando terminar.

Dicas de Submódulo

Há algumas coisas que você pode fazer para tornar o trabalho com submódulos um pouco mais fácil.

Submódulo Foreach

Existe um comando de submódulo foreach para executar algum comando arbitrário em cada submódulo. Isso pode ser muito útil se você tiver vários submódulos no mesmo projeto.

Por exemplo, digamos que queremos iniciar um novo recurso ou fazer uma correção de bug e temos trabalho em andamento em vários submódulos. Podemos facilmente armazenar todo o trabalho em todos os nossos submódulos.

```
$ submódulo git foreach 'git stash'
Entrando em 'CryptoLibrary'
Nenhuma alteração local para salvar
Inserindo 'DbConnector'
Diretório de trabalho salvo e estado do índice WIP no estável: 82d2ad3 Mesclar de origem/estável

HEAD está agora em 82d2ad3 Mesclar da origem/estável
```

Então podemos criar uma nova ramificação e mudar para ela em todos os nossos submódulos.

```
$ git submodule foreach 'git checkout -b featureA'
Entrando em 'CryptoLibrary'
Mudou para uma nova ramificação 'featureA'
Inserindo 'DbConnector'
```

Machine Translated by Google

```
Mudou para uma nova ramificação 'featureA'
```

Você entendeu a ideia. Uma coisa realmente útil que você pode fazer é produzir uma boa comparação unificada do que foi alterado em seu projeto principal e também em todos os seus subprojetos.

```
$ git diff; git submodule foreach 'git diff'
O submódulo DbConnector contém conteúdo modificado diff --git a/
src/main.cb/src/main.c index 210f1ae..1f0acdc 100644 --- a/src/
main.c +++ b/src/main.c @@ -245,6 +245,8 @@ static int
handle_alias(int *argcp, const char ***argv)




        commit_pager_choice();

+       url = url_decode(url_orig);
+
        /* build alias_argv */ alias_argv
        = xmalloc(sizeof(*alias_argv) * (argc + 1)); alias_argv[0] = alias_string + 1;
        Inserindo diff 'DbConnector' --git a/src/db.cb/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 **consulta) {

            struct strbuf out = STRBUF_INIT;
```

Aqui podemos ver que estamos definindo uma função em um submódulo e chamando-a no projeto principal.
Este é obviamente um exemplo simplificado, mas espero que dê uma ideia de como isso pode ser útil.

Aliases Úteis

Você pode querer configurar alguns aliases para alguns desses comandos, pois eles podem ser muito longos e você não pode definir opções de configuração para a maioria deles para torná-los padrão. Cobrimos a configuração de aliases do Git no Git Aliases, mas aqui está um exemplo do que você pode querer configurar se planeja trabalhar muito com submódulos no Git.

Machine Translated by Google

```
$ git config alias.sdiff '!"'git diff && git submodule foreach 'git diff"' $ git config
alias.spush 'push --recurse-submodules=on-demand' $ git config alias.supdate
'submodule update -- remoto --merge'
```

Dessa forma, você pode simplesmente executar git supdate quando quiser atualizar seus submódulos ou git spush para fazer push com verificação de dependência de submódulo.

Problemas com submódulos

Usar submódulos não é sem soluços, no entanto.

Mudando de ramificação

Por exemplo, alternar ramificações com submódulos também pode ser complicado com versões do Git anteriores ao Git 2.13. Se você criar uma nova ramificação, adicionar um submódulo lá e depois voltar para uma ramificação sem esse submódulo, ainda terá o diretório do submódulo como um diretório não rastreado:

```
$ git --version
git versão 2.12.2

$ git checkout -b add-crypto
Mudou para uma nova ramificação 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Clonagem em 'CryptoLibrary'...
...

$ git commit -am 'Adicionar biblioteca de
criptografia' [add-crypto 4445836] Adicionar
biblioteca de criptografia 2 arquivos alterados, 4
inserções (+) modo de criação 160000 CryptoLibrary

$ git checkout master
warning: incapaz de rmdir CryptoLibrary: Diretório não vazio
Alterado para branch 'master'
Sua filial está atualizada com 'origin/master'.

$ git status
On branch master
Sua branch está atualizada com 'origin/master'.

Arquivos não rastreados:
(use "git add <arquivo>..." para incluir no que será confirmado)

CryptoLibrary/

nada adicionado ao commit, mas arquivos não rastreados presentes (use "git add" para rastrear)
```

Machine Translated by Google

Remover o diretório não é difícil, mas pode ser um pouco confuso tê-lo lá. Se você removê-lo e, em seguida, voltar para a ramificação que possui esse submódulo, será necessário executar submodule update --init para preenchê-lo novamente.

```
$ git clean -ffdx
Removendo CryptoLibrary/

$ git checkout add-crypto
Alterado para ramificação 'add-crypto'

$ ls CryptoLibrary/

$ atualização do submódulo git --init
Caminho do submódulo 'CryptoLibrary': verificado 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile inclui                    scripts          origem
```

Novamente, não é realmente muito difícil, mas pode ser um pouco confuso.

Versões mais recentes do Git (Git >= 2.13) simplificam tudo isso adicionando o sinalizador --recurse-submodules ao comando git checkout , que cuida de colocar os submódulos no estado correto para a ramificação para a qual estamos mudando.

```
$ git --version git
versão 2.13.3

$ git checkout -b add-crypto
Mudou para uma nova ramificação 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary Clonagem em
'CryptoLibrary'...
 ...

$ git commit -am 'Adicionar biblioteca de
 criptografia' [add-crypto 4445836] Adicionar
biblioteca de criptografia 2 arquivos alterados, 4
inserções (+) modo de criação 160000 CryptoLibrary

$ git checkout --recurse-submodules mestre
Comutado para ramificação 'mestre'
Sua filial está atualizada com 'origin/master'.

$ git status
No mestre da filial
Sua filial está atualizada com 'origin/master'.

nada a comprometer, árvore de trabalho limpa
```

Machine Translated by Google

Usar o sinalizador --recurse-submodules do git checkout também pode ser útil quando você trabalha em várias ramificações no superprojeto, cada uma com seu submódulo apontando para diferentes commits. De fato, se você alternar entre branches que registram o submódulo em diferentes commits, ao executar git status o submódulo aparecerá como “modificado” e indicará “novos commits”. Isso ocorre porque, por padrão, o estado do submódulo não é transferido ao alternar ramificações.

Isso pode ser realmente confuso, por isso é uma boa ideia sempre git checkout --recurse-submodules quando seu projeto tiver submódulos. Para versões mais antigas do Git que não possuem o sinalizador --recurse-submodules , após o checkout, você pode usar git submodule update --init --recursive para colocar os submódulos no estado correto.

Felizmente, você pode dizer ao Git (>=2.14) para sempre usar o sinalizador --recurse-submodules definindo a opção de configuração submodule.recurse: git config submodule.recurse true. Conforme observado acima, isso também fará com que o Git recurse em submódulos para cada comando que tenha uma opção --recurse-submodules (exceto git clone).

Alternando de subdiretórios para submódulos

A outra advertência principal que muitas pessoas enfrentam envolve a mudança de subdiretórios para submódulos. Se você estiver rastreando arquivos em seu projeto e quiser movê-los para um submódulo, tome cuidado ou o Git ficará bravo com você. Suponha que você tenha arquivos em um subdiretório do seu projeto e queira alterná-lo para um submódulo. Se você excluir o subdiretório e executar submodule add, o Git gritará com você:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' já existe no índice
```

Você precisa remover o diretório CryptoLibrary primeiro. Então você pode adicionar o submódulo:

```
$ git rm -r CryptoLibrary $
git submodule add https://github.com/chaconinc/CryptoLibrary
Clonagem em 'CryptoLibrary'... remoto: Contagem de objetos: 11,
concluído. remote: Comprimindo objetos: 100% (10/10), concluído.
remoto: Total 11 (delta 0), reutilizado 11 (delta 0)

Desempacotando objetos: 100% (11/11), concluído.
Verificando a conectividade... concluído.
```

Agora suponha que você fez isso em um ramo. Se você tentar voltar para uma ramificação onde esses arquivos ainda estão na árvore real, em vez de um submódulo, receberá este erro:

```
$ git checkout master
error: Os seguintes arquivos da árvore de trabalho não rastreados seriam substituídos pelo
checkout: CryptoLibrary/Makefile CryptoLibrary/includes/crypto.h

...
```

Machine Translated by Google

```
Por favor, mova ou remova-os antes de poder trocar de branch.
Abortando
```

Você pode forçá-lo a mudar com checkout -f, mas tome cuidado para não ter alterações não salvas, pois elas podem ser substituídas por esse comando.

```
$ git checkout -f master warning:
incapaz de rmdir CryptoLibrary: Diretório não vazio
Comutado para ramificação 'mestre'
```

Então, quando você voltar, você obtém um diretório CryptoLibrary vazio por algum motivo e a atualização do submódulo git também pode não consertá-lo. Você pode precisar entrar no diretório do submódulo e executar um git checkout . para obter todos os seus arquivos de volta. Você pode executar isso em um script foreach do submódulo para executá-lo em vários submódulos.

É importante observar que os submódulos hoje em dia mantêm todos os seus dados do Git no diretório .git do projeto principal , portanto, ao contrário de versões muito mais antigas do Git, destruir um diretório de submódulo não perderá nenhum commit ou ramificação que você tinha.

Com essas ferramentas, os submódulos podem ser um método bastante simples e eficaz para desenvolver vários projetos relacionados, mas ainda separados, simultaneamente.

Agrupamento

Embora tenhamos abordado as formas comuns de transferir dados do Git por uma rede (HTTP, SSH, etc.), na verdade há mais uma maneira de fazer isso que não é comumente usada, mas pode ser bastante útil.

O Git é capaz de “empacotar” seus dados em um único arquivo. Isso pode ser útil em vários cenários. Talvez sua rede esteja inativa e você queira enviar alterações para seus colegas de trabalho. Talvez você esteja trabalhando em algum lugar externo e não tenha acesso à rede local por motivos de segurança. Talvez sua placa wireless/ethernet tenha quebrado. Talvez você não tenha acesso a um servidor compartilhado no momento, deseja enviar atualizações por e-mail para alguém e não deseja transferir 40 commits via patch de formato.

É aqui que o comando git bundle pode ser útil. O comando bundle empacotará tudo o que normalmente seria enviado via cabo com um comando git push em um arquivo binário que você pode enviar por e-mail para alguém ou colocar em uma unidade flash e, em seguida, desempacotar em outro repositório.

Vejamos um exemplo simples. Digamos que você tenha um repositório com dois commits:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102 Autor:
Scott Chacon <schacon@gmail.com> Data: Quarta, 10 de
março, 07:34:10 2010 -0800

Segunda confirmação

commit b1ec3248f39900d2a406049d762aa68e9641be25 Autor:
Scott Chacon <schacon@gmail.com>
```

Machine Translated by Google

```
Data: Quarta-feira, 10 de março 07:34:01 2010 -0800

Primeiro commit
```

Se você deseja enviar esse repositório para alguém e não tem acesso a um repositório para enviar, ou simplesmente não deseja configurar um, você pode empacotá-lo com git bundle create .

```
$ git bundle create repo.bundle HEAD master Contando
objetos: 6, concluído.
Compressão Delta usando até 2 threads.
Comprimindo objetos: 100% (2/2), concluído.
Escrevendo objetos: 100% (6/6), 441 bytes, concluído.
Total 6 (delta 0), reutilizado 0 (delta 0)
```

Agora você tem um arquivo chamado repo.bundle que contém todos os dados necessários para recriar a ramificação principal do repositório . Com o comando bundle , você precisa listar todas as referências ou intervalos específicos de commits que deseja incluir. Se você pretende que isso seja clonado em outro lugar, você deve adicionar HEAD como referência, assim como fizemos aqui.

Você pode enviar este arquivo repo.bundle por e-mail para outra pessoa ou colocá-lo em uma unidade USB e acompanhá-lo.

Por outro lado, digamos que você recebeu este arquivo repo.bundle e deseja trabalhar no projeto. Você pode clonar do arquivo binário em um diretório, da mesma forma que faria a partir de um URL.

```
$ git clone repo.bundle repo
Clonagem em 'repo'...
 ...
$ cd repo
$ git log --oneline
9a466c5 Segundo commit
b1ec324 Primeiro commit
```

Se você não incluir HEAD nas referências, também deverá especificar -b master ou qualquer ramificação incluída, porque, caso contrário, não saberá qual ramificação verificar.

Agora, digamos que você faça três commits nele e queira enviar os novos commits de volta por meio de um pacote em um pendrive ou e-mail.

```
$ git log --oneline
71b84da Último commit - segundo
repositório c99cf5b Quarto commit - segundo
repositório 7011d3d Terceiro commit -
segundo repositório 9a466c5 Segundo commit
b1ec324 Primeiro commit
```

Primeiro, precisamos determinar o intervalo de commits que queremos incluir no pacote. Ao contrário dos protocolos de rede que determinam o conjunto mínimo de dados a serem transferidos pela rede para nós,

Machine Translated by Google

teremos que descobrir isso manualmente. Agora, você pode simplesmente fazer a mesma coisa e agrupar todo o repositório, o que funcionará, mas é melhor apenas agrupar a diferença - apenas os três commits que acabamos de fazer localmente.

Para fazer isso, você terá que calcular a diferença. Conforme descrevemos em Intervalos de confirmação, você pode especificar um intervalo de confirmações de várias maneiras. Para obter os três commits que temos em nosso branch master que não estavam no branch que originalmente clonamos, podemos usar algo como origin/master..master ou master ^origin/master. Você pode testar isso com o comando log .

```
$ git log --oneline master ^origin/master 71b84da
Último commit - segundo repositório c99cf5b Quarto
commit - segundo repositório 7011d3d Terceiro
commit - segundo repositório
```

Agora que temos a lista de commits que queremos incluir no pacote, vamos agrupá-los. Fazemos isso com o comando git bundle create , dando a ele um nome de arquivo que queremos que nosso pacote tenha e o intervalo de commits que queremos incluir nele.

```
$ git bundle create commits.bundle master ^9a466c5 Contando
objetos: 11, concluído.
Compressão Delta usando até 2 threads.
Comprimindo objetos: 100% (3/3), concluído.
Objetos de escrita: 100% (9/9), 775 bytes, concluído.
Total 9 (delta 0), reutilizado 0 (delta 0)
```

Agora temos um arquivo commits.bundle em nosso diretório. Se pegarmos isso e enviarmos para nosso parceiro, ela poderá importá-lo para o repositório original, mesmo que mais trabalho tenha sido feito nesse meio tempo.

Quando ela recebe o pacote, ela pode inspecioná-lo para ver o que ele contém antes de importá-lo para seu repositório. O primeiro comando é o comando bundle Verify que garantirá que o arquivo seja realmente um pacote Git válido e que você tenha todos os ancestrais necessários para reconstituí-lo adequadamente.

```
$ git bundle Verify ../commits.bundle O bundle
contém 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
O pacote requer estes 1 segundo
commit ref 9a466c572fe88b195efd356c3f2bbeccdb504102
../commits.bundle está ok
```

Se o empacotador tivesse criado um pacote com apenas os dois últimos commits que ele fez, em vez de todos os três, o repositório original não seria capaz de importá-lo, pois falta o histórico de requisitos. Em vez disso, o comando de verificação ficaria assim:

```
$ git bundle Verify ../commits-bad.bundle error: O
repositório não possui estes commits de pré-requisito:
```

Machine Translated by Google

```
erro: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Terceiro commit - segundo repo
```

No entanto, nosso primeiro bundle é válido, então podemos buscar commits dele. Se você quiser ver quais branches estão no pacote que podem ser importados, há também um comando para listar apenas os cabeçalhos:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

O subcomando de verificação também informará as cabeças. O objetivo é ver o que pode ser obtido, para que você possa usar os comandos fetch ou pull para importar commits deste pacote. Aqui vamos buscar o branch master do bundle para um branch chamado other-master em nosso repositório:

```
$ git fetch ../commits.bundle master:other-master From ../
commits.bundle * [new branch]
                                    mestre          -> outro mestre
```

Agora podemos ver que temos os commits importados no branch other-master , bem como quaisquer commits que fizemos nesse meio tempo em nosso próprio branch master .

```
$ git log --oneline --decorate --graph --all * 8255d41
(HEAD, master) Terceiro commit - primeiro repositório | *
 71b84da (outro-mestre) Último commit - segundo repositório | *
 c99cf5b Quarto commit - segundo repositório | * 7011d3d
 Terceiro commit - segundo repositório |/ * 9a466c5 Segundo
 commit * b1ec324 Primeiro commit
```

Portanto, git bundle pode ser muito útil para compartilhar ou fazer operações do tipo rede quando você não tem a rede adequada ou o repositório compartilhado para fazer isso.

Substituir

Como enfatizamos antes, os objetos no banco de dados de objetos do Git são imutáveis, mas o Git fornece uma maneira interessante de *simular* a substituição de objetos em seu banco de dados por outros objetos.

O comando replace permite especificar um objeto no Git e dizer "toda vez que você se referir a *este* objeto, finja que é um objeto *diferente* ". Isso é mais útil para substituir um commit em seu histórico por outro sem ter que reconstruir todo o histórico com, digamos, git filter-branch.

Por exemplo, digamos que você tenha um grande histórico de código e queira dividir seu repositório em um pequeno histórico para novos desenvolvedores e outro muito mais longo e amplo para pessoas interessadas em mineração de dados. Você pode enxertar um histórico no outro "substituindo" o commit mais antigo na nova linha pelo commit mais recente no antigo. Isso é bom porque significa que você não precisa reescrever todos os commits no novo histórico, como normalmente teria que fazer para juntá-los

Machine Translated by Google

(porque o parentesco afeta os SHA-1s).

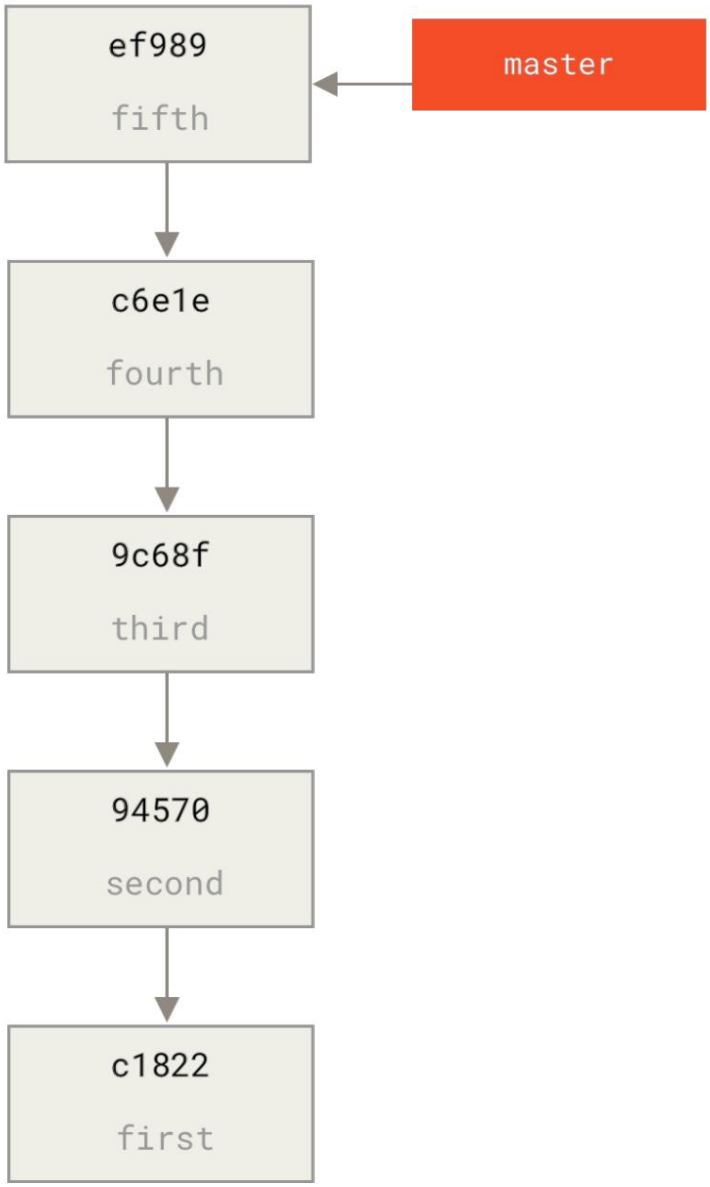
Vamos tentar isso. Vamos pegar um repositório existente, dividi-lo em dois repositórios, um recente e um histórico, e então veremos como podemos recombiná-los sem modificar os valores SHA-1 dos repositórios recentes via replace .

Usaremos um repositório simples com cinco commits simples:

```
$ git log --oneline
ef989d8 Quinto commit
c6e1e95 Quarto commit
9c68fdc Terceiro commit
945704c Segundo commit
c1822cf Primeiro commit
```

Queremos dividir isso em duas linhas da história. Uma linha vai do commit um ao commit quatro - esse será o histórico. A segunda linha será apenas os commits quatro e cinco - esse será o histórico recente.

Machine Translated by Google

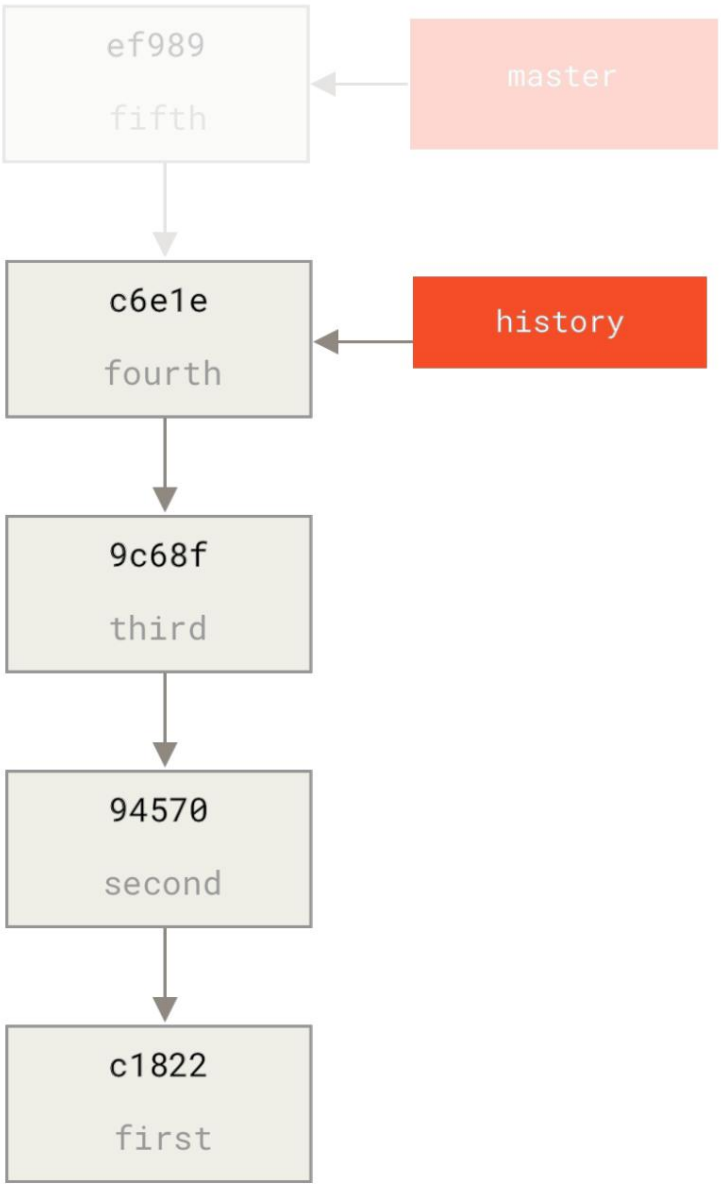


Bem, criar o histórico histórico é fácil, podemos apenas colocar uma ramificação no histórico e, em seguida, enviar essa ramificação para a ramificação principal de um novo repositório remoto.

```
$ git branch history c6e1e95 $ git log --
oneline --decorate ef989d8 (HEAD,
master) Quinto commit c6e1e95 (history) Quarto
commit 9c68fdc Terceiro commit

945704c Segundo commit
c1822cf Primeiro commit
```

Machine Translated by Google



Agora podemos enviar o novo branch de histórico para o branch master do nosso novo repositório:

```
$ git remote add project-history https://github.com/schacon/project-history $ git push project-history
history:master Counting objects: 12, done.

Compressão Delta usando até 2 threads.
Comprimindo objetos: 100% (4/4), concluído.
Escrevendo objetos: 100% (12/12), 907 bytes, concluído.
Total 12 (delta 0), reutilizado 0 (delta 0)
Desempacotando objetos: 100% (12/12), concluído.
Para git@github.com:schacon/project-history.git * [new
branch] history -> master
```

OK, então nossa história é publicada. Agora, a parte mais difícil é truncar nossa história recente, então é

Machine Translated by Google

menor. Precisamos de uma sobreposição para que possamos substituir um commit em um por um commit equivalente no outro, então vamos truncar isso para apenas os commits quatro e cinco (portanto, comite quatro sobreposições).

```
$ git log --oneline --decorate ef989d8
(HEAD, master) Quinto commit c6e1e95 (história)
Quarto commit 9c68fdc Terceiro commit

945704c Segundo commit
c1822cf Primeiro commit
```

É útil neste caso criar um commit básico que tenha instruções sobre como expandir o histórico, para que outros desenvolvedores saibam o que fazer se acertarem o primeiro commit no histórico truncado e precisarem de mais. Então, o que vamos fazer é criar um objeto commit inicial como nosso ponto de base com instruções, então rebase os commits restantes (quatro e cinco) em cima dele.

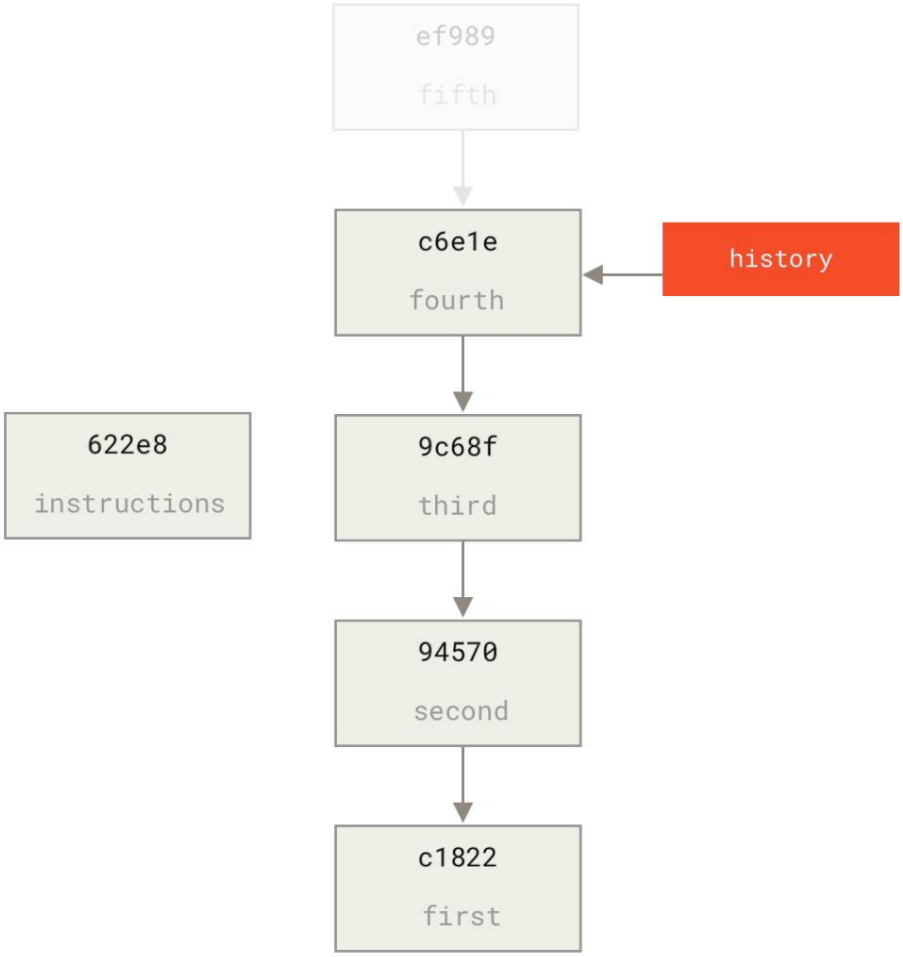
Para fazer isso, precisamos escolher um ponto para dividir, que para nós é o terceiro commit, que é 9c68fdc na linguagem SHA. Portanto, nosso commit base será baseado nessa árvore. Podemos criar nosso commit básico usando o comando commit-tree , que apenas pega uma árvore e nos dará um novo objeto de commit sem pai SHA-1 de volta.

```
$ echo 'Obter histórico de blá blá blá' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

ÿ

O comando commit-tree faz parte de um conjunto de comandos comumente referidos como comandos de 'encanamento'. Esses são comandos que geralmente não devem ser usados diretamente, mas, em vez disso, são usados por **outros** comandos do Git para fazer trabalhos menores.
Em ocasiões em que estamos fazendo coisas mais estranhas como essa, eles nos permitem fazer coisas realmente de baixo nível, mas não são para uso diário. Você pode ler mais sobre os comandos de encanamento em Encanamento e Porcelana.

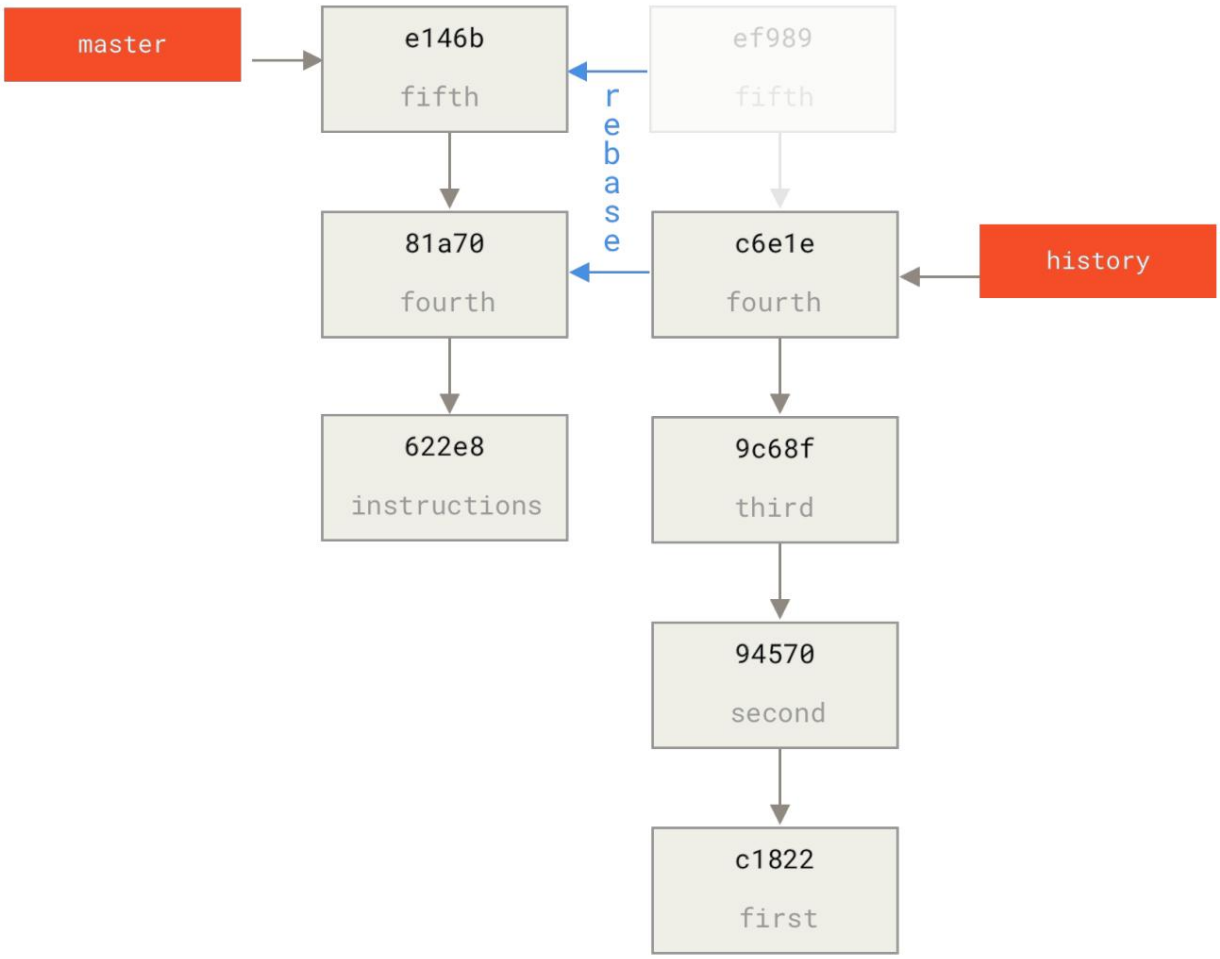
Machine Translated by Google



OK, agora que temos um commit base, podemos rebasear o resto do nosso histórico com git rebase --onto. O argumento --onto será o SHA-1 que acabamos de receber da commit-tree e o ponto de rebase será o terceiro commit (o pai do primeiro commit que queremos manter, 9c68fdc):

```
$ git rebase --onto 622e88 9c68fdc
Primeiro, rebobinando a cabeça para reproduzir seu trabalho em cima dela...
Aplicando: quarto commit
Aplicando: quinto commit
```

Machine Translated by Google



OK, agora nós reescrevemos nosso histórico recente em cima de um commit base descartável que agora contém instruções sobre como reconstituir todo o histórico, se quisermos. Podemos enviar esse novo histórico para um novo projeto e agora, quando as pessoas clonarem esse repositório, elas verão apenas os dois commits mais recentes e, em seguida, um commit básico com instruções.

Vamos agora trocar os papéis para alguém que está clonando o projeto pela primeira vez e que deseja o histórico completo. Para obter os dados do histórico após a clonagem deste repositório truncado, seria necessário adicionar um segundo controle remoto para o repositório histórico e buscar:

```
$ git clone https://github.com/schacon/project $ cd
project

$ git log --oneline master
e146b5f Quinto commit
81a708d Quarto commit
622e88e Obter histórico de blá blá blá

$ git remote add project-history https://github.com/schacon/project-history $ git fetch
project-history De https://github.com/schacon/project-history * [new branch] master

                                    -> histórico do projeto/mestre
```

Machine Translated by Google

Agora o colaborador teria seus commits recentes no branch master e o histórico commits no branch project-history/master .

```
$ git log --oneline master
e146b5f Quinto commit 81a708d
Quarto commit 622e88e Obter
histórico de blá blá blá

$ git log --oneline project-history/master c6e1e95
Quarto commit 9c68fdc Terceiro commit

945704c Segundo commit
c1822cf Primeiro commit
```

Para combiná-los, você pode simplesmente chamar git replace com o commit que deseja substituir e depois o commit com o qual deseja substituí-lo. Portanto, queremos substituir o "quarto" commit no branch master pelo "quarto" commit no branch project-history/master :

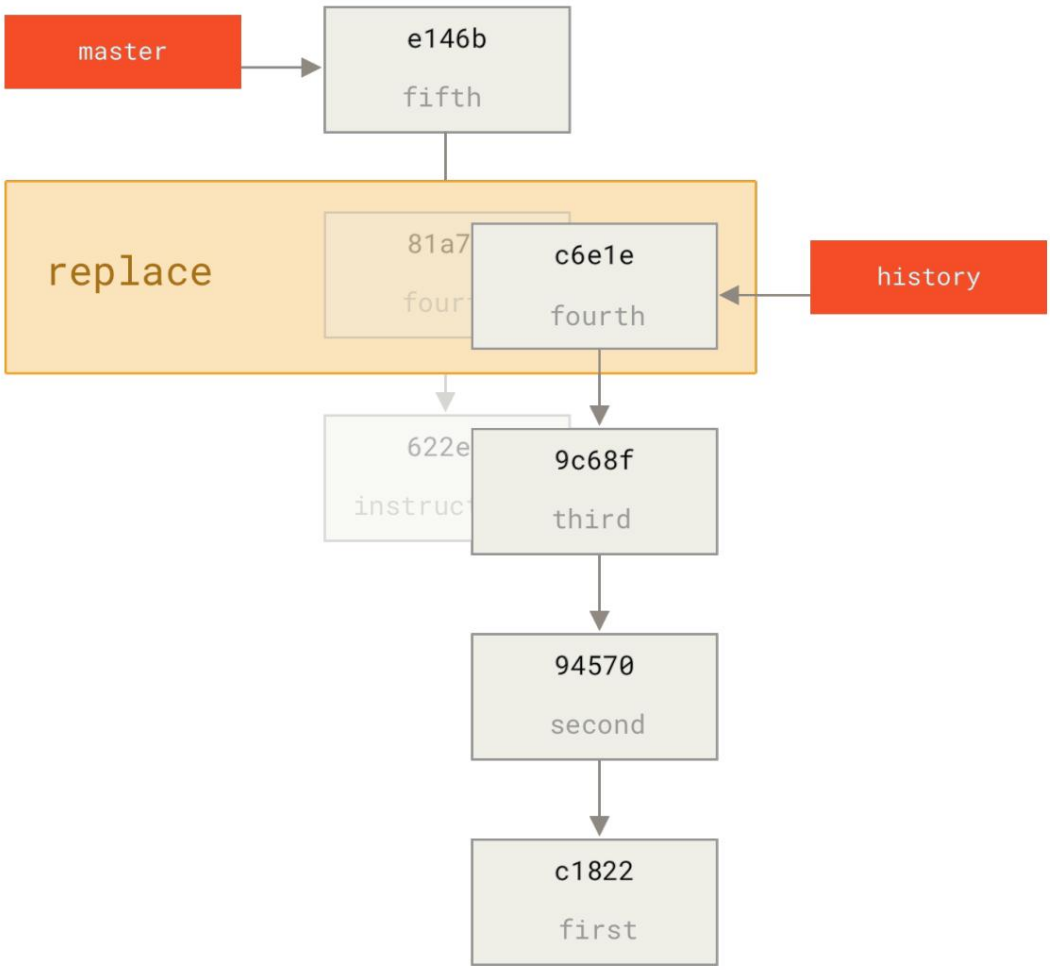
```
$ git substituir 81a708d c6e1e95
```

Agora, se você observar o histórico do branch master , ele parecerá assim:

```
$ git log --oneline master
e146b5f Quinto commit 81a708d
Quarto commit 9c68fdc Terceiro
commit 945704c Segundo
commit
c1822cf Primeiro commit
```

Legal certo? Sem ter que alterar todos os SHA-1 upstream, fomos capazes de substituir um commit em nosso histórico por um commit totalmente diferente e todas as ferramentas normais (bisect, culpe, etc) funcionarão como esperávamos.

Machine Translated by Google



Curiosamente, ele ainda mostra 81a708d como o SHA-1, embora esteja realmente usando os dados de confirmação c6e1e95 com os quais os substituímos. Mesmo se você executar um comando como cat-file, ele mostrará os dados substituídos:

```
$ git cat-file -p 81a708d árvore
7bc544cf438903b65ca9104a1e30345eee6c083d
pai 9c68fdceee073230f19ebb8b5e7fc71b479c0252 autor
Scott Chacon <schacon@gmail.com> 1268712581 -0700 committer
Scott Chacon <schacon@gmail.com> 1268712581 -0700

quarto compromisso
```

Lembre-se de que o pai real de 81a708d era nosso commit de espaço reservado (622e88e), não 9c68fdce como afirma aqui.

Outra coisa interessante é que esses dados são mantidos em nossas referências:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
```

Machine Translated by Google

```
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/
81a708dd0e167a3f691541c7a6463343bc457040
```

Isso significa que é fácil compartilhar nossa substituição com outras pessoas, porque podemos enviá-la para nosso servidor e outras pessoas podem baixá-la facilmente. Isso não é tão útil no cenário de enxerto de histórico que examinamos aqui (já que todos estariam baixando os dois históricos de qualquer maneira, então por que separá-los?), mas pode ser útil em outras circunstâncias.

Armazenamento de credenciais

Se você usar o transporte SSH para conectar-se a controles remotos, é possível que você tenha uma chave sem senha, o que permite transferir dados com segurança sem digitar seu nome de usuário e senha. No entanto, isso não é possível com os protocolos HTTP – toda conexão precisa de um nome de usuário e senha. Isso fica ainda mais difícil para sistemas com autenticação de dois fatores, onde o token que você usa para uma senha é gerado aleatoriamente e impronunciável.

Felizmente, o Git possui um sistema de credenciais que pode ajudar nisso. O Git tem algumas opções fornecidas na caixa:

- O padrão é não armazenar em cache. Cada conexão solicitará seu nome de usuário e senha.
- O modo “cache” mantém as credenciais na memória por um determinado período de tempo. Nenhuma das senhas é armazenada em disco e são removidas do cache após 15 minutos.
- O modo “armazenar” salva as credenciais em um arquivo de texto simples no disco e elas nunca expiram. Isso significa que, até que você altere sua senha para o host Git, nunca mais precisará digitar suas credenciais novamente. A desvantagem dessa abordagem é que suas senhas são armazenadas em texto não criptografado em um arquivo simples em seu diretório pessoal.
- Se você estiver usando um Mac, o Git vem com um modo “osxkeychain”, que armazena as credenciais no chaveiro seguro anexado à sua conta do sistema. Esse método armazena as credenciais no disco e elas nunca expiram, mas são criptografadas com o mesmo sistema que armazena certificados HTTPS e preenchimentos automáticos do Safari.
- Se estiver usando o Windows, você pode habilitar o recurso **Git Credential Manager** ao instalar o Git para Windows ou instale separadamente o GCM mais recente como um serviço independente. Isso é semelhante ao auxiliar “osxkeychain” descrito acima, mas usa o Windows Credential Store para controlar informações confidenciais. Ele também pode fornecer credenciais para WSL1 ou WSL2. Veja a instalação do GCM Instruções Para maiores informações.

Você pode escolher um destes métodos definindo um valor de configuração do Git:

```
$ git config --global credential.helper cache
```

Alguns desses ajudantes têm opções. O auxiliar “store” pode receber um argumento --file <path> , que personaliza onde o arquivo de texto simples é salvo (o padrão é ~/.git-credentials). O auxiliar de “cache”

Machine Translated by Google

aceita a opção --timeout <segundos> , que altera a quantidade de tempo que seu daemon é mantido em execução (o padrão é “900” ou 15 minutos). Aqui está um exemplo de como você configuraria o auxiliar “store” com um nome de arquivo personalizado:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

O Git ainda permite que você configure vários ajudantes. Ao procurar credenciais para um host específico, o Git as consultará em ordem e parará após a primeira resposta ser fornecida. Ao salvar as credenciais, o Git enviará o nome de usuário e a senha para **todos** os ajudantes da lista, e eles poderão escolher o que fazer com eles. Veja como seria um .gitconfig se você tivesse um arquivo de credenciais em um pen drive, mas quisesse usar o cache na memória para economizar digitação se o drive não estivesse conectado:

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials helper = cache --
timeout 30000
```

sob o capô

Como tudo isso funciona? O comando raiz do Git para o sistema auxiliar de credenciais é git credential, que recebe um comando como argumento e, em seguida, mais entradas por meio de stdin.

Isso pode ser mais fácil de entender com um exemplo. Digamos que um auxiliar de credencial foi configurado e o auxiliar armazenou credenciais para mygithost. Aqui está uma sessão que usa o comando “fill”, que é invocado quando o Git está tentando encontrar credenciais para um host:

```
$ git credencial fill ÿ
protocol=https ÿ host=mygithost
ÿ protocol=https ÿ
host=mygithost username=bob



password=s3cre7
$ git credencial fill ÿ
protocol=https
host=unknownhost

Nome de usuário para 'https://unknownhost': bob
Senha para 'https://bob@unknownhost':
protocol=https host=unknownhost username=bob
password=s3cre7
```

ÿ Esta é a linha de comando que inicia a interação.

ÿ A credencial do Git está aguardando a entrada no stdin. Fornecemo-lo com as coisas que sabemos: o

Machine Translated by Google

protocolo e nome do host.

ÿ Uma linha em branco indica que a entrada foi concluída e o sistema de credenciais deve responder com o que sabe.

ÿ Git-credential assume o controle e grava em stdout com as informações encontradas.

ÿ Se as credenciais não forem encontradas, o Git solicitará ao usuário o nome de usuário e a senha e fornecerá eles de volta ao stdout de chamada (aqui eles estão anexados ao mesmo console).

O sistema de credenciais está, na verdade, invocando um programa separado do próprio Git; qual e como depende do valor de configuração credential.helper . Existem várias formas que pode assumir:

Valor de configuração	Comportamento
foo	Executa git-credential-foo
foo -a --opt=bcd	Executa git-credential-foo -a --opt=bcd
/absoluto/caminho/foo -xyz	Executa /absolute/path/foo -xyz
!f() { echo "senha=s3cre7"; }; f	Código depois ! avaliado em shell

Portanto, os auxiliares descritos acima são, na verdade, denominados git-credential-cache, git-credential-store e assim por diante, e podemos configurá-los para aceitar argumentos de linha de comando. A forma geral para isso é “git-credential-foo [args] <action>”. O protocolo stdin/stdout é o mesmo que git-credential, mas eles usam um conjunto ligeiramente diferente de ações:

- get é uma solicitação para um par de nome de usuário/senha.
- store é uma solicitação para salvar um conjunto de credenciais na memória desse auxiliar.
- apagar limpar as credenciais para as propriedades fornecidas da memória deste auxiliar.

Para as ações store e erase , nenhuma resposta é necessária (o Git a ignora de qualquer maneira). Para a ação get , no entanto, o Git está muito interessado no que o auxiliar tem a dizer. Se o auxiliar não souber nada útil, ele pode simplesmente sair sem saída, mas se souber, deve aumentar as informações fornecidas com as informações que armazenou. A saída é tratada como uma série de instruções de atribuição; qualquer coisa fornecida substituirá o que o Git já conhece.

Aqui está o mesmo exemplo acima, mas pulando git-credential e indo direto para git credential-store:

```
$ git credential-store --file ~/git.store store ÿ protocol=https
host=mygithost username=bob password=s3cre7 $ git credential-
store --file ~/git.store get ÿ protocol=https host=mygithost




nome de
usuário=bob ÿ senha=s3cre7
```

Machine Translated by Google

ÿ Aqui dizemos ao git-credential-store para salvar algumas credenciais: o nome de usuário “bob” e o a senha “s3cre7” deve ser usada quando https://mygithost é acessado.

ÿ Agora vamos recuperar essas credenciais. Fornecemos as partes da conexão que já conhecemos (https://mygithost), e uma linha vazia.

ÿ git-credential-store responde com o nome de usuário e a senha que armazenamos acima.

Aqui está a aparência do arquivo ~/git.store :

```
https://bob:s3cre7@mygithost
```

É apenas uma série de linhas, cada uma contendo um URL decorado com credenciais. Os auxiliares osxkeychain e wincred usam o formato nativo de seus armazenamentos de apoio, enquanto o cache usa seu próprio formato na memória (que nenhum outro processo pode ler).

Um cache de credenciais personalizado

Dado que git-credential-store e friends são programas separados do Git, não é difícil perceber que *qualquer* programa pode ser um auxiliar de credencial do Git. Os helpers fornecidos pelo Git abrangem muitos casos de uso comuns, mas não todos. Por exemplo, digamos que sua equipe tenha algumas credenciais que são compartilhadas com toda a equipe, talvez para implantação. Eles são armazenados em um diretório compartilhado, mas você não deseja copiá-los para seu próprio armazenamento de credenciais, porque eles mudam com frequência. Nenhum dos ajudantes existentes cobre este caso; vamos ver o que seria necessário para escrever o nosso próprio. Existem vários recursos principais que este programa precisa ter:

1. A única ação à qual precisamos prestar atenção é obter; armazenar e apagar são operações de gravação, então vamos basta sair de forma limpa quando eles são recebidos.
2. O formato do arquivo de credencial compartilhada é o mesmo usado por git-credential-store.
3. A localização desse arquivo é bastante padrão, mas devemos permitir que o usuário passe um caminho personalizado apenas no caso de.

Mais uma vez, escreveremos esta extensão em Ruby, mas qualquer linguagem funcionará desde que o Git possa executar o produto final. Aqui está o código-fonte completo do nosso novo auxiliar de credencial:

```
#!/usr/bin/env rubi

requer 'optparse'

path = File.expand_path '~/.git-credentials' ÿ
OptionParser.new do |opts| opts.banner = 'USO: git-
credential-read-only [opções] <ação>' opts.on('-f', '--file PATH', 'Especificar
caminho para armazenamento de apoio') do |argpath| path = File.expand_path argpath

fim
end.parse!

exit(0) a menos que ARGV[0].downcase == 'obter' ÿ
exit(0) a menos que File.existir? caminho
```

```
known = {} ÿ while
line = STDIN.gets break if line.strip
== k,v = line.strip.split '=', 2 known[k] = v
end




File.readlines(path).cada um faz |fileline| ÿ prot,user,pass,host =
fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first if prot == conhecido['protocolo'] e host == conhecido['host'] e usuário ==
conhecido['nome de usuário'] então

                puts "protocol=#{prot}" puts
                "host=#{host}" puts "username=#{user}"
                puts "password=#{pass}" exit(0)



fim
fim
```

ÿ Aqui analisamos as opções de linha de comando, permitindo que o usuário especifique o arquivo de entrada. O padrão é ~/.git-credentials.

ÿ Este programa só responde se a ação for get e o arquivo backing-store existir.

ÿ Este loop lê de stdin até que a primeira linha em branco seja alcançada. As entradas são armazenadas no conhecido hash para referência posterior.

ÿ Este loop lê o conteúdo do arquivo de armazenamento, procurando correspondências. Se o protocolo, o host e o nome de usuário de known corresponderem a essa linha, o programa imprimirá os resultados em stdout e sairá.

Vamos salvar nosso auxiliar como git-credential-read-only, colocá-lo em algum lugar em nosso PATH e marcá-lo como executável. Veja como é uma sessão interativa:

```
$ git credential-read-only --file=/mnt/shared/creds get protocol=https host=mygithost
username=bob




protocol=https
host=mygithost
username=bob
password=s3cre7
```

Como seu nome começa com “git-”, podemos usar a sintaxe simples para o valor de configuração:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

Como você pode ver, estender esse sistema é bastante simples e pode resolver alguns problemas comuns para você e sua equipe.

Machine Translated by Google

Resumo

Você já viu várias ferramentas avançadas que permitem manipular seus commits e área de preparação com mais precisão. Ao notar problemas, você deve ser capaz de descobrir facilmente qual commit os introduziu, quando e por quem. Se quiser usar subprojetos em seu projeto, você aprendeu como atender a essas necessidades. Neste ponto, você deve ser capaz de fazer a maioria das coisas no Git que você precisa na linha de comando do dia a dia e se sentir confortável fazendo isso.

Machine Translated by Google

Personalizando o Git

Até agora, cobrimos o básico de como o Git funciona e como usá-lo, e apresentamos várias ferramentas que o Git fornece para ajudá-lo a usá-lo de maneira fácil e eficiente. Neste capítulo, veremos como você pode fazer o Git operar de maneira mais personalizada, introduzindo várias definições de configuração importantes e o sistema de ganchos. Com essas ferramentas, é fácil fazer o Git funcionar exatamente da maneira que você, sua empresa ou seu grupo precisam.

Configuração do Git

Conforme você lê brevemente em Introdução, você pode especificar as definições de configuração do Git com o comando git config . Uma das primeiras coisas que você fez foi configurar seu nome e endereço de e-mail:

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

Agora você aprenderá algumas das opções mais interessantes que podem ser definidas dessa maneira para personalizar o uso do Git.

Primeiro, uma revisão rápida: o Git usa uma série de arquivos de configuração para determinar o comportamento não padrão que você pode desejar. O primeiro lugar onde o Git procura por esses valores é no arquivo [caminho]/etc/gitconfig de todo o sistema , que contém configurações que são aplicadas a todos os usuários do sistema e a todos os seus repositórios. Se você passar a opção --system para git config, ele lê e escreve deste arquivo especificamente.

O próximo local onde o Git procura é o arquivo ~/.gitconfig (ou ~/.config/git/config) , que é específico para cada usuário. Você pode fazer o Git ler e gravar neste arquivo passando a opção --global .

Por fim, o Git procura valores de configuração no arquivo de configuração no diretório Git (.git/config) de qualquer repositório que você esteja usando no momento. Esses valores são específicos desse único repositório e representam a passagem da opção --local para git config. Se você não especificar com qual nível deseja trabalhar, este é o padrão.

Cada um desses “níveis” (sistema, global, local) sobrescreve os valores do nível anterior, então os valores em .git/config superam aqueles em [path]/etc/gitconfig, por exemplo.

ÿ

Os arquivos de configuração do Git são de texto simples, portanto, você também pode definir esses valores editando manualmente o arquivo e inserindo a sintaxe correta. No entanto , geralmente é mais fácil executar o comando git config .

Configuração básica do cliente

As opções de configuração reconhecidas pelo Git se enquadram em duas categorias: do lado do cliente e do lado do servidor. A maioria das opções são do lado do cliente — configurando suas preferências pessoais de trabalho. Muitas, *muitas* opções de configuração são suportadas, mas uma grande fração delas é útil apenas em certos casos extremos; abordaremos apenas as opções mais comuns e úteis aqui. Se você quiser ver uma lista de todas as opções que sua versão do Git reconhece, você pode executar:

Machine Translated by Google

```
$ man git-config
```

Este comando lista todas as opções disponíveis com bastante detalhes. Você também pode encontrar este material de referência em https://git-scm.com/docs/git-config.

core.editor

Por padrão, o Git usa o que você definiu como seu editor de texto padrão por meio de uma das variáveis de ambiente do shell VISUAL ou EDITOR, ou então recorre ao editor vi para criar e editar suas mensagens de confirmação e marcação. Para alterar esse padrão para outra coisa, você pode usar a configuração core.editor :

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

Agora, não importa o que esteja definido como seu editor de shell padrão, o Git iniciará o Emacs para editar mensagens.

commit.template

Se você definir isso como o caminho de um arquivo em seu sistema, o Git usará esse arquivo como a mensagem inicial padrão quando você confirmar. O valor de criar um modelo de commit personalizado é que você pode usá-lo para lembrar a si mesmo (ou a outras pessoas) do formato e estilo adequados ao criar uma mensagem de commit.

Por exemplo, considere um arquivo de modelo em ~/.gitmessage.txt que se parece com isto:

```
Linha de assunto (tente manter menos de 50 caracteres)

Descrição multilinha do commit, sinta-se
à vontade para detalhar.

[Bilhete: X]
```

Observe como este modelo de commit lembra o committer para manter a linha de assunto curta (pelo bem do git log --oneline output), para adicionar mais detalhes sob isso e para se referir a um número de problema ou rastreador de bug, se houver.

Para dizer ao Git para usá-lo como a mensagem padrão que aparece em seu editor quando você executa git commit, defina o valor de configuração commit.template :

```
$ git config --global commit.template ~/.gitmessage.txt $ git commit
```

Em seguida, seu editor abrirá algo como isto para sua mensagem de confirmação de espaço reservado quando você confirmar:

```
Linha de assunto (tente manter menos de 50 caracteres)

Descrição multilinha do commit,
```

Machine Translated by Google

```
sinta-se livre para ser detalhado.

[Bilhete: X]
# Por favor, insira a mensagem de confirmação para suas alterações. Linhas
começando com # com '#' serão ignoradas, e uma mensagem vazia aborta o commit.
# No mestre da ramificação
# Mudanças a serem confirmadas:
# (use "git reset HEAD <arquivo>..." para retirar o estágio) #

# modificado: lib/test.rb #

~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Se sua equipe tiver uma política de mensagem de confirmação, colocar um modelo para essa política em seu sistema e configurar o Git para usá-la por padrão pode ajudar a aumentar a chance dessa política ser seguida regularmente.

core.pager

Essa configuração determina qual pager é usado quando as páginas do Git são geradas, como log e diff. Você pode configurá-lo para mais ou para seu pager favorito (por padrão, é menos) ou pode desativá-lo definindo-o como uma string em branco:

```
$ git config --global core.pager "
```

Se você executar isso, o Git paginará toda a saída de todos os comandos, não importa quanto tempo eles tenham.

usuário.chave de assinatura

Se você estiver fazendo tags anotadas assinadas (conforme discutido em Assinando seu trabalho), definir sua chave de assinatura GPG como uma definição de configuração facilita as coisas. Defina seu ID de chave assim:

```
$ git config --global user.signingkey <gpg-key-id>
```

Agora, você pode assinar tags sem precisar especificar sua chave todas as vezes com o comando git tag :

```
$ git tag -s <nome da tag>
```

arquivo core.excludes

Você pode colocar padrões no arquivo .gitignore do seu projeto para que o Git não os veja como arquivos não rastreados ou tente organizá-los quando executar git add neles, conforme discutido em Ignorando arquivos.

Mas às vezes você deseja ignorar determinados arquivos para todos os repositórios com os quais trabalha. Se o seu computador estiver executando o macOS, você provavelmente está familiarizado com os arquivos .DS_Store . Se o seu editor preferido

Machine Translated by Google

seja Emacs ou Vim, você conhece nomes de arquivos que terminam com ~ ou .swp.

Essa configuração permite que você escreva um tipo de arquivo .gitignore global . Se você criar um arquivo ~/.gitignore_global com este conteúdo:

```
*~

.*.swp .DS_Store
```

…e você executar git config --global core.excludesfile ~/.gitignore_global, o Git nunca mais o incomodará com esses arquivos.

ajuda.correção automática

Se você digitar um comando errado, ele mostrará algo como isto:

```
$ git chekcout master git:
'chekcout' não é um comando git. Consulte 'git --help'.

O comando mais semelhante é
checkout
```

O Git tenta ajudar a descobrir o que você quis dizer, mas ainda se recusa a fazê-lo. Se você definir help.autocorrect como 1, o Git executará este comando para você:

```
$ git chekcout master
AVISO: Você chamou um comando Git chamado 'chekcout', que não existe.
Continuando supondo que você quis dizer 'checkout' em 0,1
segundos automaticamente...
```

Observe o negócio de “0,1 segundos”. help.autocorrect é, na verdade, um número inteiro que representa décimos de segundo. Portanto, se você definir como 50, o Git lhe dará 5 segundos para mudar de ideia antes de executar o comando autocorrigido.

Cores no Git

O Git suporta totalmente a saída colorida do terminal, o que ajuda muito na análise visual da saída do comando de forma rápida e fácil. Várias opções podem ajudá-lo a definir a coloração de acordo com sua preferência.

color.ui

O Git colore automaticamente a maior parte de sua saída, mas há uma chave principal se você não gostar desse comportamento. Para desativar todas as saídas de terminal coloridas do Git, faça o seguinte:

```
$ git config --global color.ui false
```

Machine Translated by Google

A configuração padrão é auto, que colore a saída quando vai direto para um terminal, mas omite os códigos de controle de cores quando a saída é redirecionada para um canal ou arquivo.

Você também pode configurá-lo para sempre ignorar a diferença entre terminais e tubos. Você raramente vai querer isso; na maioria dos cenários, se você quiser códigos de cores em sua saída redirecionada, você pode passar um sinalizador --color para o comando Git para forçá-lo a usar códigos de cores. A configuração padrão é quase sempre o que você deseja.

cor.*

Se você quiser ser mais específico sobre quais comandos são coloridos e como, o Git fornece configurações de cores específicas para verbos. Cada um deles pode ser definido como verdadeiro, falso ou sempre:

```
color.branch
color.diff
color.interactive
cor.status
```

Além disso, cada um deles possui subconfigurações que você pode usar para definir cores específicas para partes da saída, se desejar substituir cada cor. Por exemplo, para definir as meta informações em sua saída diff para primeiro plano azul, fundo preto e texto em negrito, você pode executar:

```
$ git config --global color.diff.meta "blue black bold"
```

Você pode definir a cor para qualquer um dos seguintes valores: normal, preto, vermelho, verde, amarelo, azul, magenta , ciano ou branco. Se você deseja um atributo como negrito no exemplo anterior, pode escolher entre negrito, dim, ul (sublinhado), piscar e reverso (trocar o primeiro plano e o plano de fundo).

Ferramentas Externas de Mesclagem e Diferença

Embora o Git tenha uma implementação interna do diff, que é o que mostramos neste livro, você pode configurar uma ferramenta externa. Você também pode configurar uma ferramenta gráfica de resolução de conflitos de mesclagem em vez de ter que resolver os conflitos manualmente. Demonstraremos a configuração da Perforce Visual Merge Tool (P4Merge) para fazer suas diferenças e mesclar resoluções, porque é uma boa ferramenta gráfica e é gratuita.

Se você quiser experimentar, o P4Merge funciona em todas as principais plataformas, então você deve ser capaz de fazê-lo. Usaremos nomes de caminho nos exemplos que funcionam em sistemas macOS e Linux; para Windows, você terá que alterar /usr/local/bin para um caminho executável em seu ambiente.

Para começar, baixe o P4Merge do Perforce. Em seguida, você configurará scripts wrapper externos para executar seus comandos. Usaremos o caminho do macOS para o executável; em outros sistemas, será onde seu binário p4merge está instalado. Configure um script wrapper de mesclagem chamado extMerge que chama seu binário com todos os argumentos fornecidos:

```
$ cat /usr/local/bin/extMerge #!/
bin/sh
```

Machine Translated by Google

```
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

O wrapper diff verifica se sete argumentos são fornecidos e passa dois deles para seu script de mesclagem. Por padrão, o Git passa os seguintes argumentos para o programa diff:

```
caminho arquivo antigo hex antigo modo novo arquivo novo hex novo modo
```

Como você deseja apenas os argumentos do arquivo antigo e do arquivo novo , use o script wrapper para transmitir aqueles de que precisa.

```
$ cat /usr/local/bin/extDiff #!/
bin/sh [ $# -eq 7 ] && /usr/local/
bin/extMerge "$2" "$5"
```

Você também precisa garantir que essas ferramentas sejam executáveis:

```
$ sudo chmod +x /usr/local/bin/extMerge $
sudo chmod +x /usr/local/bin/extDiff
```

Agora você pode configurar seu arquivo de configuração para usar sua resolução de mesclagem personalizada e ferramentas de comparação. Isso requer várias configurações personalizadas: merge.tool para informar ao Git qual estratégia usar, mergetool.<tool>.cmd para especificar como executar o comando, mergetool.<tool>.trustExitCode para informar ao Git se o código de saída desse programa indica uma resolução de mesclagem bem-sucedida ou não, e diff.external para dizer ao Git qual comando executar para diffs. Portanto, você pode executar quatro comandos de configuração:

```
$ 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
```

ou você pode editar seu arquivo ~/.gitconfig para adicionar estas linhas:

```
[merge]
tool = extMerge
[mergetool "extMerge"]
cmd = extMerge "$BASE" "$LOCAL" "$REMOTE"
"$MERGED" trustExitCode = false [diff] external = extDiff
```

Depois de tudo isso definido, se você executar comandos diff como este:

Machine Translated by Google

```
$ git diff 32d1776b1^ 32d1776b1
```

Em vez de obter a saída diff na linha de comando, o Git aciona o P4Merge, que se parece com isto:

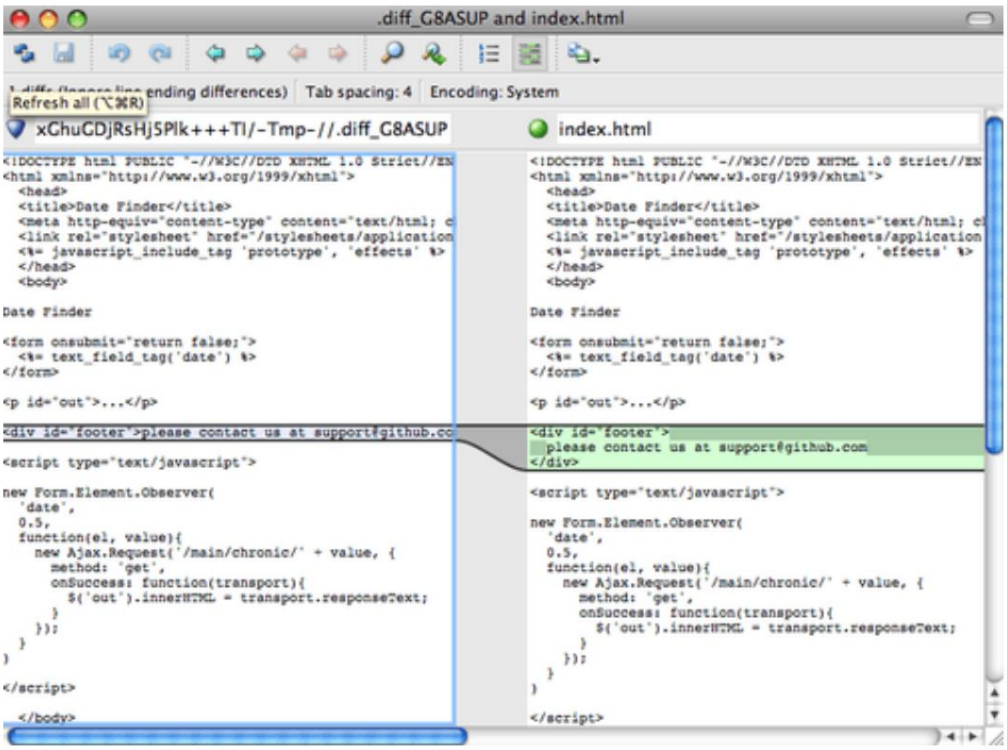


Figura 142. P4Merge

Se você tentar mesclar duas ramificações e subsequentemente tiver conflitos de mesclagem, poderá executar o comando git mergetool; ele inicia o P4Merge para permitir que você resolva os conflitos por meio dessa ferramenta GUI.

O bom dessa configuração de wrapper é que você pode alterar facilmente suas ferramentas de comparação e mesclagem. Por exemplo, para alterar suas ferramentas extDiff e extMerge para executar a ferramenta KDiff3, tudo o que você precisa fazer é editar seu arquivo extMerge :

```
$ cat /usr/local/bin/extMerge #!/bin/sh /Applications/
kdiff3.app/Contents/MacOS/kdiff3 $*
```

Agora, o Git usará a ferramenta KDiff3 para visualização de diferenças e resolução de conflitos de mesclagem.

O Git vem predefinido para usar várias outras ferramentas de resolução de mesclagem sem que você precise definir a configuração do cmd. Para ver uma lista das ferramentas suportadas, tente isto:

```
$ git mergetool --tool-help 'git mergetool --
 tool=<ferramenta>' pode ser definido como um dos seguintes:
                emerge
                gvimdiff
                gvimdiff2 opendiff
```

Machine Translated by Google

```
                    p4merge vimdiff

                    vimdiff2

As seguintes ferramentas são válidas, mas não estão disponíveis no momento: araxis

                    bc3

                    codecompare deltawalker

                    difundir difundir

                    ecmerge kdiff3

                    fundir tkdiff

                    tortoisemerge

                    xxdiff

Algumas das ferramentas listadas acima funcionam apenas em um ambiente com janelas. Se forem executados em uma sessão somente de terminal,

eles falharão.
```

Se você não estiver interessado em usar o KDiff3 para diff, mas quiser usá-lo apenas para resolução de mesclagem, e o comando kdiff3 estiver em seu caminho, você poderá executar:

```
$ git config --global merge.tool kdiff3
```

Se você executar isso em vez de configurar os arquivos extMerge e extDiff , o Git usará o KDiff3 para resolução de mesclagem e a ferramenta diff normal do Git para diffs.

Formatação e espaço em branco

Problemas de formatação e espaço em branco são alguns dos problemas mais frustrantes e sutis que muitos desenvolvedores encontram ao colaborar, especialmente entre plataformas. É muito fácil para patches ou outro trabalho colaborativo introduzir alterações sutis de espaço em branco porque os editores as introduzem silenciosamente e, se seus arquivos tocarem em um sistema Windows, suas terminações de linha poderão ser substituídas. O Git tem algumas opções de configuração para ajudar com esses problemas.

core.autocrlf

Se você estiver programando no Windows e trabalhando com pessoas que não estão (ou vice-versa), provavelmente encontrará problemas de finalização de linha em algum momento. Isso ocorre porque o Windows usa um caractere de retorno de carro e um caractere de avanço de linha para novas linhas em seus arquivos, enquanto os sistemas macOS e Linux usam apenas o caractere de avanço de linha. Este é um fato sutil, mas incrivelmente irritante, do trabalho entre plataformas; muitos editores no Windows substituem silenciosamente as terminações de linha existentes no estilo LF por CRLF ou inserem os dois caracteres de finalização de linha quando o usuário pressiona a tecla enter.

O Git pode lidar com isso convertendo automaticamente as terminações de linha CRLF em LF quando você adiciona um arquivo ao índice e vice-versa quando verifica o código em seu sistema de arquivos. Você pode ativar essa funcionalidade com a configuração core.autocrlf . Se você estiver em uma máquina Windows, defina-o como verdadeiro - isso converte as terminações LF

Machine Translated by Google

em CRLF quando você verificar o código:

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

Se você estiver em um sistema Linux ou macOS que usa terminações de linha LF, não deseja que o Git as converta automaticamente ao retirar os arquivos; no entanto, se um arquivo com terminações CRLF for introduzido acidentalmente, talvez você queira que o Git o corrija. Você pode dizer ao Git para converter CRLF para LF no commit, mas não o contrário, definindo core.autocrlf como input:

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

Essa configuração deve deixá-lo com terminações CRLF em checkouts do Windows, mas terminações LF em sistemas macOS e Linux e no repositório.

Se você for um programador do Windows fazendo um projeto somente do Windows, poderá desativar essa funcionalidade, registrando os retornos de carro no repositório definindo o valor de configuração como false:

```
$ git config --global core.autocrlf false
```

core.whitespace

O Git vem predefinido para detectar e corrigir alguns problemas de espaço em branco. Ele pode procurar por seis problemas primários de espaço em branco - três são ativados por padrão e podem ser desativados e três são desativados por padrão, mas podem ser ativados.

Os três ativados por padrão são blank-at-eol, que procura por espaços no final de uma linha; blank-at-eof, que detecta linhas em branco no final de um arquivo; e space-before-tab, que procura espaços antes de tabulações no início de uma linha.

Os três que estão desativados por padrão, mas podem ser ativados, são indent-with-non-tab, que procura por linhas que começam com espaços em vez de tabulações (e é controlado pela opção tabwidth); indentação de tabulação, que verifica as tabulações na parte de indentação de uma linha; e cr-at-eol, que diz ao Git que os retornos de carro no final das linhas estão OK.

Você pode dizer ao Git quais deles deseja habilitar definindo core.whitespace com os valores que deseja ativar ou desativar, separados por vírgulas. Você pode desabilitar uma opção acrescentando um - antes de seu nome ou usar o valor padrão deixando-o totalmente fora da string de configuração. Por exemplo, se você deseja que todos , exceto o espaço antes da tabulação, sejam definidos, você pode fazer isso (com o espaço à direita sendo uma abreviação para cobrir tanto o espaço em branco quanto o espaço em branco):

```
$ git config --global core.whitespace \
trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Ou você pode especificar apenas a parte de personalização:

Machine Translated by Google

```
$ git config --global core.whitespace \
-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

O Git detectará esses problemas quando você executar um comando git diff e tentar colori-los para que você possa corrigi-los antes de confirmar. Ele também usará esses valores para ajudá-lo ao aplicar patches com git apply. Ao aplicar patches, você pode pedir ao Git para avisá-lo se estiver aplicando patches com os problemas de espaço em branco especificados:

```
$ git apply --whitespace=warn <patch>
```

Ou você pode fazer com que o Git tente corrigir automaticamente o problema antes de aplicar o patch:

```
$ git apply --whitespace=fix <patch>
```

Essas opções também se aplicam ao comando git rebase . Se você cometeu problemas de espaço em branco, mas ainda não enviou upstream, pode executar git rebase --whitespace=fix para que o Git corrija automaticamente os problemas de espaço em branco enquanto reescreve os patches.

Configuração do servidor

Não há tantas opções de configuração disponíveis para o lado do servidor do Git, mas há algumas interessantes que você pode querer anotar.

receber.fsckObjects

O Git é capaz de garantir que cada objeto recebido durante um push ainda corresponda à sua soma de verificação SHA-1 e aponte para objetos válidos. No entanto, ele não faz isso por padrão; é uma operação bastante cara e pode desacelerar a operação, especialmente em grandes repositórios ou pushes. Se você deseja que o Git verifique a consistência do objeto a cada push, pode forçá-lo a fazer isso definindo receive.fsckObjects como

verdadeiro:

```
$ git config --system receive.fsckObjects true
```

Agora, o Git verificará a integridade do seu repositório antes que cada push seja aceito para garantir que clientes defeituosos (ou mal-intencionados) não estejam introduzindo dados corrompidos.

receive.denyNonFastForwards

Se você rebase commits que você já enviou e tente enviar novamente, ou tente enviar um commit para um branch remoto que não contém o commit para o qual o branch remoto aponta atualmente, você será negado. Esta é geralmente uma boa política; mas no caso do rebase, você pode determinar que sabe o que está fazendo e pode forçar a atualização do branch remoto com um sinalizador -f para o seu comando push.

Para dizer ao Git para recusar pushes forçados, configure receive.denyNonFastForwards:

Machine Translated by Google

```
$ git config --system receive.denyNonFastForwards true
```

A outra maneira de fazer isso é por meio de ganchos de recebimento do lado do servidor, que abordaremos daqui a pouco. Essa abordagem permite que você faça coisas mais complexas, como negar avanços não rápidos para um determinado subconjunto de usuários.

receber.negarExcluir

Uma das soluções alternativas para a política denyNonFastForwards é que o usuário exclua a ramificação e, em seguida, envie-a de volta com a nova referência. Para evitar isso, defina receive.denyDeletes como true:

```
$ git config --system receive.denyDeletes true
```

Isso nega qualquer exclusão de branches ou tags — nenhum usuário pode fazer isso. Para remover ramificações remotas, você deve remover os arquivos ref do servidor manualmente. Também há maneiras mais interessantes de fazer isso por usuário por meio de ACLs, como você aprenderá em Um exemplo de política aplicada pelo Git.

Atributos Git

Algumas dessas configurações também podem ser especificadas para um caminho, para que o Git aplique essas configurações apenas para um subdiretório ou subconjunto de arquivos. Essas configurações específicas de caminho são chamadas de atributos do Git e são definidas em um arquivo .gitattributes em um de seus diretórios (normalmente a raiz do seu projeto) ou no arquivo .git/info/attributes se você não quiser o arquivo de atributos comprometido com seu projeto.

Usando atributos, você pode fazer coisas como especificar estratégias de mesclagem separadas para arquivos ou diretórios individuais em seu projeto, dizer ao Git como diferenciar arquivos que não sejam de texto ou fazer com que o Git filtre o conteúdo antes de fazer check-in ou check-out do Git. Nesta seção, você aprenderá sobre alguns dos atributos que pode definir em seus caminhos em seu projeto Git e verá alguns exemplos de como usar esse recurso na prática.

Arquivos binários

Um truque legal para o qual você pode usar os atributos do Git é dizer ao Git quais arquivos são binários (nos casos em que ele não consiga descobrir) e dar ao Git instruções especiais sobre como lidar com esses arquivos. Por exemplo, alguns arquivos de texto podem ser gerados por máquina e não diferenciáveis, enquanto alguns arquivos binários podem ser diferenciados. Você verá como dizer ao Git qual é qual.

Identificando Arquivos Binários

Alguns arquivos parecem arquivos de texto, mas para todos os efeitos devem ser tratados como dados binários. Por exemplo, os projetos Xcode no macOS contêm um arquivo que termina em .pbxproj, que é basicamente um conjunto de dados JSON (formato de dados JavaScript de texto simples) gravado em disco pelo IDE, que registra suas configurações de compilação e assim por diante. Embora seja tecnicamente um arquivo de texto (porque é todo UTF-8), você não quer tratá-lo como tal porque é realmente um banco de dados leve - você não pode mesclar o conteúdo se duas pessoas o alterarem e as diferenças geralmente não são t útil. O arquivo deve ser consumido por uma máquina. Em essência, você deseja tratá-lo como um arquivo binário.

Para dizer ao Git para tratar todos os arquivos pbxproj como dados binários, adicione a seguinte linha ao seu arquivo .gitattributes :

Machine Translated by Google

```
*.pbxproj binário
```

Agora, o Git não tentará converter ou corrigir problemas de CRLF; nem tentará calcular ou imprimir um diff para alterações neste arquivo quando você executar git show ou git diff em seu projeto.

Diferença de Arquivos Binários

Você também pode usar a funcionalidade de atributos do Git para diferenciar efetivamente os arquivos binários. Você faz isso dizendo ao Git como converter seus dados binários em um formato de texto que pode ser comparado por meio do diff normal.

Primeiro, você usará essa técnica para resolver um dos problemas mais irritantes conhecidos pela humanidade: controle de versão de documentos do Microsoft Word. Se você deseja controlar a versão de documentos do Word, pode colocá-los em um repositório Git e fazer commit de vez em quando; mas que bem isso faz? Se você executar o git diff normalmente, verá apenas algo assim:

```
$ git diff diff --
git a/chapter1.docx b/chapter1.docx index 88839c4..4afcb7c
100644 Arquivos binários a/chapter1.docx e b/
chapter1.docx diferem
```

Você não pode comparar duas versões diretamente, a menos que as verifique e verifique manualmente, certo?
Acontece que você pode fazer isso muito bem usando os atributos do Git. Coloque a seguinte linha em seu arquivo .gitattributes :

```
*.docx diff=palavra
```

Isso informa ao Git que qualquer arquivo que corresponda a esse padrão (.docx) deve usar o filtro “palavra” quando você tentar visualizar um diff que contenha alterações. O que é o filtro “palavra”? Você tem que configurá-lo. Aqui você configurará o Git para usar o programa docx2txt para converter documentos do Word em arquivos de texto legíveis, que serão então diferenciados corretamente.

Primeiro, você precisará instalar o docx2txt; você pode baixá-lo em https://sourceforge.net/projects/docx2txt . Siga as instruções no arquivo INSTALL para colocá-lo em algum lugar onde seu shell possa encontrá-lo. Em seguida, você escreverá um script wrapper para converter a saída para o formato que o Git espera. Crie um arquivo em algum lugar no seu caminho chamado docx2txt e adicione este conteúdo:

```
#!/bin/bash
docx2txt.pl "$1" -
```

Não se esqueça de chmod a+x nesse arquivo. Finalmente, você pode configurar o Git para usar este script:

```
$ git config diff.word.textconv docx2txt
```

Agora o Git sabe que se tentar fazer uma comparação entre dois instantâneos e qualquer um dos arquivos terminar em .docx ,

Machine Translated by Google

deve executar esses arquivos através do filtro “word”, que é definido como o programa docx2txt . Isso efetivamente cria boas versões baseadas em texto de seus arquivos do Word antes de tentar diferenciá-los.

Aqui está um exemplo: o capítulo 1 deste livro foi convertido para o formato Word e submetido a um repositório Git. Em seguida, um novo parágrafo foi adicionado. Aqui está o que git diff mostra:

```
$ git diff diff
--git a/chapter1.docx b/chapter1.docx index
0b013ca..ba25db5 100644
--- a/chapter1.docx +++
b/chapter1.docx @@
-2,6 +2,7 @@ Este
capítulo será sobre como começar a usar o Git. Começaremos explicando algumas informações básicas
sobre ferramentas de controle de versão, depois passaremos para como fazer o Git rodar em seu sistema e,
finalmente, como configurá-lo para começar a trabalhar. No final deste capítulo, você deve entender por que
o Git existe, por que você deve usá-lo e deve estar preparado para isso.

1.1. Sobre o controle de versão O
que é "controle de versão" e por que você deveria se importar? O controle de versão é um sistema que registra
alterações em um arquivo ou conjunto de arquivos ao longo do tempo para que você possa recuperar versões
específicas posteriormente. Para os exemplos deste livro, você usará o código-fonte do software como os arquivos
cuja versão está sendo controlada, embora, na realidade, você possa fazer isso com praticamente qualquer tipo
de arquivo em um computador.
+Teste: 1, 2, 3.
Se você é um designer gráfico ou web e deseja manter todas as versões de uma imagem ou layout (o que
você certamente deseja), um Sistema de Controle de Versão (VCS) é uma coisa muito sábia para usar. Ele
permite que você reverta arquivos para um estado anterior, reverta todo o projeto para um estado anterior,
compare alterações ao longo do tempo, veja quem modificou pela última vez algo que pode estar causando
um problema, quem introduziu um problema e quando e muito mais. Usar um VCS geralmente também
significa que, se você estragar tudo ou perder arquivos, poderá recuperá-los facilmente. Além disso, você
obtém tudo isso com muito pouca sobrecarga.
1.1.1. Sistemas de controle de versão local O
método de controle de versão preferido por muitas pessoas é copiar arquivos para outro diretório
(talvez um diretório com carimbo de data/hora, se forem espertos). Essa abordagem é muito comum porque é
muito simples, mas também é incrivelmente propensa a erros. É fácil esquecer em qual diretório você está e
acidentalmente gravar no arquivo errado ou copiar arquivos que não pretendia.
```

O Git nos diz sucintamente e com sucesso que adicionamos a string “Testing: 1, 2, 3.”, que está correta. Não é perfeito – as alterações de formatação não apareceriam aqui – mas certamente funciona.

Outro problema interessante que você pode resolver dessa maneira envolve a comparação de arquivos de imagem. Uma maneira de fazer isso é passar os arquivos de imagem por um filtro que extrai suas informações EXIF – metadados que são gravados com a maioria dos formatos de imagem. Se você baixar e instalar o programa exiftool , poderá usá-lo para converter suas imagens em texto sobre os metadados, para que pelo menos o diff mostre uma representação textual de quaisquer alterações ocorridas. Coloque a seguinte linha em seu arquivo .gitattributes :

```
*.png diff=exif
```

Machine Translated by Google

Configure o Git para usar esta ferramenta:

```
$ git config diff.exif.textconv exiftool
```

Se você substituir uma imagem em seu projeto e executar git diff, verá algo assim:

```
diff --git a/image.png b/image.png index
88839c4..4afcb7c 100644 --- a/image.png +++ b/
image.png @@ -1,12 +1,12 @@ ExifTool Version
Número -Tamanho do arquivo
                                        : 7,74
                                        : 70 KB
-Data/hora da modificação do arquivo    : 2009:04:21 07:02:45-07:00 : 94 kB :
+Tamanho do arquivo                     2009:04:21 07:02:43-07:00
+Data/hora da modificação do arquivo
Tipo de arquivo                         : PNG
Tipo MIME                               : imagem/png :
-Largura da imagem                      1058 : 889
-Altura da imagem
+ Largura da imagem                     : 1056
+Altura da imagem                       : 827 :
Profundidade de bits                    8
Tipo de cor                             : RGB com Alfa
```

Você pode ver facilmente que o tamanho do arquivo e as dimensões da imagem foram alterados.

Expansão de palavras-chave

A expansão de palavras-chave no estilo SVN ou CVS é frequentemente solicitada por desenvolvedores acostumados com esses sistemas. O principal problema com isso no Git é que você não pode modificar um arquivo com informações sobre o commit depois de ter feito o commit, porque o Git faz o checksum do arquivo primeiro. No entanto, você pode injetar texto em um arquivo quando ele for retirado e removê-lo novamente antes de adicioná-lo a um commit. Os atributos do Git oferecem duas maneiras de fazer isso.

Primeiro, você pode injetar a soma de verificação SHA-1 de um blob em um campo Id no arquivo automaticamente. Se você definir esse atributo em um arquivo ou conjunto de arquivos, na próxima vez que verificar essa ramificação, o Git substituirá esse campo pelo SHA-1 do blob. É importante notar que não é o SHA-1 do commit, mas do próprio blob. Coloque a seguinte linha em seu arquivo .gitattributes :

```
*identificação .txt
```

Adicione uma referência Id a um arquivo de teste:

```
$ echo '$Id$' > test.txt
```

Machine Translated by Google

Na próxima vez que você verificar este arquivo, o Git injetará o SHA-1 do blob:

```
$ rm test.txt
$ git checkout -- test.txt $
cat test.txt $Id:
42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

No entanto, esse resultado é de uso limitado. Se você usou a substituição de palavra-chave no CVS ou no Subversion, pode incluir um registro de data – o SHA-1 não é muito útil, porque é bastante aleatório e você não pode dizer se um SHA-1 é mais antigo ou mais recente que outro apenas olhando para eles.

Acontece que você pode escrever seus próprios filtros para fazer substituições em arquivos no commit/checkout. Estes são chamados de filtros “limpos” e “manchas”. No arquivo .gitattributes , você pode definir um filtro para determinados caminhos e, em seguida, configurar scripts que processarão os arquivos antes de serem retirados (“smudge”, consulte O filtro “smudge” é executado no check-out) e antes de eles 're staged (“clean”, consulte O filtro “clean” é executado quando os arquivos são testados). Esses filtros podem ser configurados para fazer todos os tipos de coisas dive

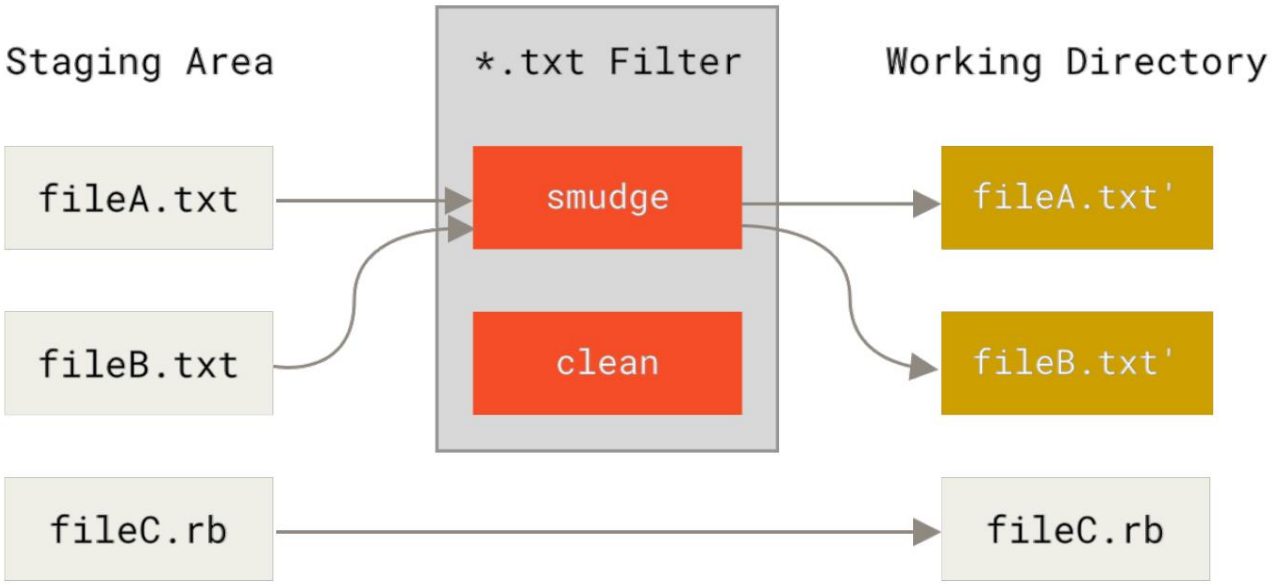


Figura 143. O filtro “smudge” é executado na finalização da compra

Machine Translated by Google

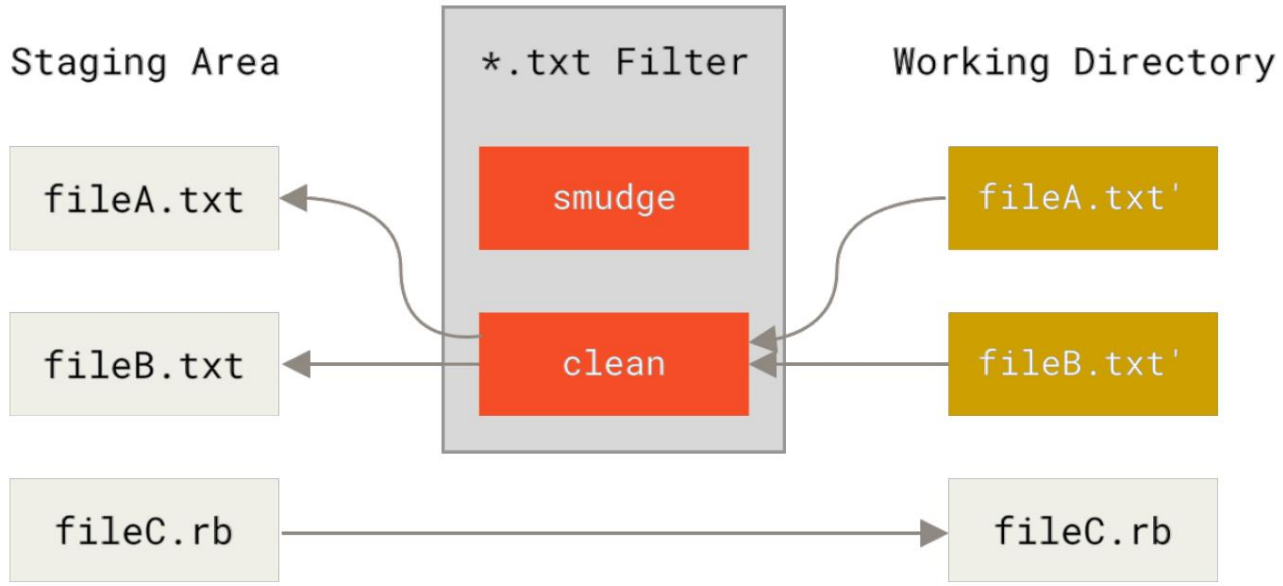


Figura 144. O filtro "limpo" é executado quando os arquivos são testados

A mensagem de confirmação original para esse recurso fornece um exemplo simples de execução de todo o seu código-fonte C por meio do programa de indentação antes da confirmação. Você pode configurá-lo definindo o atributo de filtro em seu arquivo .gitattributes para filtrar arquivos *.c com o filtro "indent":

```
*.c filtro=recuo
```

Em seguida, diga ao Git o que o filtro "indent" faz em smudge e clean:

```
$ git config --global filter.indent.clean indent $ git
config --global filter.indent.smudge cat
```

Nesse caso, quando você enviar arquivos que correspondem a *.c, o Git os executará por meio do programa indent antes de organizá-los e, em seguida, executá-los por meio do programa cat antes de devolvê-los ao disco. O programa cat não faz essencialmente nada: ele cospe os mesmos dados que ele recebe. Essa combinação efetivamente filtra todos os arquivos de código-fonte C por meio de indentação antes de confirmar.

Outro exemplo interessante é a expansão da palavra-chave $Date$, estilo RCS. Para fazer isso corretamente, você precisa de um pequeno script que recebe um nome de arquivo, descobre a última data de confirmação para este projeto e insere a data no arquivo. Aqui está um pequeno script Ruby que faz isso:

```
#! /usr/bin/env dados
ruby = STDIN.read
last_date = `git log --pretty=format:"%ad" -1` puts
data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Tudo o que o script faz é obter a data de confirmação mais recente do comando git log , inseri-la em qualquer string $Date$ que encontrar no stdin e imprimir os resultados - deve ser simples de fazer em qualquer idioma em que você se sinta mais confortável. Você pode nomear este arquivo como expand_date e colocá-lo em seu caminho. Agora, você precisa

Machine Translated by Google

configure um filtro no Git (chame-o de datador) e diga a ele para usar seu filtro expand_date para borrar os arquivos no checkout. Você usará uma expressão Perl para limpar isso no commit:

```
$ git config filter.dater.smudge expand_date $ git
config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\ $/"'
```

Este snippet Perl remove tudo o que vê em uma string $Date$, para voltar para onde você começou. Agora que seu filtro está pronto, você pode testá-lo configurando um atributo Git para esse arquivo que envolve o novo filtro e criando um arquivo com sua palavra-chave $Date$:

```
data*.txt filtro=data
```

```
$ echo '# $Date$' > date_test.txt
```

Se você confirmar essas alterações e verificar o arquivo novamente, verá a palavra-chave substituída corretamente:

```
$ git add date_test.txt .gitattributes $ git
commit -m "Expansão de data de teste no Git" $ rm
date_test.txt $ git checkout date_test.txt $ cat
date_test.txt # $Date: Ter 21 de abril 07:26:52 2009
- 0700$
```

Você pode ver como essa técnica pode ser poderosa para aplicativos personalizados. Você deve ter cuidado, porém, porque o arquivo .gitattributes é confirmado e repassado com o projeto, mas o driver (neste caso, datador) não é, então não funcionará em todos os lugares. Quando você projeta esses filtros, eles devem ser capazes de falhar normalmente e fazer com que o projeto ainda funcione corretamente.

Exportando seu repositório

Os dados de atributos do Git também permitem que você faça algumas coisas interessantes ao exportar um arquivo do seu projeto.

exportar-ignorar

Você pode dizer ao Git para não exportar certos arquivos ou diretórios ao gerar um arquivo. Se houver um subdiretório ou arquivo que você não deseja incluir em seu arquivo compactado, mas que deseja verificar em seu projeto, você pode determinar esses arquivos por meio do atributo export-ignore .

Por exemplo, digamos que você tenha alguns arquivos de teste em um subdiretório test/ e não faça sentido incluí-los na exportação tarball do seu projeto. Você pode adicionar a seguinte linha ao seu arquivo de atributos do Git:

```
testar/exportar-ignorar
```

Machine Translated by Google

Agora, quando você executar git archive para criar um tarball do seu projeto, esse diretório não será incluído no arquivo.

exportação-subst

Ao exportar arquivos para implantação, você pode aplicar a formatação do git log e o processamento de expansão de palavras-chave a partes selecionadas de arquivos marcados com o atributo export-subst .

Por exemplo, se você deseja incluir um arquivo chamado LAST_COMMIT em seu projeto e ter metadados sobre o último commit automaticamente injetado nele quando o arquivo git é executado, você pode, por exemplo, configurar seus arquivos .gitattributes e LAST_COMMIT como este:

```
LAST_COMMIT export-subst
```

```
$ echo 'Data do último commit: $Formato:%cd por %aN$' > LAST_COMMIT $
git add LAST_COMMIT .gitattributes $ git commit -am 'adicionando arquivo
LAST_COMMIT para arquivos'
```

Ao executar git archive, o conteúdo do arquivo arquivado ficará assim:

```
$ arquivo git HEAD | tar xCf ../deployment-testing - $ cat ../deployment-
testing/LAST_COMMIT Última data de confirmação: terça-feira, 21
de abril 08:38:48 2009 -0700 por Scott Chacon
```

As substituições podem incluir, por exemplo, a mensagem de confirmação e quaisquer notas do git, e o git log pode fazer quebra de linha simples:

```
$ echo '$Format:Último commit: %h por %aN em %cd%n%+w(76,6,9)%B$' > LAST_COMMIT $ git
commit -am 'export-subst usa git log'\' formatador personalizado do

O arquivo git usa o processador `pretty=format:` do git log'\''s diretamente
e remove as marcações `$Format:` e `$` ao redor da saída.
'

$ arquivo git @ | tar xfO - LAST_COMMIT
Último commit: 312ccc8 por Jim Hill na sexta-feira, 8 de maio 09:14:04 2015 -0700
        export-subst usa o formatador personalizado do git log

            O arquivo git usa o processador `pretty=format:` do git log diretamente e remove as
            marcações `$Format:` e `$` ao redor da saída.
```

O arquivo resultante é adequado para trabalho de implantação, mas, como qualquer arquivo exportado, não é adequado para trabalho de desenvolvimento posterior.

Machine Translated by Google

Mesclar estratégias

Você também pode usar os atributos do Git para dizer ao Git para usar diferentes estratégias de mesclagem para arquivos específicos em seu projeto. Uma opção muito útil é dizer ao Git para não tentar mesclar arquivos específicos quando eles tiverem conflitos, mas usar o seu lado da mesclagem sobre o de outra pessoa.

Isso é útil se uma ramificação em seu projeto divergiu ou é especializada, mas você deseja mesclar as alterações de volta a partir dela e deseja ignorar determinados arquivos. Digamos que você tenha um arquivo de configurações de banco de dados chamado database.xml que é diferente em duas ramificações e deseja mesclar em sua outra ramificação sem bagunçar o arquivo de banco de dados. Você pode configurar um atributo como este:

```
database.xml merge=nosso
```

E então defina um dummy nossa estratégia de mesclagem com:

```
$ git config --global merge.ours.driver true
```

Se você mesclar na outra ramificação, em vez de ter conflitos de mesclagem com o arquivo database.xml , verá algo assim:

```
$ git merge topic
Mesclagem automática do
database.xml Mesclagem feita por recursivo.
```

Nesse caso, database.xml permanece em qualquer versão que você tinha originalmente.

Git Hooks

Como muitos outros sistemas de controle de versão, o Git tem uma maneira de disparar scripts personalizados quando certas ações importantes ocorrem. Existem dois grupos desses hooks: do lado do cliente e do lado do servidor. Os ganchos do lado do cliente são acionados por operações como confirmação e mesclagem, enquanto os ganchos do lado do servidor são executados em operações de rede, como recebimento de confirmações enviadas. Você pode usar esses ganchos para todos os tipos de razões.

Instalando um Gancho

Os hooks são todos armazenados no subdiretório hooks do diretório Git. Na maioria dos projetos, é .git/hooks. Quando você inicializa um novo repositório com git init, o Git preenche o diretório hooks com vários scripts de exemplo, muitos dos quais são úteis por si só; mas também documentam os valores de entrada de cada script. Todos os exemplos são escritos como scripts de shell, com algum Perl adicionado, mas qualquer script executável com nome adequado funcionará bem - você pode escrevê-los em Ruby ou Python ou em qualquer linguagem com a qual esteja familiarizado. Se você quiser usar os scripts de gancho agrupados, terá que renomeá-los; seus nomes de arquivo terminam com .sample.

Para habilitar um script hook, coloque um arquivo no subdiretório hooks de seu diretório .git que seja nomeado apropriadamente (sem qualquer extensão) e seja executável. A partir daí, deve ser

Machine Translated by Google

chamado. Abordaremos a maioria dos principais nomes de arquivos hook aqui.

Ganchos do lado do cliente

Existem muitos ganchos do lado do cliente. Esta seção os divide em ganchos de fluxo de trabalho de confirmação, scripts de fluxo de trabalho de e-mail e tudo mais.

É importante observar que os ganchos do lado do cliente **não são** copiados quando você clona um repositório. Se sua intenção com esses scripts for impor uma política, você provavelmente desejará fazer isso no lado do servidor; veja o exemplo em Um exemplo de política aplicada pelo Git.

Ganchos de fluxo de trabalho de confirmação

Os primeiros quatro ganchos têm a ver com o processo de confirmação.

O hook pré-commit é executado primeiro, antes mesmo de você digitar uma mensagem de commit. Ele é usado para inspecionar o instantâneo que está prestes a ser confirmado, para ver se você esqueceu algo, para garantir que os testes sejam executados ou para examinar o que você precisa inspecionar no código. Sair diferente de zero desse gancho aborta o commit, embora você possa ignorá-lo com git commit --no-verify. Você pode fazer coisas como verificar o estilo do código (executar lint ou algo equivalente), verificar se há espaços em branco à direita (o gancho padrão faz exatamente isso) ou verificar a documentação apropriada sobre novos métodos.

O hook prepare-commit-msg é executado antes que o editor de mensagens de commit seja acionado, mas depois que a mensagem padrão é criada. Ele permite que você edite a mensagem padrão antes que o autor do commit a veja.
Este gancho recebe alguns parâmetros: o caminho para o arquivo que contém a mensagem de commit até o momento, o tipo de commit e o commit SHA-1 se este for um commit alterado. Esse hook geralmente não é útil para commits normais; em vez disso, é bom para commits em que a mensagem padrão é gerada automaticamente, como mensagens de commit com modelo, commits de mesclagem, commits compactados e commits alterados. Você pode usá-lo em conjunto com um modelo de confirmação para inserir informações programaticamente.

O gancho commit-msg usa um parâmetro, que novamente é o caminho para um arquivo temporário que contém a mensagem de commit escrita pelo desenvolvedor. Se este script for diferente de zero, o Git abortará o processo de commit, então você pode usá-lo para validar o estado do seu projeto ou mensagem de commit antes de permitir que um commit seja feito. Na última seção deste capítulo, demonstraremos como usar esse gancho para verificar se sua mensagem de commit está em conformidade com um padrão exigido.

Após a conclusão de todo o processo de confirmação, o gancho pós-confirmação é executado. Ele não requer nenhum parâmetro, mas você pode obter facilmente o último commit executando git log -1 HEAD. Geralmente, esse script é usado para notificação ou algo semelhante.

Ganchos de fluxo de trabalho de e-mail

Você pode configurar três ganchos do lado do cliente para um fluxo de trabalho baseado em e-mail. Eles são todos invocados pelo comando git am , portanto, se você não estiver usando esse comando em seu fluxo de trabalho, poderá pular com segurança para a próxima seção. Se você estiver recebendo patches por e-mail preparados pelo git format-patch, alguns deles podem ser úteis para você.

O primeiro gancho executado é applypatch-msg. Leva um único argumento: o nome do temporário

Machine Translated by Google

arquivo que contém a mensagem de confirmação proposta. O Git aborta o patch se este script for diferente de zero.
Você pode usar isso para garantir que uma mensagem de confirmação seja formatada corretamente ou para normalizar a mensagem fazendo com que o script a edite no local.

O próximo gancho a ser executado ao aplicar patches via git am é pre-applypatch. Um tanto confuso, ele é executado *depois que* o patch é aplicado, mas antes de um commit ser feito, então você pode usá-lo para inspecionar o instantâneo antes de fazer o commit. Você pode executar testes ou inspecionar a árvore de trabalho com este script. Se algo estiver faltando ou os testes não passarem, sair diferente de zero aborta o script git am sem confirmar o patch.

O último gancho a ser executado durante uma operação git am é post-applypatch, que é executado depois que o commit é feito. Você pode usá-lo para notificar um grupo ou o autor do patch que você baixou.
Você não pode interromper o processo de correção com este script.

Outros ganchos de cliente

O gancho de pré-rebase é executado antes de você rebase qualquer coisa e pode interromper o processo saindo de zero. Você pode usar esse gancho para impedir o rebase de quaisquer commits que já foram enviados. O gancho pré-rebase de exemplo que o Git instala faz isso, embora faça algumas suposições que podem não corresponder ao seu fluxo de trabalho.

O hook pós-reescrita é executado por comandos que substituem commits, como git commit --amend e git rebase (mas não por git filter-branch). Seu único argumento é qual comando acionou a reescrita e recebe uma lista de reescritas no stdin. Este gancho tem muitos dos mesmos usos que os ganchos pós-checkout e pós-fusão .

Depois de executar um git checkout bem-sucedido, o gancho pós-checkout é executado; você pode usá-lo para configurar seu diretório de trabalho adequadamente para seu ambiente de projeto. Isso pode significar mover arquivos binários grandes que você não deseja controlar a origem, documentação de geração automática ou algo nesse sentido.

O gancho pós-fusão é executado após um comando de mesclagem bem-sucedido . Você pode usá-lo para restaurar dados na árvore de trabalho que o Git não pode rastrear, como dados de permissões. Este gancho também pode validar a presença de arquivos externos ao controle Git que você pode querer copiar quando a árvore de trabalho mudar.

O gancho pré-push é executado durante o git push, depois que as referências remotas foram atualizadas, mas antes de qualquer objeto ser transferido. Ele recebe o nome e a localização do controle remoto como parâmetros e uma lista de referências a serem atualizadas por meio do stdin. Você pode usá-lo para validar um conjunto de atualizações de referência antes que ocorra um push (um código de saída diferente de zero abortará o push).

O Git ocasionalmente faz coleta de lixo como parte de sua operação normal, invocando git gc --auto.
O gancho pre-auto-gc é invocado logo antes da coleta de lixo ocorrer e pode ser usado para notificá-lo de que isso está acontecendo ou para abortar a coleta se agora não for um bom momento.

Ganchos do lado do servidor

Além dos ganchos do lado do cliente, você pode usar alguns ganchos importantes do lado do servidor como administrador do sistema para impor praticamente qualquer tipo de política para seu projeto. Esses scripts são executados antes e depois dos pushes para o servidor. Os pré-ganchos podem sair diferentes de zero a qualquer momento para rejeitar o push, bem como

Machine Translated by Google

imprimir uma mensagem de erro de volta para o cliente; você pode configurar uma política de push tão complexa quanto desejar.

pré-receber

O primeiro script a ser executado ao lidar com um push de um cliente é o pré-recebimento. Leva uma lista de referências que estão sendo enviadas do stdin; se sair diferente de zero, nenhum deles é aceito. Você pode usar esse gancho para fazer coisas como garantir que nenhuma das referências atualizadas não seja um avanço rápido ou para fazer o controle de acesso para todas as referências e arquivos que estão modificando com o push.

atualizar

O script de atualização é muito semelhante ao script de pré-recebimento , exceto que é executado uma vez para cada ramificação que o pusher está tentando atualizar. Se o pusher estiver tentando enviar para várias ramificações, o pré-recebimento é executado apenas uma vez, enquanto a atualização é executada uma vez por ramificação para a qual ele está enviando. Em vez de ler de stdin, esse script recebe três argumentos: o nome da referência (ramificação), o SHA-1 para o qual a referência apontou antes do push e o SHA-1 que o usuário está tentando enviar. Se o script de atualização sair diferente de zero, apenas essa referência será rejeitada; outras referências ainda podem ser atualizadas.

pós-recebimento

O gancho pós-recebimento é executado após a conclusão de todo o processo e pode ser usado para atualizar outros serviços ou notificar usuários. Leva os mesmos dados stdin do gancho de pré-recebimento . Os exemplos incluem enviar uma lista por e-mail, notificar um servidor de integração contínua ou atualizar um sistema de rastreamento de tickets – você pode até analisar as mensagens de confirmação para ver se algum ticket precisa ser aberto, modificado ou fechado.
Este script não pode interromper o processo de envio, mas o cliente não desconecta até que seja concluído, portanto, tenha cuidado se tentar fazer algo que possa demorar muito.

ÿ Se estiver escrevendo um roteiro/gancho que outras pessoas precisarão ler, prefira os longos versões de sinalizadores de linha de comando, daqui a seis meses você vai nos agradecer.

Um exemplo de política aplicada pelo Git

Nesta seção, você usará o que aprendeu para estabelecer um fluxo de trabalho do Git que verifica um formato de mensagem de confirmação personalizado e permite que apenas determinados usuários modifiquem determinados subdiretórios em um projeto.
Você criará scripts de cliente que ajudam o desenvolvedor a saber se o envio será rejeitado e scripts de servidor que realmente impõem as políticas.

Os scripts que mostraremos são escritos em Ruby; em parte por causa de nossa inércia intelectual, mas também porque Ruby é fácil de ler, mesmo que você não possa necessariamente escrevê-lo. No entanto, qualquer linguagem funcionará – todos os exemplos de scripts de gancho distribuídos com o Git estão em Perl ou Bash, portanto, você também pode ver muitos exemplos de ganchos nesses idiomas observando os exemplos.

Gancho do lado do servidor

Todo o trabalho do lado do servidor irá para o arquivo de atualização em seu diretório de hooks . O gancho de atualização é executado uma vez por ramificação sendo enviada e recebe três argumentos:

- O nome da referência que está sendo enviada para

Machine Translated by Google

- A revisão antiga onde essa ramificação estava
- A nova revisão sendo enviada

Você também tem acesso ao usuário que está fazendo o push se o push estiver sendo executado por SSH. Se você permitiu que todos se conectassem com um único usuário (como “git”) por meio de autenticação de chave pública, talvez seja necessário fornecer a esse usuário um shell wrapper que determina qual usuário está se conectando com base na chave pública e definir um ambiente variável de acordo. Aqui, assumiremos que o usuário conectado está na variável de ambiente $USER , portanto, seu script de atualização começa reunindo todas as informações necessárias:

```
#!/usr/bin/env rubi

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user       = ENV['USUÁRIO']

puts "Aplicando Políticas..." puts
"(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Sim, essas são variáveis globais. Não julgue – é mais fácil demonstrar desta forma.

Impondo um formato específico de mensagem de confirmação

Seu primeiro desafio é garantir que cada mensagem de confirmação siga um formato específico. Apenas para ter um alvo, suponha que cada mensagem tenha que incluir uma string que se pareça com “ref: 1234” porque você deseja que cada confirmação seja vinculada a um item de trabalho em seu sistema de tickets. Você deve observar cada commit sendo enviado, ver se essa string está na mensagem de commit e, se a string estiver ausente de qualquer um dos commits, saia diferente de zero para que o push seja rejeitado.

Você pode obter uma lista dos valores SHA-1 de todos os commits que estão sendo enviados, pegando os valores $newrev e $oldrev e passando-os para um comando de encanamento do Git chamado git rev-list. Este é basicamente o comando git log , mas por padrão ele imprime apenas os valores SHA-1 e nenhuma outra informação. Assim, para obter uma lista de todos os commits SHA-1 introduzidos entre um commit SHA-1 e outro, você pode executar algo como isto:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Você pode pegar essa saída, percorrer cada um desses commits SHA-1, pegar a mensagem para ela e testar essa mensagem em uma expressão regular que procura um padrão.

Você tem que descobrir como obter a mensagem de commit de cada um desses commits para testar. Para obter o

Machine Translated by Google

dados brutos de confirmação, você pode usar outro comando de encanamento chamado git cat-file. Examinaremos todos esses comandos de encanamento em detalhes no Git Internals; mas, por enquanto, aqui está o que esse comando lhe dá:

```
$ git cat-file commit ca82a6 tree
cfda3bf379e4f8dba8717dee55aab78aef7f4daf parent
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 autor Scott Chacon
<schacon@gmail.com> 1205815931 -0700 committer Scott Chacon <schacon@gmail.com>
1205815931 -0700 committer Scott Chacon <schacon@gmail.com> 1205815931 -0700 committer Scott Chacon <schacon@gmail.com>

Alterar o número da versão
```

Uma maneira simples de obter a mensagem de commit de um commit quando você tem o valor SHA-1 é ir para a primeira linha em branco e pegar tudo depois disso. Você pode fazer isso com o comando sed em sistemas Unix:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
Alterar o número da versão
```

Você pode usar esse encantamento para obter a mensagem de confirmação de cada confirmação que está tentando ser enviada e sair se vir algo que não corresponda. Para sair do script e rejeitar o push, saia diferente de zero. Todo o método se parece com isso:

```
$regex = /\[ref: (\d+)\]/

# formato de mensagem de commit personalizado
aplicado def check_message_format missing_revs
= `git rev-list #{$oldrev}..#{$newrev}`.split("\n") missing_revs.each do |rev| message =
`git cat-file commiter #{rev} | sed '1,/^$/d'` if !$regex.match(message) puts "[POLÍTICA]
Sua mensagem não está formatada corretamente" exit 1 end end




fim
check_message_format
```

Colocar isso em seu script de atualização rejeitará atualizações que contenham commits com mensagens que não aderem à sua regra.

Aplicação de um sistema ACL baseado em usuário

Suponha que você queira adicionar um mecanismo que usa uma lista de controle de acesso (ACL) que especifica quais usuários têm permissão para enviar alterações para quais partes de seus projetos. Algumas pessoas têm acesso total e outras podem enviar alterações apenas para determinados subdiretórios ou arquivos específicos. Para impor isso, você escreverá essas regras em um arquivo chamado acl que reside em seu repositório Git vazio no servidor. Você terá

Machine Translated by Google

o gancho de atualização examina essas regras, vê quais arquivos estão sendo introduzidos para todos os commits sendo enviados e determina se o usuário que está fazendo o envio tem acesso para atualizar todos esses arquivos.

A primeira coisa que você fará é escrever sua ACL. Aqui você usará um formato muito parecido com o mecanismo CVS ACL: ele usa uma série de linhas, onde o primeiro campo é disponível ou não disponível, o próximo campo é uma lista delimitada por vírgulas dos usuários aos quais a regra se aplica e o último campo é o caminho ao qual a regra se aplica (em branco significa acesso aberto). Todos esses campos são delimitados por uma barra vertical (|) .

Nesse caso, você tem alguns administradores, alguns escritores de documentação com acesso ao diretório doc e um desenvolvedor que só tem acesso aos diretórios lib e tests , e seu arquivo ACL se parece com isto:

```
avail|nickh,pjhyett,defunkt,tpw avail|
usinclair,cdickens,ebronte|doc avail|schacon|lib
avail|schacon|testes
```

Você começa lendo esses dados em uma estrutura que pode usar. Nesse caso, para simplificar o exemplo, você aplicará apenas as diretivas de disponibilidade . Aqui está um método que fornece um array associativo onde a chave é o nome do usuário e o valor é um array de caminhos para os quais o usuário escreveu
acesso:

```
def get_acl_access_data(acl_file) # lê dados
ACL acl_file = File.read(acl_file).split("\n").reject
{ |line| linha == acesso = {} acl_file.each do |linha| avail, users, path = line.split('|') próximo a "  }
menos que avail == 'avail' users.split(',').each do |user| acesso[usuário] ||= [] acesso[usuário]
<< caminho fim fim





acesso
fim
```

No arquivo ACL que você examinou anteriormente, esse método get_acl_access_data retorna uma estrutura de dados semelhante a esta:

```
{"defunkt"=>[nil],
"tpw"=>[nil], "nickh"=>[nil],
"pjhyett"=>[nil],
"schacon"=>["lib",
"testes" ], "cdickens"=>["doc"],
```

```
"usinclair"=>["doc"], "ebronte"=>["doc"]}
```

Agora que você definiu as permissões, precisa determinar quais caminhos os commits que estão sendo enviados modificaram, para garantir que o usuário que está enviando tenha acesso a todos eles.

Você pode facilmente ver quais arquivos foram modificados em um único commit com a opção --name-only para o comando git log (mencionado brevemente no Git Basics):

```
$ git log -1 --name-only --pretty=formato:'' 9f585d

LEIA-ME lib/
test.rb
```

Se você usar a estrutura ACL retornada do método get_acl_access_data e compará-la com os arquivos listados em cada um dos commits, poderá determinar se o usuário tem acesso para enviar todos os seus commits:

```
# permite apenas que certos usuários modifiquem certos subdiretórios em um projeto def check_directory_perms access = get_acl_access_data('acl')

# veja se alguém está tentando enviar algo que não pode 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 |caminho| next if
path.size == 0 has_file_access = false access[$user].each do |access_path| if !access_path # o usuário tem acesso a tudo

                || (path.start_with? access_path) # acesso a este caminho
            has_file_access = verdadeiro fim fim

        se !has_file_access
            puts "[POLÍTICA] Você não tem acesso para empurrar para #{path}" saída 1

        fim
fim fim fim

check_directory_perms
```

Você obtém uma lista de novos commits sendo enviados para o seu servidor com git rev-list. Então, para cada um desses commits, você encontra quais arquivos são modificados e garante que o usuário que está fazendo o push tenha acesso a todos os caminhos que estão sendo modificados.

Machine Translated by Google

Agora seus usuários não podem enviar nenhum commit com mensagens mal formadas ou com arquivos modificados fora de seus caminhos designados.

Testando

Se você executar chmod u+x .git/hooks/update, que é o arquivo no qual você deveria ter colocado todo esse código, e tentar enviar um commit com uma mensagem não compatível, obterá algo como isto:

```
$ git push -f origin master Contando
objetos: 5, concluído.
Comprimindo objetos: 100% (3/3), concluído.
Escrevendo objetos: 100% (3/3), 323 bytes, concluído.
Total 3 (delta 1), reutilizado 0 (delta 0)
Desempacotando objetos: 100% (3/3), concluído.
Aplicação de Políticas...
(refs/heads/master) (8338c5) (c5b616)
[POLÍTICA] Sua mensagem não está formatada corretamente
erro: ganchos/atualização encerrados com código de erro 1 erro:
gancho recusou-se a atualizar refs/heads/master To
git@gitserver:project.git ! [remote rejeitado] mestre -> mestre
(hook recusado) erro: falha ao enviar algumas referências para
'git@gitserver:project.git'
```

Há algumas coisas interessantes aqui. Primeiro, você vê onde o gancho começa a correr.

```
Aplicação de políticas...
(refs/heads/master) (fb8c72) (c56860)
```

Lembre-se de que você imprimiu isso no início do seu script de atualização. Qualquer coisa que seu script ecoar para stdout será transferido para o cliente.

A próxima coisa que você notará é a mensagem de erro.

```
[POLÍTICA] Sua mensagem não está formatada corretamente
erro: ganchos/atualização encerrados com código de erro 1 erro:
gancho recusou-se a atualizar refs/heads/master
```

A primeira linha foi impressa por você, as outras duas foram Git informando que o script de atualização saiu diferente de zero e é isso que está recusando seu push. Por último, você tem isso:

```
Para git@gitserver:project.git ! [remote
rejeitado] mestre -> mestre (hook recusado) erro: falha ao enviar algumas
referências para 'git@gitserver:project.git'
```

Você verá uma mensagem de rejeição remota para cada referência que seu gancho recusou e informa que foi recusado especificamente devido a uma falha de gancho.

Machine Translated by Google

Além disso, se alguém tentar editar um arquivo ao qual não tem acesso e enviar um commit que o contenha, verá algo semelhante. Por exemplo, se um autor de documentação tentar enviar um commit modificando algo no diretório lib , ele verá:

```
[POLÍTICA] Você não tem acesso para enviar para lib/test.rb
```

A partir de agora, desde que o script de atualização esteja lá e seja executável, seu repositório nunca terá uma mensagem de confirmação sem o seu padrão e seus usuários serão colocados em sandbox.

Ganchos do lado do cliente

A desvantagem dessa abordagem é a lamentação que inevitavelmente resultará quando os pushes de confirmação de seus usuários forem rejeitados. Ter seu trabalho cuidadosamente elaborado rejeitado no último minuto pode ser extremamente frustrante e confuso; além disso, eles terão que editar seu histórico para corrigi-lo, o que nem sempre é para os fracos de coração.

A resposta para esse dilema é fornecer alguns ganchos do lado do cliente que os usuários possam executar para notificá-los quando estiverem fazendo algo que o servidor provavelmente rejeitará. Dessa forma, eles podem corrigir quaisquer problemas antes de confirmar e antes que esses problemas se tornem mais difíceis de corrigir. Como os hooks não são transferidos com um clone de um projeto, você deve distribuir esses scripts de alguma outra forma e, em seguida, fazer com que seus usuários os copiem para o diretório .git/hooks e torná-los executáveis. Você pode distribuir esses ganchos dentro do projeto ou em um projeto separado, mas o Git não os configurará automaticamente.

Para começar, você deve verificar sua mensagem de commit antes de cada commit ser registrado, para que você saiba que o servidor não rejeitará suas alterações devido a mensagens de commit mal formatadas. Para fazer isso, você pode adicionar o hook commit-msg . Se você ler a mensagem do arquivo passado como o primeiro argumento e compará-lo com o padrão, poderá forçar o Git a abortar o commit se não houver correspondência:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message) puts
"[POLÍTICA] Sua mensagem não está formatada corretamente" exit 1 end
```

Se esse script estiver no lugar (em .git/hooks/commit-msg) e executável, e você confirmar com uma mensagem que não está formatada corretamente, você verá isto:

```
$ git commit -am 'Teste'
[POLÍTICA] Sua mensagem não está formatada corretamente
```

Machine Translated by Google

Nenhuma confirmação foi concluída nessa instância. No entanto, se sua mensagem contiver o padrão adequado, o Git permite que você confirme:

```
$ git commit -am 'Test [ref: 132]' [master
e05c914] Test [ref: 132] 1 arquivo alterado,
1 inserções(+), 0 exclusões(-)
```

Em seguida, você deseja certificar-se de que não está modificando arquivos que estão fora do escopo da ACL. Se o diretório .git do seu projeto contiver uma cópia do arquivo ACL usado anteriormente, o seguinte script de pré-confirmação aplicará essas restrições para você:

```
#!/usr/bin/env rubi

$usuário        = ENV['USUÁRIO']

# [insira o método acl_access_data acima]

# permite apenas que certos usuários modifiquem certos subdiretórios em um projeto 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 |
caminho| next if path.size == 0 has_file_access = false access[$user].each do |access_path| se !
access_path || (path.index(access_path) == 0)



            has_file_access = verdadeiro
fim
if !has_file_access coloca
            "[POLÍTICA] Você não tem acesso para empurrar para #{path}" saída 1

fim
fim fim


check_directory_perms
```

Este é aproximadamente o mesmo script da parte do lado do servidor, mas com duas diferenças importantes. Primeiro, o arquivo ACL está em um local diferente, porque esse script é executado em seu diretório de trabalho, não em seu diretório .git . Você deve alterar o caminho para o arquivo ACL a partir disso:

```
acesso = get_acl_access_data('acl')
```

para isso:

Machine Translated by Google

```
acesso = get_acl_access_data('.git/acl')
```

A outra diferença importante é a maneira como você obtém uma listagem dos arquivos que foram alterados. Como o método do lado do servidor examina o log de confirmações e, neste ponto, a confirmação ainda não foi registrada, você deve obter a listagem de arquivos da área de preparação. Em vez de:

```
files_modified = `git log -1 --name-only --pretty=formato:'' #{ref}`
```

você tem que usar:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Mas essas são as duas únicas diferenças – caso contrário, o script funciona da mesma maneira. Uma ressalva é que ele espera que você esteja executando localmente como o mesmo usuário que você envia para a máquina remota. Se for diferente, você deve definir a variável $user manualmente.

Outra coisa que podemos fazer aqui é garantir que o usuário não envie referências não avançadas. Para obter uma referência que não seja um avanço rápido, você deve fazer o rebase após um commit que você já enviou ou tentar enviar um branch local diferente para o mesmo branch remoto.

Presumivelmente, o servidor já está configurado com receive.denyDeletes e receive.denyNonFastForwards para impor essa política, portanto, a única coisa acidental que você pode tentar capturar é o rebase de commits que já foram enviados.

Aqui está um exemplo de script pré-rebase que verifica isso. Ele obtém uma lista de todos os commits que você está prestes a reescrever e verifica se eles existem em alguma de suas referências remotas. Se ele vir um que pode ser acessado de uma de suas referências remotas, ele abortará o rebase.

```
#!/usr/bin/env rubi

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_push = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}` if
shas_push.split("\n").include?(sha)
        puts "[POLICY] Commit #{sha} já foi enviado para #{remote_ref}" saída 1
```

Machine Translated by Google

```
fim fim

fim
```

Este script usa uma sintaxe que não foi abordada na Seleção de revisão. Você obtém uma lista de commits que já foram enviados executando isto:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

A sintaxe SHA^@ é resolvida para todos os pais desse commit. Você está procurando por qualquer confirmação que possa ser acessada a partir da última confirmação no controle remoto e que não seja acessível a partir de nenhum pai de nenhum dos SHA-1s que você está tentando enviar - o que significa que é um avanço rápido.

A principal desvantagem dessa abordagem é que ela pode ser muito lenta e geralmente desnecessária - se você não tentar forçar o push com -f, o servidor o avisará e não aceitará o push. No entanto, é um exercício interessante e pode, em teoria, ajudá-lo a evitar um rebase que mais tarde você pode ter que voltar e corrigir.

Resumo

Cobrimos a maioria das principais maneiras de personalizar seu cliente e servidor Git para melhor se adequar ao seu fluxo de trabalho e projetos. Você aprendeu sobre todos os tipos de definições de configuração, atributos baseados em arquivo e ganchos de evento e construiu um exemplo de servidor de aplicação de política. Agora você deve ser capaz de fazer com que o Git se encaixe em praticamente qualquer fluxo de trabalho que você possa imaginar.

Machine Translated by Google

Git e outros sistemas

O mundo não é perfeito. Normalmente, você não pode mudar imediatamente todos os projetos com os quais entra em contato para o Git. Às vezes, você está preso em um projeto usando outro VCS e gostaria que fosse o Git. Passaremos a primeira parte deste capítulo aprendendo sobre as maneiras de usar o Git como cliente quando o projeto no qual você está trabalhando estiver hospedado em um sistema diferente.

Em algum momento, você pode querer converter seu projeto existente para o Git. A segunda parte deste capítulo aborda como migrar seu projeto para o Git de vários sistemas específicos, bem como um método que funcionará se não existir nenhuma ferramenta de importação pré-criada.

Git como cliente

O Git fornece uma experiência tão boa para os desenvolvedores que muitas pessoas descobriram como usá-lo em suas estações de trabalho, mesmo que o restante da equipe esteja usando um VCS totalmente diferente. Há vários desses adaptadores, chamados “pontes”, disponíveis. Aqui, abordaremos aqueles que você provavelmente encontrará na natureza.

Git e Subversion

Uma grande fração dos projetos de desenvolvimento de código aberto e um bom número de projetos corporativos usam o Subversion para gerenciar seu código-fonte. Ele existe há mais de uma década e, na maior parte desse tempo, foi a escolha *de fato* do VCS para projetos de código aberto. Também é muito semelhante em muitos aspectos ao CVS, que era o garotão do mundo do controle de origem antes disso.

Um dos grandes recursos do Git é uma ponte bidirecional para o Subversion chamada git svn. Essa ferramenta permite que você use o Git como um cliente válido para um servidor Subversion, para que você possa usar todos os recursos locais do Git e, em seguida, enviar para um servidor Subversion como se estivesse usando o Subversion localmente. Isso significa que você pode fazer ramificações e mesclagens locais, usar a área de preparação, usar rebase e seleção de cereja e assim por diante, enquanto seus colaboradores continuam a trabalhar de maneiras sombrias e antigas. É uma boa maneira de inserir o Git no ambiente corporativo e ajudar seus colegas desenvolvedores a se tornarem mais eficientes enquanto você faz lobby para mudar a infraestrutura para oferecer suporte total ao Git. A ponte Subversion é a porta de entrada para o mundo DVCS.

git svn

O comando base no Git para todos os comandos de ponte do Subversion é git svn. São necessários alguns comandos, portanto, mostraremos os mais comuns enquanto passamos por alguns fluxos de trabalho simples.

É importante observar que, ao usar o git svn, você está interagindo com o Subversion, que é um sistema que funciona de maneira muito diferente do Git. Embora você **possa** fazer ramificações e mesclagens locais, geralmente é melhor manter seu histórico o mais linear possível rebaseando seu trabalho e evitando fazer coisas como interagir simultaneamente com um repositório remoto Git.

Não reescreva seu histórico e tente enviar novamente, e não envie para um repositório Git paralelo para colaborar com outros desenvolvedores Git ao mesmo tempo. O Subversion pode ter apenas um único histórico linear, e confundi-lo é muito fácil. Se você estiver trabalhando com uma equipe e alguns estiverem usando SVN e outros usando Git, certifique-se de que todos estejam usando o servidor SVN para colaborar - isso fará com que

Machine Translated by Google

sua vida mais fácil.

Configurando

Para demonstrar essa funcionalidade, você precisa de um repositório SVN típico ao qual tenha acesso de gravação. Se você quiser copiar esses exemplos, terá que fazer uma cópia gravável de um repositório de teste SVN. Para fazer isso facilmente, você pode usar uma ferramenta chamada svnsync que vem com o Subversion.

Para acompanhar, primeiro você precisa criar um novo repositório Subversion local:

```
$ mkdir /tmp/test-svn $
svnadmin criar /tmp/test-svn
```

Em seguida, habilite todos os usuários a alterar revprops – a maneira mais fácil é adicionar um script pre-revprop-change que sempre saia 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change #!/bin/sh
exit 0; $ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Agora você pode sincronizar este projeto com sua máquina local chamando svnsync init com os repositórios de e para.

```
$ svnsync arquivo init:///tmp/test-svn \ http://your-
svn-server.example.org/svn/
```

Isso configura as propriedades para executar a sincronização. Você pode então clonar o código executando:

```
$ arquivo de sincronização svnsync:///tmp/test-
svn Revisão confirmada 1.
Propriedades copiadas para revisão 1.
Transmitindo dados de arquivo .....[...]
Revisão confirmada 2.
Propriedades copiadas para revisão 2. […]
```

Embora esta operação possa levar apenas alguns minutos, se você tentar copiar o repositório original para outro repositório remoto em vez de um local, o processo levará quase uma hora, mesmo que haja menos de 100 commits. O Subversion tem que clonar uma revisão por vez e depois colocá-la de volta em outro repositório – é ridiculamente ineficiente, mas é a única maneira fácil de fazer isso.

Começando

Agora que você tem um repositório Subversion ao qual tem acesso de gravação, você pode passar por um fluxo de trabalho típico. Você começará com o comando git svn clone , que importa um Subversion inteiro

Machine Translated by Google

repositório em um repositório Git local. Lembre-se que se você estiver importando de um repositório Subversion hospedado real, você deve substituir o file:///tmp/test-svn aqui pela URL do seu repositório Subversion:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags Inicializado
repositório Git vazio em /private/tmp/progit/test-svn/.git/ r1 =
dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/ origem/tronco)
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/origem/tronco)
Possível ponto de ramificação encontrado: file:///tmp/test-svn/trunk => file:///tmp/test
svn/branches/my-calc-branch, 75
Ramo pai encontrado: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Seguindo pai com do_switch
Pai seguido com sucesso r76 =
0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
CABEÇA verificada:
file:///tmp/test-svn/trunk r75
```

Isso executa o equivalente a dois comandos – git svn init seguido por git svn fetch – na URL que você fornece. Isso pode demorar um pouco. Se, por exemplo, o projeto de teste tiver apenas cerca de 75 commits e a base de código não for tão grande, o Git deverá verificar cada versão, uma por vez, e confirmá-la individualmente. Para um projeto com centenas ou milhares de commits, isso pode literalmente levar horas ou até dias para ser concluído.

A parte -T trunk -b branches -t tags diz ao Git que este repositório Subversion segue as convenções básicas de ramificação e marcação. Se você nomear seu tronco, ramos ou identificadores de maneira diferente, poderá alterar essas opções. Como isso é muito comum, você pode substituir toda essa parte por -s, o que significa layout padrão e implica todas essas opções. O seguinte comando é equivalente:

```
$ arquivo git svn clone:///tmp/test-svn -s
```

Neste ponto, você deve ter um repositório Git válido que importou suas ramificações e tags:

```
$ git ramo -a
* mestre
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 remotos/origem/tronco
```

Machine Translated by Google

Observe como esta ferramenta gerencia tags Subversion como refs remotos. Vamos dar uma olhada mais de perto no comando show-ref do Git plumbing:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

O Git não faz isso quando clona de um servidor Git; aqui está a aparência de um repositório com tags após um novo clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

O Git busca as tags diretamente em refs/tags, em vez de tratá-las como branches remotos.

Comprometendo-se de volta ao Subversion

Agora que você tem um diretório de trabalho, pode fazer algum trabalho no projeto e enviar seus commits de volta para o upstream, usando o Git efetivamente como um cliente SVN. Se você editar um dos arquivos e confirmá-lo, terá um commit que existe no Git localmente e não existe no servidor Subversion:

```
$ git commit -am 'Adicionando instruções git-svn ao README' [master
 4af61fd] Adicionando instruções git-svn ao README 1 arquivo
alterado, 5 inserções(+)
```

Em seguida, você precisa enviar sua alteração upstream. Observe como isso muda a maneira como você trabalha com o Subversion – você pode fazer vários commits offline e depois enviá-los todos de uma vez para o servidor Subversion. Para enviar para um servidor Subversion, execute o comando git svn dcommit :

```
$ git svn dcommit
Confirmando para o arquivo:///tmp/test-svn/trunk ...
M README.txt
Confirmado r77 M
README.txt r77 =
95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
Nenhuma alteração entre 4af61fd05045e07598c553167e0f31c84fd6ffe1 e
```

Machine Translated by Google

```
refs/remotes/origem/tronco
Redefinindo para as últimas refs/remotes/origem/tronco
```

Isso pega todos os commits que você fez no código do servidor Subversion, faz um commit do Subversion para cada um e então reescreve seu commit local do Git para incluir um identificador único. Isso é importante porque significa que todas as somas de verificação SHA-1 para seus commits mudam. Em parte por esse motivo, trabalhar com versões remotas baseadas em Git de seus projetos simultaneamente com um servidor Subversion não é uma boa ideia. Se você observar o último commit, poderá ver o novo git-svn-id que foi adicionado:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Autor: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Data: quinta-feira, 24 de julho 03:08:36 2014 +0000

Adicionando instruções git-svn ao README

git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Observe que a soma de verificação SHA-1 que começou originalmente com 4af61fd quando você fez o commit agora começa com 95e0222. Se você deseja enviar para um servidor Git e um servidor Subversion, você deve enviar (dcommit) para o servidor Subversion primeiro, porque essa ação altera seus dados de confirmação.

Puxando Novas Mudanças

Se você estiver trabalhando com outros desenvolvedores, em algum momento um de vocês fará push e o outro tentará fazer push de uma alteração conflitante. Essa alteração será rejeitada até que você mescle no trabalho deles. No git svn, fica assim:

```
$ git svn dcommit
Confirmando para o arquivo:///tmp/test-svn/trunk ...

ERRO do SVN:
A transação está desatualizada: Arquivo '/trunk/README.txt' está desatualizado
W: d5837c4b461b7c0e018b49d12398769d2bfc240a e refs/remotes/origin/trunk diferem, usando
rebase: :100644 100644 f414c433af0fd673442429abadf8d5adacf

c80b6127dd04f5fcda218730ddf3a2da4eb39138 M README.txt O
branch master atual está atualizado.
ERRO: Nem todas as alterações foram confirmadas no SVN, no entanto, as confirmadas
(se houver) parecem ter sido integradas com sucesso à árvore de trabalho.
Consulte as mensagens acima para obter detalhes.
```

Para resolver essa situação, você pode executar git svn rebase, que baixa todas as alterações no servidor que você ainda não possui e rebase qualquer trabalho que você tenha sobre o que está no servidor:

```
$ git svn rebase
```

Machine Translated by Google

```
Comprometendo-se com file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date W:
eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ, using rebase: :100644
100644 65536c6e30d263495c17d781962cfff12422693a b34372b25ccf4945fe5658fa381b075045e7702a M
README.txt First, rebobinando a cabeça para reproduzir seu trabalho em cima dela...



Aplicando: atualizar foo
Usando informações de índice para reconstruir uma árvore base...
M README.txt
Voltando ao patch base e 3-way merge...
ERRO de README.txt de
mesclagem automática: Nem todas as alterações foram confirmadas no SVN, no entanto, as
confirmadas (se houver) parecem ter sido integradas com sucesso à árvore de trabalho.
Consulte as mensagens acima para obter detalhes.
```

Agora, todo o seu trabalho está em cima do que está no servidor Subversion, então você pode dcommit com sucesso:

```
$ git svn dcommit
Confirmando para o arquivo:///tmp/test-svn/trunk ...
M README.txt
Comprometida r85
M README.txt r85 =
9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
Nenhuma alteração entre 5762f56732a958d6cfda681b661d2a239cc53ef5 e refs/
remotes/origin/trunk Redefinindo para as últimas refs/remotes/origin/trunk
```

Observe que, ao contrário do Git, que exige que você mescle o trabalho upstream que ainda não possui localmente antes de poder enviar, git svn faz com que você faça isso apenas se as alterações entrarem em conflito (muito parecido com o funcionamento do Subversion). Se alguém enviar uma alteração para um arquivo e você enviar uma alteração para outro arquivo, seu dcommit funcionará bem:

```
$ git svn dcommit
Confirmando para o arquivo:///tmp/test-svn/trunk ...
M configure.ac Confirmado
r87 M autogen.sh r86 =


d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
M configure.ac r87 =
f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ, using rebase: :100755
100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb65d4a7d0389ed6d M
autogen.sh First, rewinding head to replay your work on top of it...
```

Machine Translated by Google

É importante lembrar disso, porque o resultado é um estado de projeto que não existia em nenhum de seus computadores quando você o pressionou. Se as alterações forem incompatíveis, mas não conflitantes, você poderá ter problemas difíceis de diagnosticar. Isso é diferente de usar um servidor Git – no Git, você pode testar totalmente o estado em seu sistema cliente antes de publicá-lo, enquanto no SVN, você nunca pode ter certeza de que os estados imediatamente antes e depois do commit são idênticos.

Você também deve executar este comando para obter as alterações do servidor Subversion, mesmo que não esteja pronto para se comprometer. Você pode executar git svn fetch para obter os novos dados, mas git svn rebase faz a busca e atualiza seus commits locais.

```
$ git svn rebase
M autogen.sh r88 =
c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
Primeiro, rebobinando a cabeça para reproduzir seu trabalho em cima dela...
Mestre avançado para refs/remotes/origin/trunk.
```

Executar git svn rebase de vez em quando garante que seu código esteja sempre atualizado. Você precisa ter certeza de que seu diretório de trabalho está limpo ao executar isso. Se você tiver alterações locais, você deve ocultar seu trabalho ou confirmá-lo temporariamente antes de executar git svn rebase - caso contrário, o comando será interrompido se perceber que o rebase resultará em um conflito de mesclagem.

Problemas de ramificação do Git

Quando você se sentir confortável com um fluxo de trabalho do Git, provavelmente criará ramificações de tópicos, trabalhará neles e depois os mesclará . em uma única ramificação a cada vez, em vez de mesclar as ramificações. A razão para preferir o rebase é que o Subversion tem um histórico linear e não lida com mesclagens como o Git, então git svn segue apenas o primeiro pai ao converter os instantâneos em submissões do Subversion.

Suponha que seu histórico se pareça com o seguinte: você criou um branch experimental , fez dois commits e depois os fundiu novamente no master. Quando você dcommit, você vê uma saída como esta:

```
$ git svn dcommit
Confirmando para o arquivo:///tmp/test-svn/trunk ...
M CHANGES.txt
Confirmado r89 M
CHANGES.txt r89 =
89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
M COPIANDO.txt
M INSTALL.txt
Confirmado r90 M
INSTALL.txt
M COPYING.txt r90
= cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
Nenhuma alteração entre 71af502c214ba13123992338569f4669877f55fd
e refs/remotes/origin/trunk Redefinindo para as últimas refs/remotes/
origin/trunk
```

Machine Translated by Google

Executar dcommit em uma ramificação com histórico mesclado funciona bem, exceto que, quando você olha para o histórico do projeto Git, ele não reescreveu nenhum dos commits que você fez na ramificação experimental - em vez disso, todas essas alterações aparecem na versão SVN do confirmação de mesclagem única.

Quando outra pessoa clona esse trabalho, tudo o que vê é o merge commit com todo o trabalho compactado nele, como se você executasse git merge --squash; eles não veem os dados de confirmação sobre de onde veio ou quando foi confirmado.

Ramificação do Subversion

Ramificar no Subversion não é o mesmo que ramificar no Git; se você pode evitar usá-lo muito, provavelmente é melhor. No entanto, você pode criar e comprometer ramificações no Subversion usando git svn.

Criando uma nova ramificação SVN

Para criar uma nova ramificação no Subversion, execute git svn branch [new-branch]:

```
$ git svn branch opera
Copiando arquivo:///tmp/test-svn/trunk em r90 para arquivo:///tmp/test-svn/branches/opera...
Possível ponto de ramificação encontrado: file:///tmp/test-svn/trunk => file:///tmp/
test svn/branches/opera, 90 Ramificação pai encontrada: (refs/remotes/origin/
opera) cb522197870e61467473391799148f6721bcf9a0 Seguinte pai com
do_switch Pai seguido com sucesso r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302
(refs/remotes/origin/opera)
```

Isso faz o equivalente ao comando svn copy trunk branches/opera no Subversion e opera no servidor Subversion. É importante observar que ele não verifica sua saída nesse ramo; se você confirmar neste ponto, esse commit irá para o trunk no servidor, não para o opera.

Alternando Ramos Ativos

O Git descobre para qual ramificação seus dcommits vão procurando a ponta de qualquer uma das suas ramificações do Subversion em seu histórico – você deve ter apenas uma, e deve ser a última com um git-svn-id em seu histórico de ramificação atual.

Se você quiser trabalhar em mais de uma ramificação simultaneamente, você pode configurar ramificações locais para dcommit para ramificações específicas do Subversion, iniciando-as na submissão importada do Subversion para aquela ramificação. Se você deseja uma ramificação da ópera na qual possa trabalhar separadamente, execute:

```
$ git branch opera remotes/origin/opera
```

Agora, se você quiser mesclar seu branch opera com trunk (seu branch master), você pode fazer isso com um git merge normal. Mas você precisa fornecer uma mensagem de confirmação descritiva (via -m), ou a mesclagem dirá “Merge branch opera” em vez de algo útil.

Lembre-se de que, embora você esteja usando o git merge para fazer esta operação, a mesclagem provavelmente será

Machine Translated by Google

muito mais fácil do que seria no Subversion (porque o Git detectará automaticamente a base de mesclagem apropriada para você), este não é um commit de mesclagem normal do Git. Você tem que enviar esses dados de volta para um servidor Subversion que não pode lidar com um commit que rastreia mais de um pai; então, depois que você o empurrar para cima, parecerá um único commit que esmagou todo o trabalho de outro branch em um único commit. Depois de mesclar uma ramificação com outra, você não pode facilmente voltar e continuar trabalhando nessa ramificação, como faria normalmente no Git. O comando dcommit que você executa apaga qualquer informação que diga em qual ramo foi mesclado, então os cálculos subsequentes da base de mesclagem estarão errados - o dcommit faz com que o resultado do git merge pareça que você executou git merge --squash.
Infelizmente, não há uma boa maneira de evitar essa situação – o Subversion não pode armazenar essas informações, então você sempre será prejudicado por suas limitações enquanto estiver usando-o como seu servidor. Para evitar problemas, você deve excluir a ramificação local (neste caso, ópera) depois de mesclá-la no tronco.

Comandos do Subversion

O conjunto de ferramentas git svn fornece vários comandos para ajudar a facilitar a transição para o Git, fornecendo algumas funcionalidades semelhantes às que você tinha no Subversion. Aqui estão alguns comandos que mostram o que o Subversion costumava fazer.

História do Estilo SVN

Se você está acostumado com o Subversion e deseja ver seu histórico no estilo de saída SVN, você pode executar git svn log para ver seu histórico de commits na formatação SVN:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sáb, 02 de maio de 2014) | 2 linhas

mudança automática

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sáb, 02 de maio de 2014) | 2 linhas

Mesclar ramificação 'experiência'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sáb, 02 de maio de 2014) | 2 linhas

atualizei o changelog
```

Você deve saber duas coisas importantes sobre git svn log. Primeiro, ele funciona offline, ao contrário do comando real svn log , que solicita os dados ao servidor Subversion. Em segundo lugar, ele mostra apenas os commits que foram enviados ao servidor Subversion. Commits locais do Git que você não fez dcommit não aparecem; nem os commits que as pessoas fizeram no servidor Subversion nesse meio tempo. É mais como o último estado conhecido dos commits no servidor Subversion.

Anotação SVN

Assim como o comando git svn log simula o comando svn log offline, você pode obter o equivalente a svn annotate executando git svn Blase [FILE]. A saída se parece com isto:

Machine Translated by Google

```
$ git svn culpa README.txt 2
Buffers de protocolo temporal - formato de intercâmbio de dados do Google 2
temporal Copyright 2008 Google Inc. 2 temporal http://code.google.com/apis/
protocolbuffers/ 2 temporal 22 temporal Instalação C++ - Unix 22 temporal =
====================== 2 temporal 79 schacon Confirmando no git-svn. schacon




78 2 temporal Para construir e instalar o tempo de execução do buffer de protocolo C++ e o compilador de
buffer temporal do protocolo 2 (protoc), execute o seguinte: 2 temporal
```

Novamente, ele não mostra os commits que você fez localmente no Git ou que foram enviados para o Subversion nesse meio tempo.

Informações do Servidor SVN

Você também pode obter o mesmo tipo de informação que svn info fornece executando git svn info:

```
$ git svn info
Caminho: .
URL: https://schacon-test.googlecode.com/svn/trunk Raiz do
repositório: https://schacon-test.googlecode.com/svn UUID do repositório:
4c93b258-373f-11de-be05-5f7a86268029 Revisão: 87 Tipo de nó :
directory Agenda: normal Última alteração Autor: schacon Última
alteração Rev: 87 Data da última alteração: 2009-05-02 16:07:37 -0700
(Sáb, 02 de maio de 2009)
```

Isso é como culpa e login que roda offline e está atualizado apenas a partir da última vez que você se comunicou com o servidor Subversion.

Ignorando o que o Subversion ignora

Se você clonar um repositório do Subversion que tenha as propriedades svn:ignore definidas em qualquer lugar, provavelmente desejará definir os arquivos .gitignore correspondentes para que não envie acidentalmente arquivos que não deveria. git svn tem dois comandos para ajudar com esse problema. O primeiro é git svn create-ignore, que cria automaticamente os arquivos .gitignore correspondentes para você, para que seu próximo commit possa incluí-los.

O segundo comando é git svn show-ignore, que imprime em stdout as linhas que você precisa colocar em um arquivo .gitignore para que você possa redirecionar a saída para o arquivo de exclusão do seu projeto:

```
$ git svn show-ignore > .git/info/exclude
```

Machine Translated by Google

Dessa forma, você não suja o projeto com arquivos .gitignore . Esta é uma boa opção se você for o único usuário do Git em uma equipe do Subversion e seus colegas de equipe não quiserem arquivos .gitignore no projeto.

Resumo do Git-Svn

As ferramentas git svn são úteis se você estiver preso a um servidor Subversion ou estiver em um ambiente de desenvolvimento que necessite executar um servidor Subversion. Você deve considerá-lo um Git aleijado, no entanto, ou encontrará problemas na tradução que podem confundir você e seus colaboradores. Para evitar problemas, tente seguir estas diretrizes:

- Mantenha um histórico linear do Git que não contenha confirmações de mesclagem feitas pelo git merge. Rebase qualquer trabalho que você faz fora de sua ramificação principal de volta para ela; não o mescle.
- Não configure e colabore em um servidor Git separado. Possivelmente tenha um para acelerar clones para novos desenvolvedores, mas não envie nada para ele que não tenha uma entrada git-svn-id . Você pode até querer adicionar um gancho de pré-recebimento que verifique cada mensagem de commit para um git-svn-id e rejeite pushes que contenham commits sem ele.

Se você seguir essas diretrizes, trabalhar com um servidor Subversion pode ser mais suportável. No entanto, se for possível mudar para um servidor Git real, isso pode ganhar muito mais para sua equipe.

Git e Mercurial

O universo DVCS é maior do que apenas o Git. Na verdade, existem muitos outros sistemas neste espaço, cada um com seu próprio ângulo de como fazer o controle de versão distribuído corretamente. Além do Git, o mais popular é o Mercurial, e os dois são muito semelhantes em muitos aspectos.

A boa notícia, se você prefere o comportamento do lado do cliente do Git, mas está trabalhando com um projeto cujo código-fonte é controlado pelo Mercurial, é que existe uma maneira de usar o Git como um cliente para um repositório hospedado pelo Mercurial. Como o Git se comunica com os repositórios do servidor por meio de remotos, não é surpresa que essa ponte seja implementada como um auxiliar remoto. O nome do projeto é git-remote-hg e pode ser encontrado em https://github.com/felipec/git-remote-hg.

git-remote-hg

Primeiro, você precisa instalar o git-remote-hg. Isso basicamente envolve soltar seu arquivo em algum lugar no seu caminho, assim:

```
$ curl -o ~/bin/git-remote-hg \
https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg $
chmod +x ~/bin/git-remote -hg
```

…supondo que ~/bin esteja em seu $PATH. Git-remote-hg tem uma outra dependência: a biblioteca mercurial para Python. Se você tiver o Python instalado, isso é tão simples quanto:

```
$ pip instalar mercurial
```

Se você não tiver o Python instalado, visite https://www.python.org/ e pegue primeiro.

Machine Translated by Google

A última coisa que você precisa é do cliente Mercurial. Acesse https://www.mercurial-scm.org/ e instale-o se ainda não o fez.

Agora você está pronto para o rock. Tudo o que você precisa é de um repositório Mercurial para o qual você possa enviar. Felizmente, todos os repositórios do Mercurial podem agir dessa forma, então usaremos apenas o repositório "hello world" que todos usam para aprender o Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Começando

Agora que temos um repositório “do lado do servidor” adequado, podemos passar por um fluxo de trabalho típico. Como você verá, esses dois sistemas são semelhantes o suficiente para não haver muito atrito.

Como sempre com o Git, primeiro clonamos:

```
$ 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) Crie um makefile * 65bb417 Crie um programa
padrão 'hello, world'
```

Você notará que trabalhar com um repositório Mercurial usa o comando git clone padrão . Isso ocorre porque git-remote-hg está trabalhando em um nível bastante baixo, usando um mecanismo semelhante a como o protocolo HTTP/S do Git é implementado (remote helpers). Como o Git e o Mercurial foram projetados para que cada cliente tenha uma cópia completa do histórico do repositório, este comando faz um clone completo, incluindo todo o histórico do projeto, e o faz rapidamente.

O comando log mostra dois commits, o último dos quais é apontado por uma série de refs. Acontece que alguns deles não estão realmente lá. Vamos dar uma olhada no que realmente está no diretório .git :

```
$ tree .git/refs .git/
refs ÿÿÿ heads ÿ
 ÿÿÿ master ÿÿÿ
 hg ÿ ÿÿÿ origin ÿ
 ÿÿÿ bookmarks ÿ
 ÿ ÿÿÿ master ÿÿÿ ÿ
 branches ÿ default ÿÿ
 notas ÿ ÿÿÿ hg ÿÿÿ remotos
 ÿ ÿÿÿ origem
```

Machine Translated by Google

```
ÿ ÿÿÿ CABEÇA
ÿÿÿ        Tag

9 diretórios, 5 arquivos
```

O Git-remote-hg está tentando tornar as coisas mais idiomáticas no estilo Git, mas sob o capô está gerenciando o mapeamento conceitual entre dois sistemas ligeiramente diferentes. O diretório refs/hg é onde as referências remotas reais são armazenadas. Por exemplo, o refs/hg/origin/branches/default é um arquivo Git ref que contém o SHA-1 começando com “ac7955c”, que é o commit para o qual o master aponta. Portanto, o diretório refs/hg é como um falso refs/remotes/origin, mas tem a distinção adicional entre favoritos e ramificações.

O arquivo notes/hg é o ponto de partida para como o git-remote-hg mapeia hashes de commit do Git para IDs de changeset do Mercurial. Vamos explorar um pouco:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386... tree
1781c96... author remote-hg <>
1408066400 -0800 committer remote-hg <>
1408066400 -0800

Notas para mestre

$ git ls-tree 1781c96... 100644
blob ac9117f... 65bb417... 100644 blob
485e178... ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Então refs/notes/hg aponta para uma árvore, que no banco de dados de objetos do Git é uma lista de outros objetos com nomes. git ls-tree gera o modo, tipo, hash de objeto e nome de arquivo para itens dentro de uma árvore. Depois de pesquisar um dos itens da árvore, descobrimos que dentro dele há um blob chamado “ac9117f” (o hash SHA-1 do commit apontado pelo mestre), com conteúdo “0a04b98” (que é o ID do Conjunto de alterações do Mercurial na ponta da ramificação padrão).

A boa notícia é que, na maioria das vezes, não precisamos nos preocupar com tudo isso. O fluxo de trabalho típico não será muito diferente de trabalhar com um controle remoto do Git.

Há mais uma coisa que devemos observar antes de continuar: ignora. O Mercurial e o Git usam um mecanismo muito semelhante para isso, mas é provável que você não queira enviar um arquivo .gitignore para um repositório do Mercurial. Felizmente, o Git tem uma maneira de ignorar arquivos locais para um repositório em disco, e o formato Mercurial é compatível com o Git, então você só precisa copiá-lo:

```
$ cp .hgignore .git/info/exclude
```

Machine Translated by Google

O arquivo .git/info/exclude funciona exatamente como um .gitignore, mas não está incluído nos commits.

fluxo de trabalho

Vamos supor que fizemos algum trabalho e alguns commits no branch master , e você está pronto para enviá-lo para o repositório remoto. Aqui está a aparência do nosso repositório agora:

```
$ 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) Criar um makefile * 65bb417 Criar um programa
padrão 'hello, world'
```

Nosso branch master está dois commits à frente de origin/master, mas esses dois commits existem apenas em nossa máquina local. Vamos ver se mais alguém tem feito um trabalho importante ao mesmo tempo:

```
$ git fetch
De hg::/tmp/hello
ac7955c..df85e87 mestre                -> 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) Adicione alguma documentação | *
ba04a2a (HEAD, mestre) Atualizar makefile | * d25d16f Adeus |/ * ac7955c Crie
um makefile




* 65bb417 Crie um programa padrão 'olá, mundo'
```

Como usamos o sinalizador --all , vemos as referências de “notas” que são usadas internamente pelo git-remote-hg, mas podemos ignorá-las. O resto é o que esperávamos; origin/master avançou um commit, e nossa história agora divergiu. Ao contrário dos outros sistemas com os quais trabalhamos neste capítulo, o Mercurial é capaz de lidar com mesclagens, então não faremos nada extravagante.

```
$ git merge origin/master
Mesclagem automática
hello.c Mesclagem feita pela estratégia
'recursiva'. olá.c | 2 +- 1 arquivo alterado,
1 inserção(+), 1 exclusão(-) $ git log --oneline --
graph --decorate * 0c64627 (HEAD, master)
Mesclar branch de rastreamento remoto 'origin/master' |\ | * df85e87 (origin/
master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default,
refs/hg/origin/bookmarks/master) Adicione alguns
```

Machine Translated by Google

```
documentação *
| ba04a2a Atualizar makefile * |
d25d16f Adeus |/ * ac7955c Crie um
 makefile

* 65bb417 Crie um programa padrão 'olá, mundo'
```

Perfeito. Fazemos os testes e tudo passa, então estamos prontos para compartilhar nosso trabalho com o resto da equipe:

```
$ git push To
hg::/tmp/hello
df85e87..0c64627 master -> master
```

É isso! Se você der uma olhada no repositório do Mercurial, verá que isso fez o que esperávamos:

```
$ hg log -G --style compact 5[dica]:4,2
°          dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\|            Mesclar ramificação de rastreamento remoto 'origem/mestre'
| |
o 4 64f27bcefc35 2014-08-14 19:27 -0700 ben |
               Atualizar makefile | |
o 3:1 4256fc29598f 2014-08-14
19:27 -0700 ben |
               adeus |
@ | 2 7db0b4848b3c
2014-08-14 19:30 -0700 ben |/
               Adicione alguma documentação
| o 1 82e55d328c8c 26/08/2005 01:21
-0700 mpm | | o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
          Criar um makefile



          Crie um programa padrão 'olá, mundo'
```

O changeset de número *2* foi feito pelo Mercurial, e os changeset de número *3* e *4* foram feitos por git-remote-hg, por push de commits feitos com Git.

Ramos e marcadores

O Git tem apenas um tipo de ramificação: uma referência que se move quando os commits são feitos. No Mercurial, esse tipo de referência é chamado de “marcador” e se comporta da mesma maneira que uma ramificação do Git.

O conceito de “ramificação” da Mercurial é mais pesado. A ramificação em que um conjunto de alterações é feito é registrada *com o conjunto de alterações,* o que significa que sempre estará no histórico do repositório. Aqui está um exemplo de um commit que foi feito no branch develop :

Machine Translated by Google

```
$ hg log -l 1
changeset: 6:8f65e5e02793 branch:
develop tip Ben Straub <ben@straub.cc>
marcação:          Thu Aug 14
do utilizador:     20:06:38 2014 -0700 Mais
data:              documentação
resumo:
```

Observe a linha que começa com “branch”. O Git não pode realmente replicar isso (e não precisa; ambos os tipos de ramificação podem ser representados como uma referência do Git), mas git-remote-hg precisa entender a diferença, porque o Mercurial se importa.

Criar marcadores do Mercurial é tão fácil quanto criar ramificações do Git. Do lado do Git:

```
$ git checkout -b featureA
Mudou para um novo branch 'featureA' $ git
push origin featureA
Para hg::/tmp/olá *
[novo ramo]                recursoA -> recursoA
```

Isso é tudo. No lado do Mercurial, fica assim:

```
$ hg bookmarks
featureA $ hg log --                    5:bd5ac26f11f9
style compact -G @ 6[tip]
8f65e5e02793 2014-08-14 20:06 -0700 ben |
        Mais documentação |

◦       5[recursoA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
|\          Mesclar ramificação de rastreamento remoto 'origem/mestre'
| |
| o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben | atualizar makefile | |
o 3:1 318[illegible] 2014-08-14 20:01 -0700 ben | adeus | o | 2
  ''

  ''

          Adicione alguma documentação
| o 1 82e55d328c8c 26/08/2005 01:21
-0700 mpm | | o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
        Criar um makefile

        Crie um programa padrão 'olá, mundo'
```

Observe a nova tag [featureA] na revisão 5. Eles agem exatamente como as ramificações do Git no lado do Git, com

Machine Translated by Google

uma exceção: você não pode excluir um marcador do lado do Git (essa é uma limitação dos auxiliares remotos).

Você também pode trabalhar em um branch “heavyweight” do Mercurial: basta colocar um branch no namespace branches :

```
$ git checkout -b branches/permanent
Mudou para um novo branch 'branches/permanent' $ vi
Makefile $ git commit -am 'Uma mudança permanente' $ git
push origin branches/permanent

Para hg::/tmp/olá *
[novo ramo]                    filiais/permanentes -> filiais/permanentes
```

Aqui está o que parece no lado do Mercurial:

```
$ hg branches
desenvolvimento                    7:a4529d07aad4
permanente                         6:8f65e5e02793
padrão $ hg log                    5:bd5ac26f11f9 (inativo)
-G o changeset:
7:a4529d07aad4 | ramo: permanente |
 tag: dica | pai: 5:bd5ac26f11f9 | usuário:
 <ben@straub.cc> | data: quarta-feira,
 20:21:09 2014 -0700 | resumo: Uma
 permanente | | @ changeset:
 desenvolver | Ben Straub <ben@straub.cc> | data: qui 14 ago 2014 20:06:33 -0700 | resumo: Mais
 documentação |
```

```
        do utilizador:
        data:
        resumo:

◦       conjunto de alterações:
|\ marcador: | |
pai: usuário: | |
data: [a:] | resumo:
                5:bd5ac26f11f9
                featureA 4:0434aaa6b91f
                2: f098c7f45c4f
                Ben Straub <ben@straub.cc> Thu
                Aug 14 20:02:21 2014 -0700 Merge
                remote-tracking branch 'origin/master'
```

O nome da ramificação “permanente” foi registrado com o changeset marcado como 7.

Do lado do Git, trabalhar com qualquer um desses estilos de branch é o mesmo: apenas fazer checkout, commit, fetch, merge, pull e push como faria normalmente. Uma coisa que você deve saber é que o Mercurial não suporta reescrever o histórico, apenas adicionar a ele. Aqui está a aparência do nosso repositório Mercurial

Machine Translated by Google

após um rebase interativo e um push de força:

```
$ hg log --style compact -G o 10[dica]
99611176cbc9 2014-08-14 20:21 -0700 ben |
        Uma mudança permanente
| o 9 f23e12f939c3 2014-08-14
20:01 -0700 ben | | o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
        Adicione alguma documentação



        adeus

| | | o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben | | | | | @ 6 8f65e5e02793
14/08/2014 [illegible] ben |
adeus | | o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben |/

            Mais documentação

          5[recursoA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
            Mesclar ramificação de rastreamento remoto 'origem/mestre'



            atualizar makefile
      ''



    ''

          Adicione alguma documentação
| o 1 82e55d328c8c 26/08/2005 01:21
-0700 mpm | | o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
        Criar um makefile



        Crie um programa padrão "olá, mundo"
```

Os conjuntos de alterações 8, 9 e *10* foram criados e pertencem à ramificação permanente , mas os conjuntos de alterações antigos ainda estão lá. Isso pode ser **muito** confuso para seus colegas de equipe que estão usando o Mercurial, então tente evitá-lo.

Resumo do Mercurial

Git e Mercurial são semelhantes o suficiente para que trabalhar além da fronteira seja bastante indolor. Se você evitar alterar o histórico deixado em sua máquina (como geralmente é recomendado), talvez nem saiba que a outra ponta é o Mercurial.

Git e Bazar

Entre os DVCS, outro famoso é o Bazaar. O Bazaar é gratuito e de código aberto e faz parte do

Machine Translated by Google

Projeto GNU. Ele se comporta de maneira muito diferente do Git. Às vezes, para fazer a mesma coisa que com o Git, você precisa usar uma palavra-chave diferente, e algumas palavras-chave comuns não têm o mesmo significado. Em particular, o gerenciamento de filiais é muito diferente e pode causar confusão, principalmente quando alguém vem do universo Git. No entanto, é possível trabalhar em um repositório Bazaar a partir de um Git.

Existem muitos projetos que permitem usar o Git como um cliente Bazaar. Aqui usaremos o projeto de Felipe Contreras que você encontra em https://github.com/felipec/git-remote-bzr. Para instalá-lo, basta baixar o arquivo git-remote-bzr em uma pasta contida em seu $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
```

Você também precisa ter o Bazaar instalado. Isso é tudo!

Crie um repositório Git a partir de um repositório Bazaar

É simples de usar. Basta clonar um repositório Bazaar prefixando-o por bzr::. Como o Git e o Bazaar fazem clones completos em sua máquina, é possível anexar um clone do Git ao seu clone local do Bazaar, mas isso não é recomendado. É muito mais fácil anexar seu clone do Git diretamente ao mesmo local ao qual seu clone do Bazaar está anexado — o repositório central.

Vamos supor que você trabalhou com um repositório remoto que está no endereço bzr+ssh://developer@mybazaarserver:myproject. Então você deve cloná-lo da seguinte maneira:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git $
cd myProject-Git
```

Neste ponto, seu repositório Git é criado, mas não é compactado para uso otimizado do disco. É por isso que você também deve limpar e compactar seu repositório Git, especialmente se for grande:

```
$ git gc --aggressive
```

filiais do bazar

O Bazaar permite apenas clonar branches, mas um repositório pode conter vários branches, e o git remote-bzr pode clonar ambos. Por exemplo, para clonar uma ramificação:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

E para clonar todo o repositório:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

Machine Translated by Google

O segundo comando clona todas as ramificações contidas no repositório emacs; no entanto, é possível apontar alguns ramos:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Alguns repositórios remotos não permitem que você liste suas ramificações; nesse caso, você deve especificá-los manualmente e, embora possa especificar a configuração no comando de clonagem, isso pode ser mais fácil:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs $
git config remote-bzr.branches 'trunk, xwindow' $ git fetch
```

Ignore o que é ignorado com .bzrignore

Como você está trabalhando em um projeto gerenciado com o Bazaar, você não deve criar um arquivo .gitignore porque *pode* acidentalmente configurá-lo sob controle de versão e as outras pessoas que trabalham com o Bazaar podem ser incomodadas. A solução é criar o arquivo .git/info/exclude como um link simbólico ou como um arquivo normal. Veremos mais adiante como resolver esta questão.

O Bazaar usa o mesmo modelo do Git para ignorar arquivos, mas também possui dois recursos que não têm equivalente no Git. A descrição completa pode ser encontrada na documentação. as duas características são:

1. "!!" permite que você ignore certos padrões de arquivo, mesmo que sejam especificados usando um "!" regra.
2. "RE:" no início de uma linha permite que você especifique uma expressão regular do Python (Somente Git permite globos de concha).

Como consequência, existem duas situações diferentes a considerar:

1. Se o arquivo .bzrignore não contiver nenhum desses dois prefixos específicos, você pode simplesmente faça um link simbólico para ele no repositório: ln -s .bzrignore .git/info/exclude.
2. Caso contrário, você deve criar o arquivo .git/info/exclude e adaptá-lo para ignorar exatamente o mesmo arquivos em .bzrignore.

Seja qual for o caso, você terá que permanecer vigilante contra qualquer alteração de .bzrignore para garantir que o arquivo .git/info/exclude sempre reflita .bzrignore. De fato, se o arquivo .bzrignore fosse alterado e contivesse uma ou mais linhas começando com "!!" ou "RE:", não podendo o Git interpretar essas linhas, você terá que adaptar seu arquivo .git/info/exclude para ignorar os mesmos arquivos que foram ignorados com .bzrignore. Além disso, se o arquivo .git/info/exclude for um link simbólico, você terá que primeiro deletar o link simbólico, copiar .bzrignore para .git/info/exclude e depois adaptar este último.

No entanto, tenha cuidado com sua criação, pois com o Git é impossível reincluir um arquivo se um diretório pai desse arquivo for excluído.

Machine Translated by Google

Buscar as alterações do repositório remoto

Para buscar as alterações do controle remoto, você puxa as alterações normalmente, usando comandos do Git. Supondo que suas alterações estejam na ramificação master , você mescla ou rebase seu trabalho na ramificação origin/master :

```
$ git pull --rebase origem
```

Envie seu trabalho para o repositório remoto

Como o Bazaar também tem o conceito de merge commits, não haverá problema se você enviar um merge commit. Assim, você pode trabalhar em uma ramificação, mesclar as alterações no mestre e enviar seu trabalho.
Em seguida, você cria suas ramificações, testa e confirma seu trabalho como de costume. Você finalmente envia seu trabalho para o repositório do Bazaar:

```
$ git push mestre de origem
```

Ressalvas

A estrutura de ajuda remota do Git tem algumas limitações que se aplicam. Em particular, esses comandos não funcionam:

- git push origin :branch-to-delete (o Bazaar não pode aceitar ref deleções desta forma)
- git push origin old:new (enviará push old)
- git push --dry-run origin branch (enviará push)

Resumo

Como os modelos do Git e do Bazaar são semelhantes, não há muita resistência ao trabalhar além do limite. Contanto que você esteja atento às limitações e esteja sempre ciente de que o repositório remoto não é nativamente Git, você ficará bem.

Git e Perforce

Perforce é um sistema de controle de versão muito popular em ambientes corporativos. Ele existe desde 1995, o que o torna o sistema mais antigo abordado neste capítulo. Como tal, foi projetado com as restrições de sua época; ele assume que você está sempre conectado a um único servidor central e apenas uma versão é mantida no disco local. Para ter certeza, seus recursos e restrições são adequados para vários problemas específicos, mas há muitos projetos usando Perforce onde o Git realmente funcionaria melhor.

Há duas opções se você quiser misturar o uso do Perforce e do Git. A primeira que abordaremos é a ponte “Git Fusion” dos criadores do Perforce, que permite expor subárvores de seu depósito Perforce como repositórios Git de leitura/gravação. O segundo é o git-p4, uma ponte do lado do cliente que permite usar o Git como um cliente Perforce, sem exigir nenhuma reconfiguração do servidor Perforce.

Machine Translated by Google

Git Fusion

A Perforce fornece um produto chamado Git Fusion (disponível em http://www.perforce.com/git-fusion), que sincroniza um servidor Perforce com repositórios Git no lado do servidor.

Configurando

Para nossos exemplos, usaremos o método de instalação mais fácil para o Git Fusion, que é o download de uma máquina virtual que executa o daemon Perforce e o Git Fusion. Você pode obter a imagem da máquina virtual em http://www.perforce.com/downloads/Perforce/20-User, e quando terminar o download, importe-o para o seu software de virtualização favorito (usaremos o VirtualBox).

Ao iniciar a máquina pela primeira vez, ela solicita que você personalize a senha para três usuários do Linux (root, perforce e git) e forneça um nome de instância, que pode ser usado para distinguir esta instalação de outras na mesma rede. Quando tudo estiver concluído, você verá isto:

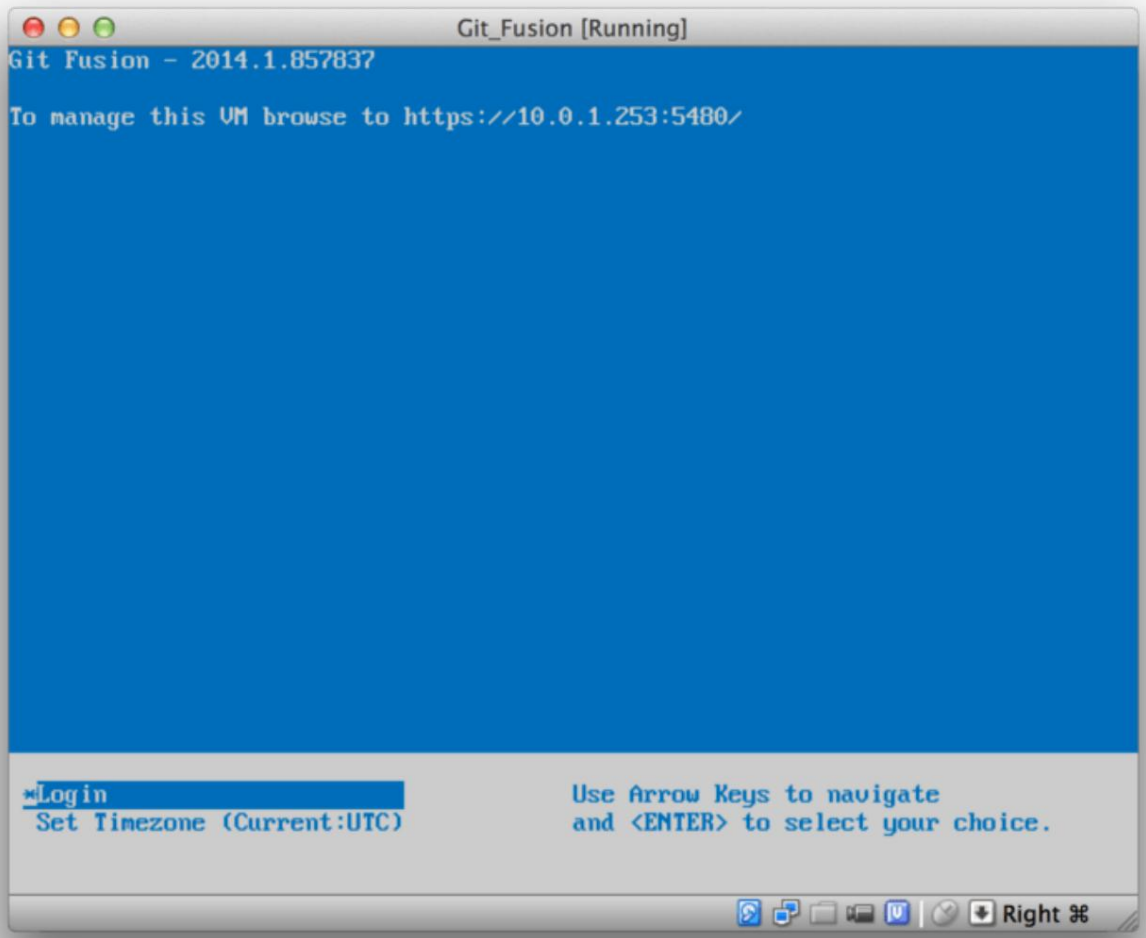


Figura 145. A tela de inicialização da máquina virtual Git Fusion

Você deve anotar o endereço IP mostrado aqui, vamos usá-lo mais tarde. Em seguida, criaremos um usuário Perforce. Selecione a opção “Login” na parte inferior e pressione enter (ou SSH para a máquina) e faça login como root. Em seguida, use estes comandos para criar um usuário:

Machine Translated by Google

```
$ p4 -p localhost:1666 -u superusuário -f john $ p4 -p
localhost:1666 -u john passwd $ exit
```

O primeiro abrirá um editor de VI para personalizar o usuário, mas você pode aceitar os padrões digitando :wq e pressionando enter. O segundo solicitará que você digite uma senha duas vezes. Isso é tudo que precisamos fazer com um prompt de shell, então saia da sessão.

A próxima coisa que você precisa fazer para acompanhar é dizer ao Git para não verificar os certificados SSL. A imagem do Git Fusion vem com um certificado, mas é para um domínio que não corresponde ao endereço IP da sua máquina virtual, então o Git rejeitará a conexão HTTPS. Se esta for uma instalação permanente, consulte o manual Perforce Git Fusion para instalar um certificado diferente; para nossos propósitos de exemplo, isso será suficiente:

```
$ exportação GIT_SSL_NO_VERIFY=verdadeiro
```

Agora podemos testar se tudo está funcionando.

```
$ git clone https://10.0.1.254/Talkhouse Clonagem em
'Talkhouse'...
Nome de usuário para 'https://10.0.1.254': john
Senha para 'https://john@10.0.1.254': remote:
Counting objects: 630, done. remote: Comprimindo
objetos: 100% (581/581), concluído. remoto: Total 630 (delta 172),
reutilizado 0 (delta 0)
Recebendo objetos: 100% (630/630), 1,22 MiB | 0 byte/s, pronto.
Resolvendo deltas: 100% (172/172), concluído.
Verificando a conectividade... concluído.
```

A imagem da máquina virtual vem equipada com um projeto de amostra que você pode clonar. Aqui estamos clonando por HTTPS, com o usuário john que criamos acima; O Git solicita credenciais para esta conexão, mas o cache de credenciais nos permitirá pular esta etapa para quaisquer solicitações subsequentes.

Configuração de Fusão

Depois de instalar o Git Fusion, você desejará ajustar a configuração. Na verdade, isso é bastante fácil de fazer usando seu cliente Perforce favorito; apenas mapeie o diretório //.git-fusion no servidor Perforce em seu espaço de trabalho. A estrutura do arquivo fica assim:

```
$ árvore
.
 ÿÿÿ objetos ÿ ÿÿÿ
 repositórios ÿ ÿ ÿÿÿ
 [...] ÿ ÿÿÿ árvores ÿ ÿÿÿ [...]
```

Machine Translated by Google

```
ÿ ÿÿÿ p4gf_config
ÿÿÿ repositórios ÿ
ÿÿÿ Talkhouse ÿ ÿÿÿ
p4gf_config
ÿÿÿ        Usuários
      ÿÿÿ        p4gf_usermap

498 diretórios, 287 arquivos
```

O diretório de objetos é usado internamente pelo Git Fusion para mapear objetos do Perforce para o Git e vice-versa, você não terá que mexer em nada lá. Há um arquivo p4gf_config global neste diretório, bem como um para cada repositório – esses são os arquivos de configuração que determinam como o Git Fusion se comporta. Vamos dar uma olhada no arquivo na raiz:

```
[repo-criação]
charset = utf8

[git-to-perforce]
change-owner = autor
enable-git-branch-creation = sim enable-
swarm-reviews = sim enable-git-merge-
commits = sim enable-git-submodules
= sim preflight-commit = nenhum
ignorar -autor-permissões = sem
verificação de permissão de leitura =
nenhum git-merge-avoidance-after-
change-num = 12107

[perforce-to-git] http-
url = nenhum ssh-
url = nenhum

[@features]
imports = False
chunked-push = False
matrix2 = False parallel-
push = False

[autenticação]
email-case-sensitivity = não
```

Não entraremos no significado desses sinalizadores aqui, mas observe que este é apenas um arquivo de texto formatado em INI, muito parecido com o que o Git usa para configuração. Esse arquivo especifica as opções globais, que podem ser substituídas por arquivos de configuração específicos do repositório, como repos/Talkhouse/p4gf_config. Se você abrir este arquivo, verá uma seção [@repo] com algumas configurações diferentes dos padrões globais.
Você também verá seções como esta:

Machine Translated by Google

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Este é um mapeamento entre uma ramificação do Perforce e uma ramificação do Git. A seção pode ter o nome que você quiser, desde que o nome seja exclusivo. git-branch-name permite converter um caminho de depósito que seria complicado no Git para um nome mais amigável. A configuração de visualização controla como os arquivos Perforce são mapeados no repositório Git, usando a sintaxe de mapeamento de visualização padrão. Mais de um mapeamento pode ser especificado, como neste exemplo:

```
[multi-project-mapping] git-
branch-name = master view
= //depot/project1/main/... project1/...
          //depósito/projeto2/linha principal/... projeto2/...
```

Dessa forma, se o mapeamento normal do espaço de trabalho incluir alterações na estrutura dos diretórios, você poderá replicá-lo com um repositório Git.

O último arquivo que discutiremos é users/p4gf_usermap, que mapeia usuários do Perforce para usuários do Git, e que você pode nem precisar. Ao converter de um conjunto de alterações do Perforce para um commit do Git, o comportamento padrão do Git Fusion é procurar o usuário do Perforce e usar o endereço de e-mail e o nome completo armazenados lá para o campo autor/committer no Git. Ao converter de outra maneira, o padrão é procurar o usuário do Perforce com o endereço de e-mail armazenado no campo de autor do commit do Git e enviar o conjunto de alterações como esse usuário (com a aplicação de permissões). Na maioria dos casos, esse comportamento funcionará bem, mas considere o seguinte arquivo de mapeamento:

```
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"
```

Cada linha está no formato <usuário> <email> "<nome completo>" e cria um único mapeamento de usuário. As duas primeiras linhas mapeiam dois endereços de e-mail distintos para a mesma conta de usuário do Perforce. Isso é útil se você criou Git commits em vários endereços de e-mail diferentes (ou alterou endereços de e-mail), mas deseja que eles sejam mapeados para o mesmo usuário do Perforce. Ao criar um commit do Git a partir de um conjunto de alterações do Perforce, a primeira linha correspondente ao usuário do Perforce é usada para informações de autoria do Git.

As duas últimas linhas mascaram os nomes reais de Bob e Joe e endereços de e-mail dos commits do Git que são criados. Isso é bom se você deseja abrir o código de um projeto interno, mas não deseja publicar seu diretório de funcionários para o mundo inteiro. Observe que os endereços de e-mail e os nomes completos devem ser exclusivos, a menos que você queira que todos os commits do Git sejam atribuídos a um único autor fictício.

fluxo de trabalho

O Perforce Git Fusion é uma ponte bidirecional entre o Perforce e o controle de versão do Git. Vamos dar uma olhada em como é trabalhar do lado do Git. Assumiremos que mapeamos no projeto “Jam” usando um

Machine Translated by Google

arquivo de configuração como mostrado acima, que podemos clonar assim:

```
$ git clone https://10.0.1.254/Jam
Clonagem em 'Jam'...
Nome de usuário para 'https://10.0.1.254': john
Senha para 'https://john@10.0.1.254': remote:
Counting objects: 2070, done. remoto:
Comprimindo objetos: 100% (1704/1704), concluído.
Recebendo objetos: 100% (2070/2070), 1,21 MiB | 0 byte/s, pronto. remoto:
Total 2070 (delta 1242), reutilizado 0 (delta 0)
Resolvendo deltas: 100% (1242/1242), concluído.
Verificando a conectividade...
concluído. $ git ramo -a
* mestre
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 filial. | * d254865
(HEAD, origin/master, origin/HEAD, master) Atualize para os
metrowerks mais recentes no Beos -- o da Intel. | * bd2f54a Corrige o vazamento do identificador NT do
congestionamento. | * c0f29e7 Corrigir URL em um documento jam | * Porta lynx de Radstone cc644ac.
[...]
```

A primeira vez que você fizer isso, pode levar algum tempo. O que está acontecendo é que o Git Fusion está convertendo todos os conjuntos de alterações aplicáveis no histórico do Perforce em confirmações do Git. Isso acontece localmente no servidor, então é relativamente rápido, mas se você tiver muito histórico, ainda pode demorar um pouco. As buscas subsequentes fazem uma conversão incremental, então parecerá mais com a velocidade nativa do Git.

Como você pode ver, nosso repositório se parece exatamente com qualquer outro repositório Git com o qual você possa trabalhar. Existem três branches, e o Git criou um branch master local que rastreia origin/master. Vamos trabalhar um pouco e criar alguns novos commits:

```
# ...
$ git log --oneline --decorate --graph --all * cfd46ab
(HEAD, master) Adicionar documentação para novo recurso * a730d77
Whitespace * d254865 (origin/master, origin/HEAD) Atualize para os
metrowerks mais recentes no Beos -- o Intel um.

* bd2f54a Corrige o vazamento do identificador NT do congestionamento.
[...]
```

Temos dois novos commits. Agora vamos verificar se mais alguém está trabalhando:

```
$ git fetch
remote: Contando objetos: 5, concluído.
```

Machine Translated by Google

```
remoto: Comprimindo objetos: 100% (3/3), concluído. remoto:
Total 3 (delta 2), reutilizado 0 (delta 0)
Desempacotando objetos: 100% (3/3), concluído.
De https://10.0.1.254/Jam
d254865..6afeb15 mestre                    -> origin/master $ git
log --oneline --decorate --graph --all * 6afeb15 (origin/master,
origin/HEAD) Atualizar direitos autorais | * cfd46ab (HEAD, mestre)
 Adicionar documentação para novo recurso | * a730d77 Whitespace |/ * d254865
 Atualize para os metrowerks mais recentes no Beos -- o da Intel. * bd2f54a Corrige
 o vazamento do identificador NT do congestionamento. [...]
```

Parece que alguém foi! Você não saberia desta visão, mas o commit 6afeb15 foi realmente criado usando um cliente Perforce. Parece apenas outro commit do ponto de vista do Git, que é exatamente o ponto. Vamos ver como o servidor Perforce lida com um merge commit:

```
$ git merge origin/master
Mesclagem automática README
Mesclagem feita pela estratégia 'recursiva'.
LEIA-ME | 2 +- 1
arquivo alterado, 1 inserção(+), 1 exclusão(-) $ git push
Contando objetos: 9, concluído.

Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (9/9), concluído.
Objetos de escrita: 100% (9/9), 917 bytes | 0 byte/s, pronto.
Total 9 (delta 6), reutilizado 0 (delta 0) remote:
Perforce: 100% (3/3) Carregando árvore de confirmação na memória... remote:
Perforce: 100% (5/5) Encontrando commits filhos... remoto : Perforce: Executando git
fast-export... remote: Perforce: 100% (3/3) Verificando commits... remote: O
processamento continuará mesmo se a conexão for fechada. remoto: Perforce: 100%
(3/3) Copiando listas de mudanças... remoto: Perforce: Enviando novos objetos Git
commit para Perforce: 4 To https://10.0.1.254/Jam 6afeb15..89cba2b master -> master
```

Git acha que funcionou. Vamos dar uma olhada no histórico do arquivo README do ponto de vista do Perforce, usando o recurso gráfico de revisão do p4v:

Machine Translated by Google

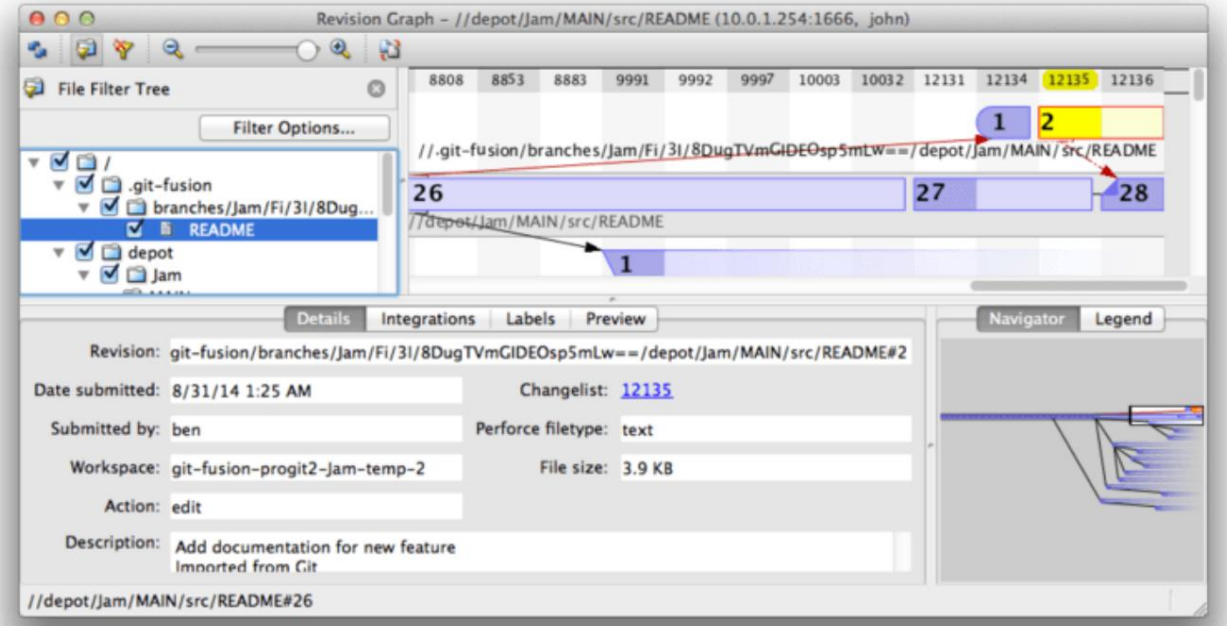


Figura 146. Gráfico de revisão Perforce resultante do Git push

Se você nunca viu essa visualização antes, pode parecer confuso, mas mostra os mesmos conceitos de um visualizador gráfico para o histórico do Git. Estamos analisando o histórico do arquivo README , portanto, a árvore de diretórios no canto superior esquerdo mostra apenas esse arquivo à medida que ele aparece em várias ramificações. No canto superior direito, temos um gráfico visual de como as diferentes revisões do arquivo estão relacionadas, e a visão geral desse gráfico está no canto inferior direito. O restante da visualização é dado à visualização de detalhes da revisão selecionada (2 neste caso).

Uma coisa a notar é que o gráfico se parece exatamente com o histórico do Git. O Perforce não tinha um branch nomeado para armazenar os commits 1 e 2 , então criou um branch “anônimo” no diretório .git-fusion para mantê-lo. Isso também acontecerá para ramificações Git nomeadas que não correspondem a uma ramificação Perforce nomeada (e você pode mapeá-las posteriormente para uma ramificação Perforce usando o arquivo de configuração).

A maior parte disso acontece nos bastidores, mas o resultado final é que uma pessoa em uma equipe pode estar usando o Git, outra pode estar usando o Perforce e nenhum deles saberá sobre a escolha do outro.

Resumo do Git-Fusion

Se você tem (ou pode obter) acesso ao seu servidor Perforce, o Git Fusion é uma ótima maneira de fazer o Git e o Perforce conversarem entre si. Há um pouco de configuração envolvida, mas a curva de aprendizado não é muito íngreme. Esta é uma das poucas seções neste capítulo em que os cuidados sobre o uso de toda a potência do Git não serão exibidos. Isso não quer dizer que o Perforce ficará feliz com tudo que você jogar nele – se você tentar reescrever a história que já foi empurrada, o Git Fusion irá rejeitá-lo – mas o Git Fusion se esforça muito para se sentir nativo. Você pode até usar submódulos do Git (embora pareçam estranhos para os usuários do Perforce) e mesclar ramificações (isso será registrado como uma integração no lado do Perforce).

Se você não conseguir convencer o administrador do seu servidor a configurar o Git Fusion, ainda existe uma maneira de usar essas ferramentas juntas.

Machine Translated by Google

Git-p4

Git-p4 é uma ponte bidirecional entre Git e Perforce. Ele roda inteiramente dentro do seu repositório Git, então você não precisará de nenhum tipo de acesso ao servidor Perforce (além das credenciais do usuário, é claro).
O Git-p4 não é uma solução tão flexível ou completa quanto o Git Fusion, mas permite que você faça a maior parte do que deseja sem ser invasivo para o ambiente do servidor.

ÿ

Você precisará da ferramenta p4 em algum lugar do seu PATH para trabalhar com git-p4. No momento em que este livro foi escrito, ele estava disponível gratuitamente em http://www.perforce.com/downloads/Perforce/20-Do utilizador.

Configurando

Para fins de exemplo, estaremos executando o servidor Perforce do Git Fusion OVA como mostrado acima, mas vamos ignorar o servidor Git Fusion e ir diretamente para o controle de versão do Perforce.

Para usar o cliente de linha de comando p4 (do qual git-p4 depende), você precisará definir algumas variáveis de ambiente:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Começando

Como em qualquer coisa no Git, o primeiro comando é clonar:

```
$ git p4 clone //depot/www/live www-shallow
Importando de //depot/www/live para www-shallow
Repositório Git vazio inicializado em /private/tmp/www-shallow/.git/ Fazendo
a importação inicial de // depot/www/live/ da revisão #head into refs/remotes/
p4/master
```

Isso cria o que em termos do Git é um clone “superficial”; apenas a versão mais recente do Perforce é importada para o Git; lembre-se, o Perforce não foi projetado para fornecer todas as revisões a todos os usuários. Isso é suficiente para usar o Git como um cliente Perforce, mas para outras finalidades não é suficiente.

Depois de concluído, temos um repositório Git totalmente funcional:

```
$ cd myproject
$ git log --oneline --all --graph --decorate * 70eaf78
(HEAD, p4/master, p4/HEAD, master) Importação inicial de //depot/www/live/ do estado na revisão
# cabeça
```

Observe como há um controle remoto “p4” para o servidor Perforce, mas todo o resto se parece com um clone padrão. Na verdade, isso é um pouco enganador; não há realmente um controle remoto lá.

Machine Translated by Google

```
$ git remoto -v
```

Nenhum controle remoto existe neste repositório. Git-p4 criou alguns refs para representar o estado do servidor, e eles se parecem com refs remotos para git log, mas eles não são gerenciados pelo próprio Git, e você não pode enviar para eles.

fluxo de trabalho

Ok, vamos fazer algum trabalho. Vamos supor que você tenha feito algum progresso em um recurso muito importante e esteja pronto para mostrá-lo ao resto de sua equipe.

```
$ git log --oneline --all --graph --decorate * 018467c (HEAD,
master) Alterar título da página * c0fb617 Atualizar link *
70eaf78 (p4/master, p4/HEAD) Importação inicial de //depot/
www/live / do estado na revisão #head
```

Fizemos dois novos commits que estamos prontos para enviar ao servidor Perforce. Vamos verificar se mais alguém estava trabalhando hoje:

```
$ sincronização git p4
sincronização git p4
Executando importação incremental em refs/remotes/p4/master git branch
Caminhos do depósito: //depot/www/live/
Destino de importação: refs/remotes/p4/master
Importando revisão 12142 (100%) $ git
log --oneline --all --graph --decorate * 75cd059 (p4/master,
p4/HEAD) Atualizar direitos autorais | * 018467c (HEAD, mestre)
 Alterar título da página | * c0fb617 Link de atualização |/ *
 70eaf78 Importação inicial de //depot/www/live/ do estado na
 revisão #head
```

Parece que sim, e master e p4/master divergiram. O sistema de ramificação do Perforce não é *nada* parecido com o do Git, portanto, enviar confirmações de mesclagem não faz sentido. O Git-p4 recomenda que você rebase seus commits e ainda vem com um atalho para fazer isso:

```
$ git p4 rebase
Executando importação incremental em refs/remotes/p4/master git branch Depot paths: //
depot/www/live/ Nenhuma alteração na importação!

Rebaseando o branch atual em remotes/p4/master Primeiro,
rebobinando a cabeça para reproduzir seu trabalho em cima dela...
Aplicando: Atualizar link
Aplicando: Alterar título da página
index.html | 2 +-
```

Machine Translated by Google

```
1 arquivo alterado, 1 inserção (+), 1 exclusão (-)
```

Você provavelmente pode dizer pela saída, mas git p4 rebase é um atalho para git p4 sync seguido por git rebase p4/master. É um pouco mais inteligente do que isso, especialmente ao trabalhar com várias ramificações, mas é uma boa aproximação.

Agora nossa história é linear novamente e estamos prontos para contribuir com nossas mudanças de volta ao Perforce. O comando git p4 submit tentará criar uma nova revisão Perforce para cada confirmação do Git entre p4/master e master. Executá-lo nos leva ao nosso editor favorito, e o conteúdo do arquivo é mais ou menos assim:

```
# Uma especificação de mudança forçada. #

# Mudar:          O número da mudança. 'novo' em uma nova lista de alterações.
# Data:           A data em que esta especificação foi modificada pela última vez.
# Cliente:        O cliente no qual a lista de alterações foi criada. Somente leitura.
# Do utilizador:  O usuário que criou a lista de alterações.
# Status:         Ou 'pendente' ou 'enviado'. Somente leitura.
# Tipo: #         Seja 'público' ou 'restrito'. O padrão é 'público'.
Descrição: Comentários sobre a changelist. Obrigatório.
# Jobs: Quais jobs abertos devem ser fechados por esta changelist.
#               Você pode excluir trabalhos desta lista. (Apenas novas listas de alterações.)
# Arquivos:     Quais arquivos abertos da lista de alterações padrão devem ser adicionados a
# #             esta lista de alterações. Você pode excluir arquivos desta lista.
                (Apenas novas listas de alterações.)

Mudança: novo

Cliente: john_bens-mbp_8487

usuário: joão

Estado: novo

Descrição:
Atualizar link


Arquivos: //depot/www/live/index.html # edit


######## git author ben@straub.cc não corresponde à sua conta p4.
######## Use a opção --preserve-user para modificar a autoria.
######## A variável git-p4.skipUserNameCheck oculta esta mensagem. ########
tudo abaixo desta linha é apenas a diferença ####### --- //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- 31/08 18:26:05.000000000 0000

@@ -60,7 +60,7 @@
</td>
```

Machine Translated by Google

```
<td valign=top>
Fonte e documentação para -<a
href="http://www.perforce.com/jam/jam.html"> +<a
href="jam.html"> Jam/MR</a >, uma ferramenta de
construção de software.

</td>
```

Este é basicamente o mesmo conteúdo que você veria executando p4 submit, exceto as coisas no final que o git-p4 incluiu de maneira útil. O Git-p4 tenta honrar suas configurações do Git e do Perforce individualmente quando precisa fornecer um nome para um commit ou changeset, mas em alguns casos você deseja substituí-lo. Por exemplo, se o commit do Git que você está importando foi escrito por um colaborador que não tem uma conta de usuário do Perforce, você ainda pode querer que o conjunto de alterações resultante pareça que foi escrito por ele (e não por você).

O Git-p4 importou de maneira útil a mensagem do commit do Git como conteúdo para este conjunto de alterações do Perforce, então tudo o que precisamos fazer é salvar e sair duas vezes (uma vez para cada commit). A saída do shell resultante será mais ou menos assim:

```
$ git p4 submit
Perforce checkout para o caminho do depósito //depot/www/live/ localizado em /Users/ben/john_bens
mbp_8487/john_bens-mbp_8487/depot/www/live/ Sincronizando p4 checkout... - file(s) up- a data.

 ...
Aplicando o link de atualização do
dbac45b //depot/www/live/index.html#4 - aberto para edição
Alteração 12143 criada com 1 arquivo(s) aberto(s).
Enviando alteração 12143.
Bloqueando 1
arquivos ... editar //depot/www/live/
index.html#5 Alteração 12143 enviada.
Aplicando 905ec6a Alterar título da página //
depot/www/live/index.html#5 - aberto para edição Alteração
12144 criada com 1 arquivo(s) aberto(s).
Enviando alteração 12144.
Bloqueando 1
arquivos ... editar //depot/www/live/
index.html#6 Alteração 12144 enviada.
Todos os commits aplicados!
Executando importação incremental em refs/remotes/p4/master git branch Depot
paths: //depot/www/live/ Import destination: refs/remotes/p4/master Importando
revisão 12144 (100%)

Rebaseando o branch atual em remotes/p4/master Primeiro,
rebobinando a cabeça para reproduzir seu trabalho em cima dela... $
git log --oneline --all --graph --decorate * 775a46f (HEAD, p4/master,
p4 /HEAD, master) Mudar título da página * 05f1ade Atualizar link * 75cd059
Atualizar direitos autorais
```

Machine Translated by Google

```
* 70eaf78 Importação inicial de //depot/www/live/ do estado na revisão #head
```

O resultado é como se tivéssemos feito um git push, que é a analogia mais próxima do que realmente aconteceu.

Observe que, durante esse processo, cada confirmação do Git é transformada em um conjunto de alterações do Perforce; se você quiser esmagá-los em um único conjunto de alterações, poderá fazer isso com um rebase interativo antes de executar git p4 submit. Observe também que os hashes SHA-1 de todos os commits que foram enviados como conjuntos de alterações foram alterados; isso ocorre porque o git-p4 adiciona uma linha ao final de cada commit que ele converte:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Autor: John Doe <john@example.com> Data: Dom 31 de
agosto 10:31:44 2014 -0800

Alterar título da página

[git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

O que acontece se você tentar enviar um merge commit? Vamos tentar. Aqui está a situação em que nos metemos:

```
$ git log --oneline --all --graph --decorate * 3be6fd8
(HEAD, master) Endereço de e-mail correto * 1dcbf21
Merge remote-tracking branch 'p4/master' |\ | * c4689fc (p4/
 master, p4/HEAD) Correção de gramática * | cbacd0a Bordas
 da tabela: sim por favor * | b4959b6 Marca registrada |/ *
775a46f Alterar título da página * 05f1ade Atualizar link *
75cd059 Atualizar direitos autorais * 70eaf78 Importação inicial
 de //depot/www/live/ do estado na revisão #head
```

A história do Git e do Perforce diverge após 775a46f. O lado do Git tem dois commits, então um merge commit com o cabeçalho Perforce, então outro commit. Vamos tentar enviá-los em cima de um único conjunto de alterações no lado do Perforce. Vamos ver o que aconteceria se tentássemos enviar agora:

```
$ git p4 submit -n
Verificação forçada para o caminho do depósito //depot/www/live/ localizado em /Users/ben/
john_bens mbp_8487/john_bens-mbp_8487/depot/www/live/
Sincronizaria o checkout p4 em /Users/ben/john_bens-mbp_8487/john_bens mbp_8487/
depot/www/live/
Aplicaria
b4959b6 Marca
registrada cbacd0a Bordas da tabela: sim, por favor
```

Machine Translated by Google

```
3be6fd8 Endereço de e-mail correto
```

O sinalizador -n é a abreviação de --dry-run, que tenta relatar o que aconteceria se o comando submit fosse executado de verdade. Nesse caso, parece que estaríamos criando três conjuntos de alterações do Perforce, que correspondem aos três commits não mesclados que ainda não existem no servidor Perforce. Isso parece exatamente o que queremos, vamos ver como fica:

```
$ git p4 submit […]
$ git log --oneline --
all --graph --decorate * dadbd89 (HEAD, p4/master, p4/HEAD,
master) Endereço de e-mail correto * 1b79a80 Bordas da tabela: sim, por favor * 0097235
Marca registrada * c4689fc Correção de gramática * 775a46f Alterar título da página *
05f1ade Atualizar link * 75cd059 Atualizar direitos autorais * 70eaf78 Importação inicial de //
depot/www/live/ do estado na revisão #head
```

Nossa história tornou-se linear, como se tivéssemos rebaseado antes de enviar (o que foi exatamente o que aconteceu). Isso significa que você pode ser livre para criar, trabalhar, descartar e mesclar ramificações no lado do Git sem medo de que seu histórico de alguma forma se torne incompatível com o Perforce. Se você pode rebaseá-lo, você pode contribuir com um servidor Perforce.

Ramificação

Se o seu projeto Perforce tiver várias ramificações, você não está sem sorte; O git-p4 pode lidar com isso de uma maneira que pareça com o Git. Digamos que seu depósito Perforce esteja organizado assim:

```
//depósito
    ÿÿÿ        projeto
ÿÿÿ principal ÿÿÿ dev
```

E digamos que você tenha uma ramificação dev , que tem uma especificação de exibição semelhante a esta:

```
//depósito/projeto/principal/... //depósito/projeto/dev/...
```

O Git-p4 pode detectar automaticamente essa situação e fazer a coisa certa:

```
$ git p4 clone --detect-branches //depot/project@all Importando de //
depot/project@all para o projeto Repositório Git vazio inicializado em /
private/tmp/project/.git/ Importando revisão 20 (50%)

Importando novo projeto/desenvolvimento de ramificação
```

Machine Translated by Google

```
Retomando com alteração 20
Importando revisão 22 (100%)
Branches atualizados: main
dev $ cd project; git log --oneline --all --graph --decorate *
eae77ae (HEAD, p4/master, p4/HEAD, master) main | *
10d55fb (p4/projeto/dev) dev | * a43cfae Populate //depósito/
projeto/principal/... //depósito/projeto/dev/.... |/ * 2b83451 Inicialização do
projeto
```

Observe o especificador “@all” no caminho do depósito; que diz ao git-p4 para clonar não apenas o conjunto de alterações mais recente para essa subárvore, mas todos os conjuntos de alterações que já tocaram nesses caminhos. Isso está mais próximo do conceito de clone do Git, mas se você estiver trabalhando em um projeto com uma longa história, pode demorar um pouco.

O sinalizador --detect-branches diz ao git-p4 para usar as especificações de ramificação do Perforce para mapear as ramificações para as referências do Git. Se esses mapeamentos não estiverem presentes no servidor Perforce (que é uma maneira perfeitamente válida de usar o Perforce), você pode informar ao git-p4 quais são os mapeamentos de ramificação e obter o mesmo resultado:

```
$ git init project
Inicializado repositório Git vazio em /tmp/project/.git/ $ cd
project $ git config git-p4.branchList main:dev $ git clone --
detect-branches //depot/project@all .
```

Definir a variável de configuração git-p4.branchList como main:dev informa ao git-p4 que “main” e “dev” são ramificações, e a segunda é filha da primeira.

Se agora git checkout -b dev p4/project/dev e fizermos alguns commits, git-p4 é inteligente o suficiente para direcionar o branch certo quando fizermos git p4 submit. Infelizmente, o git-p4 não pode misturar clones superficiais e ramificações múltiplas; se você tem um projeto enorme e quer trabalhar em mais de um branch, você terá que git p4 clone uma vez para cada branch que deseja enviar.

Para criar ou integrar filiais, você terá que usar um cliente Perforce. Git-p4 só pode sincronizar e enviar para branches existentes, e só pode fazer um changeset linear por vez. Se você mesclar duas ramificações no Git e tentar enviar o novo conjunto de alterações, tudo o que será registrado será um monte de alterações de arquivo; os metadados sobre quais branches estão envolvidos na integração serão perdidos.

Git e Perforce Resumo

Git-p4 torna possível usar um fluxo de trabalho Git com um servidor Perforce, e é muito bom nisso.
No entanto, é importante lembrar que o Perforce é responsável pela fonte e você está usando o Git apenas para trabalhar localmente. Apenas tome muito cuidado ao compartilhar commits do Git; se você tiver um controle remoto que outras pessoas usam, não envie nenhum commit que ainda não tenha sido enviado ao servidor Perforce.

Se você deseja misturar livremente o uso de Perforce e Git como clientes para controle de origem e pode convencer o administrador do servidor a instalá-lo, o Git Fusion torna o uso do Git um cliente de controle de versão de primeira classe para um servidor Perforce.

Machine Translated by Google

Migrando para o Git

Se você tem uma base de código existente em outro VCS, mas decidiu começar a usar o Git, deve migrar seu projeto de uma forma ou de outra. Esta seção aborda alguns importadores para sistemas comuns e, em seguida, demonstra como desenvolver seu próprio importador personalizado. Você aprenderá como importar dados de vários dos maiores sistemas SCM usados profissionalmente, porque eles constituem a maioria dos usuários que estão trocando e porque ferramentas de alta qualidade para eles são fáceis de encontrar.

Subversão

Se você leu a seção anterior sobre como usar git svn, você pode facilmente usar essas instruções para git svn clonar um repositório; então, pare de usar o servidor Subversion, envie para um novo servidor Git e comece a usá-lo. Se você quiser o histórico, pode fazer isso tão rápido quanto extrair os dados do servidor Subversion (o que pode demorar um pouco).

No entanto, a importação não é perfeita; e como vai demorar tanto, você também pode fazer direito. O primeiro problema é a informação do autor. No Subversion, cada pessoa que faz commit tem um usuário no sistema que é registrado nas informações de commit. Os exemplos na seção anterior mostram schacon em alguns lugares, como a saída de culpa e o git svn log. Se você quiser mapear isso para melhores dados de autor do Git, você precisa de um mapeamento dos usuários do Subversion para os autores do Git. Crie um arquivo chamado users.txt que tenha esse mapeamento em um formato como este:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Para obter uma lista dos nomes dos autores que o SVN usa, você pode executar isto:

```
$ svn log --xml --quiet | grep autor | ordenar -u | \ perl -pe
's/.*>(.*?)<.*/$1 = /'
```

Isso gera a saída de log no formato XML, mantém apenas as linhas com informações do autor, descarta duplicatas e remove as tags XML. Obviamente, isso só funciona em uma máquina com grep, sort e perl instalados. Em seguida, redirecione essa saída para o arquivo users.txt para que você possa adicionar os dados de usuário Git equivalentes ao lado de cada entrada.

ÿ

Se você estiver tentando isso em uma máquina Windows, este é o ponto em que você terá problemas. A Microsoft forneceu alguns bons conselhos e exemplos em https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from svn-to-git.

Você pode fornecer este arquivo para git svn para ajudá-lo a mapear os dados do autor com mais precisão. Você também pode dizer ao git svn para não incluir os metadados que o Subversion normalmente importa, passando --no-metadata para o comando clone ou init . Os metadados incluem um git-svn-id dentro de cada mensagem de commit que o Git irá gerar durante a importação. Isso pode sobrecarregar seu log do Git e torná-lo um pouco confuso.

ÿ Você precisa manter os metadados quando quiser espelhar os commits feitos no Git

Machine Translated by Google

> repositório de volta para o repositório SVN original. Se você não deseja a sincronização em seu log de confirmação, sinta-se à vontade para omitir o parâmetro --no-metadata .

Isso faz com que seu comando de importação fique assim:

```
$ git svn clone http://my-project.googlecode.com/svn/ \ --authors-  ""
        file=users.txt --no-metadata --prefix $ cd my_project          -s meu_projeto
```

Agora você deve ter uma importação Subversion mais agradável em seu diretório my_project . Em vez de commits que se parecem com isso:

```
cometer 37efa680e8473b615de980fa935944215428a35a
Autor: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Data: Dom 3 de maio 00:12:22 2009 +0000

instalação fixa - vá para o tronco

git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-be05-5f7a86268029
```

eles se parecem com isso:

```
cometer 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Autor: Scott Chacon <schacon@geemail.com>
Data: Dom 3 de maio 00:12:22 2009 +0000

instalação fixa - vá para o tronco
```

Não apenas o campo Autor parece muito melhor, mas o git-svn-id também não está mais lá.

Você também deve fazer um pouco de limpeza pós-importação. Por um lado, você deve limpar as referências estranhas que o git svn configurou. Primeiro, você moverá as tags para que sejam tags reais, em vez de ramificações remotas estranhas e, em seguida, moverá o restante das ramificações para que sejam locais.

Para mover as tags para serem tags Git apropriadas, execute:

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git tag ${t/tags\/} $t && git
branch -D -r $t; feito
```

Isso pega as referências que eram ramificações remotas que começaram com refs/remotes/tags/ e as torna tags reais (leves).

Em seguida, mova o restante das referências em refs/remotes para serem ramificações locais:

Machine Translated by Google

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); git branch $b refs/
remotes/$b && git branch -D -r $b; feito
```

Pode acontecer que você veja algumas ramificações extras que são sufixadas por @xxx (onde xxx é um número), enquanto no Subversion você vê apenas uma ramificação. Na verdade, esse é um recurso do Subversion chamado “peg-revisions”, que é algo para o qual o Git simplesmente não tem contraparte sintática. Portanto, git svn simplesmente adiciona o número da versão svn ao nome do ramo exatamente da mesma forma que você o escreveria no svn para endereçar a peg-revision daquele ramo. Se você não se importa mais com as revisões peg, simplesmente remova-as:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); faça git branch -D $p; feito
```

Agora todos os branches antigos são branches reais do Git e todas as tags antigas são tags reais do Git.

Há uma última coisa para limpar. Infelizmente, o git svn cria uma ramificação extra chamada trunk, que mapeia para a ramificação padrão do Subversion, mas a referência trunk aponta para o mesmo lugar que master.
Como master é mais idiomáticamente Git, veja como remover o branch extra:

```
$ git ramo -d tronco
```

A última coisa a fazer é adicionar seu novo servidor Git como remoto e enviar para ele. Aqui está um exemplo de como adicionar seu servidor como remoto:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Como você deseja que todos os seus branches e tags subam, agora você pode executar isto:

```
$ git push origin --all $
git push origin --tags
```

Todas as suas ramificações e tags devem estar em seu novo servidor Git em uma importação limpa e agradável.

Mercurial

Como o Mercurial e o Git têm modelos bastante semelhantes para representar versões, e como o Git é um pouco mais flexível, converter um repositório do Mercurial para o Git é bastante simples, usando uma ferramenta chamada "hg-fast-export", da qual você precisará de um cópia de:

```
$ git clone https://github.com/frej/fast-export.git
```

A primeira etapa da conversão é obter um clone completo do repositório Mercurial que você deseja converter:

Machine Translated by Google

```
$ hg clone <URL do repositório remoto> /tmp/hg-repo
```

A próxima etapa é criar um arquivo de mapeamento de autor. O Mercurial é um pouco mais indulgente do que o Git para o que ele colocará no campo author para changeset, então este é um bom momento para limpar a casa. Gerar isso é um comando de uma linha em um shell bash :

```
$ cd /tmp/hg-repo $
hg log | grep usuário: | classificar | único | sed 's/usuário: *//' > ../autores
```

Isso levará alguns segundos, dependendo de quanto tempo é o histórico do seu projeto, e depois o arquivo /tmp/authors ficará mais ou menos assim:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> empresa <DOT> com>
Bob Jones <bob@company.com> Joe Smith
<joe@company.com>
```

Neste exemplo, a mesma pessoa (Bob) criou conjuntos de alterações com quatro nomes diferentes, um dos quais realmente parece correto e um dos quais seria completamente inválido para um commit do Git. Hg fast-export nos permite corrigir isso transformando cada linha em uma regra: "<input>"="<output>", mapeando uma <input> para uma <output>. Dentro das strings <input> e <output> , todas as sequências de escape compreendidas pela codificação python string_escape são suportadas. Se o arquivo de mapeamento do autor não contiver um <input> correspondente, esse autor será enviado para o Git sem modificações. Se todos os nomes de usuário estiverem corretos, não precisaremos desse arquivo. Neste exemplo, queremos que nosso arquivo fique assim:

```
"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> empresa <DOT> com>"="Bob Jones <bob@company.com>"
```

O mesmo tipo de arquivo de mapeamento pode ser usado para renomear branches e tags quando o nome Mercurial não é permitido pelo Git.

A próxima etapa é criar nosso novo repositório Git e executar o script de exportação:

```
$ git init /tmp/converted $ cd /
tmp/converted $ /tmp/fast-export/
hg-fast-export.sh -r /tmp/hg-repo -A /tmp/autores
```

O sinalizador -r diz ao hg-fast-export onde encontrar o repositório Mercurial que queremos converter, e o sinalizador -A informa onde encontrar o arquivo de mapeamento de autor (arquivos de mapeamento de branch e tag são especificados por -B e - sinalizadores T respectivamente). O script analisa os conjuntos de alterações do Mercurial e os converte em um

Machine Translated by Google

script para o recurso de "importação rápida" do Git (que discutiremos em detalhes mais adiante). Isso demora um pouco (embora seja *muito* mais rápido do que seria na rede) e a saída é bastante detalhada:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors Carregado 4 autores master:
Exportando revisão completa 1/22208 com 13/0/0 adicionada/alterada/ arquivos removidos master:
Exportando revisão delta simples 2/22208 com 1/1/0 arquivos adicionados/alterados/removidos master: Exportando
revisão delta simples 3/22208 com arquivos 0/1/0 adicionados/alterados/removidos […] master: Exportando revisão delta simples
22206/22208 com 0/4/0 arquivos adicionados/alterados/removidos




master: Exportando revisão delta simples 22207/22208 com arquivos 0/2/0 adicionados/alterados/removidos master:
Exportando revisão delta completa 22208/22208 com arquivos 3/213/0 adicionados/alterados/removidos Tag de exportação
[0.4c] em [ hg r9] [git:10]



Exportando tag [0.4d] em [hg r16] [git :17] […]


Exportando tag [3.1-rc] em [hg r21926] [git:21927]
Exportando tag [3.1] em [hg r21973] [git:21974]
Comandos 22315 emitidos
estatísticas git-fast-import:
---------------------------------------------------------------------
Objetos alocados:          120000
Total de objetos:          115032     208171 duplicados                                  )
        blobs:               (40504 (     205320 duplicados      26117 deltas de          39602
tentativas)
        árvores:             52320 (        2851 duplicatas      47467 deltas de          47599
tentativas)
        comete:              22208 (           0 duplicatas          0 deltas de              0
tentativas)
        tags:                    0 (           0 duplicatas          0 deltas de              0
tentativas)
Filiais totais:                109           2 cargas
        marcas:            (1048576     22208 único        ) )
        átomos:              (1952
Total de memória:            7860 KiB
         pools:              2235 KiB
       objetos:              5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize() pack_report:                    =          4096
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 pack_report: pack_mapped
                                                           =
                                                           =          1 /          1
                                                           = 340852700 / 340852700
---------------------------------------------------------------------


$ git shortlog -sn
```

Machine Translated by Google

```
369 Bob Jones
365 Joe Smith
```

Isso é praticamente tudo que existe para isso. Todas as tags Mercurial foram convertidas em tags Git, e as ramificações e marcadores do Mercurial foram convertidos em ramificações Git. Agora você está pronto para enviar o repositório para sua nova página inicial do lado do servidor:

```
$ git remote add origin git@my-git-server:myrepository.git $ git
push origin --all
```

Bazar

O Bazaar é uma ferramenta DVCS muito parecida com o Git e, como resultado, é bastante simples converter um repositório Bazaar em um repositório Git. Para fazer isso, você precisará importar o plugin bzr-fastimport .

Obtendo o plug-in bzr-fastimport

O procedimento para instalar o plug-in fastimport é diferente em sistemas operacionais semelhantes ao UNIX e no Windows. No primeiro caso, o mais simples é instalar o pacote bzr-fastimport que irá instalar todas as dependências necessárias.

Por exemplo, com Debian e derivados, você faria o seguinte:

```
$ sudo apt-get install bzr-fastimport
```

Com o RHEL, você faria o seguinte:

```
$ sudo yum install bzr-fastimport
```

Com o Fedora, desde a versão 22, o novo gerenciador de pacotes é o dnf:

```
$ sudo dnf install bzr-fastimport
```

Se o pacote não estiver disponível, você pode instalá-lo como um plugin:

```
$ mkdir --parents ~/.bazaar/plugins              # cria as pastas necessárias para o
plugins $ cd ~/.bazaar/plugins $ bzr
branch lp:bzr-fastimport fastimport #
importa o plugin fastimport $ cd fastimport $ sudo python setup.py install --
record=files. txt # instala o plug-in
```

Machine Translated by Google

Para que este plug-in funcione, você também precisará do módulo Python fastimport . Você pode verificar se está presente ou não e instalá-lo com os seguintes comandos:

```
$ python -c "importação rápida"
Traceback (última chamada mais recente):
Arquivo "<string>", linha 1, em <module>
ImportError: Nenhum módulo chamado fastimport
$ pip install fastimport
```

Se não estiver disponível, você pode baixá-lo no endereço https://pypi.python.org/pypi/fastimport/.

No segundo caso (no Windows), o bzr-fastimport é instalado automaticamente com a versão autônoma e a instalação padrão (deixe todas as caixas de seleção marcadas). Então, neste caso, você não tem nada para fazer.

Neste ponto, a forma de importar um repositório Bazaar difere conforme você tenha uma única ramificação ou esteja trabalhando com um repositório que possui várias ramificações.

Projeto com uma única ramificação

Agora faça o cd no diretório que contém seu repositório Bazaar e inicialize o repositório Git:

```
$ cd /caminho/para/o/bzr/repositório $
git init
```

Agora, você pode simplesmente exportar seu repositório Bazaar e convertê-lo em um repositório Git usando o seguinte comando:

```
$ bzr fast-export --plain .                | git fast-import
```

Dependendo do tamanho do projeto, seu repositório Git é construído em um intervalo de alguns segundos a alguns minutos.

Caso de um projeto com uma ramificação principal e uma ramificação de trabalho

Você também pode importar um repositório do Bazaar que contenha ramificações. Vamos supor que você tenha duas ramificações: uma representa a ramificação principal (meuprojeto.trunk), a outra é a ramificação de trabalho (meuprojeto.trabalho).

```
$ ls
meuProjeto.trunk meuProjeto.trabalho
```

Crie o repositório Git e faça o cd nele:

```
$ git init git-repo
```

Machine Translated by Google

```
$ cd git-repo
```

Puxe o branch master para o git:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject.trunk | \ git fast-
import --export-marks=../marks.git
```

Puxe o ramo de trabalho para o Git:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \ git fast-
import --import-marks=../marks.git --export-marks=../marks.git
```

Agora o git branch mostra o branch master assim como o branch work . Verifique os logs para certificar-se de que estão completos e elimine os arquivos marks.bzr e marks.git .

Sincronizando a área de preparação

Qualquer que seja o número de filiais que você tenha e o método de importação usado, sua área de teste não é sincronizada com o HEAD e, com a importação de várias ramificações, seu diretório de trabalho também não é sincronizado. Esta situação é facilmente resolvida pelo seguinte comando:

```
$ git reset --hard HEAD
```

Ignorando os arquivos que foram ignorados com .bzrignore

Agora vamos dar uma olhada nos arquivos a serem ignorados. A primeira coisa a fazer é renomear .bzrignore para .gitignore. Se o arquivo .bzrignore contiver uma ou várias linhas começando com "!!" ou "RE:", você terá que modificá-lo e talvez criar vários arquivos .gitignore para ignorar exatamente os mesmos arquivos que o Bazaar estava ignorando.

Finalmente, você terá que criar um commit que contenha esta modificação para a migração:

```
$ git mv .bzrignore .gitignore $
# modifique .gitignore se
necessário $ git commit -am 'Migração do Bazaar para o Git'
```

Enviando seu repositório para o servidor

Aqui estamos! Agora você pode enviar o repositório para seu novo servidor inicial:

```
$ git remote add origin git@my-git-server:mygitrepository.git $ git
push origin --all $ git push origin --tags
```

Machine Translated by Google

Seu repositório Git está pronto para uso.

Forçosamente

O próximo sistema do qual você verá a importação é o Perforce. Como discutimos acima, há duas maneiras de permitir que o Git e o Perforce conversem entre si: git-p4 e Perforce Git Fusion.

Perforce Git Fusion

O Git Fusion torna esse processo bastante indolor. Basta definir as configurações do projeto, mapeamentos de usuários e ramificações usando um arquivo de configuração (conforme discutido no Git Fusion) e clonar o repositório. O Git Fusion deixa você com o que parece ser um repositório Git nativo, que está pronto para ser enviado para um host Git nativo, se desejar. Você pode até usar o Perforce como seu host Git, se quiser.

Git-p4

O Git-p4 também pode atuar como uma ferramenta de importação. Como exemplo, importaremos o projeto Jam do Perforce Public Depot. Para configurar seu cliente, você deve exportar a variável de ambiente P4PORT para apontar para o depósito Perforce:

```
$ export P4PORT=public.perforce.com:1666
```

ÿ

Para acompanhar, você precisará de um depósito Perforce para se conectar. Usaremos o depósito público em public.perforce.com para nossos exemplos, mas você pode usar qualquer depósito ao qual tenha acesso.

Execute o comando git p4 clone para importar o projeto Jam do servidor Perforce, fornecendo o depósito e o caminho do projeto e o caminho para o qual você deseja importar o projeto:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import Importando
de //guest/perforce_software/jam@all para p4import Repositório Git vazio
inicializado em /private/tmp/p4import/.git/ Destino da importação: refs/remotes/
p4/master Importando revisão 9957 (100%)
```

Este projeto específico tem apenas uma ramificação, mas se você tiver ramificações configuradas com exibições de ramificação (ou apenas um conjunto de diretórios), poderá usar o sinalizador --detect-branches para git p4 clone para importar todas as ramificações do projeto também . Consulte Ramificação para um pouco mais de detalhes sobre isso.

Neste ponto, você está quase pronto. Se você for ao diretório p4import e executar git log, poderá ver seu trabalho importado:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Autor: giles <giles@giles@perforce.com> Data:
quarta-feira, 8 de fevereiro 03:13:27 2012 -0800
```

Machine Translated by Google

```
Correção na linha 355; mude </UL> para </OL>.

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

cometer aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Autor: kwirth <kwirth@perforce.com> Data:
Terça-feira, 7 de julho 01:35:51 2009 -0800

Corrija o erro de ortografia na página do documento Jam (cumulativo -> cumulativo).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Você pode ver que git-p4 deixou um identificador em cada mensagem de commit. Não há problema em manter esse identificador lá, caso você precise fazer referência ao número de alteração do Perforce posteriormente. No entanto, se você quiser remover o identificador, agora é a hora de fazê-lo – antes de começar a trabalhar no novo repositório.
Você pode usar git filter-branch para remover as strings do identificador em massa:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Reescrever e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' foi reescrito
```

Se você executar git log, poderá ver que todas as somas de verificação SHA-1 para os commits foram alteradas, mas as strings git-p4 não estão mais nas mensagens de commit:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Autor: giles <giles@giles@perforce.com> Data:
quarta-feira, 8 de fevereiro 03:13:27 2012 -0800

Correção na linha 355; mude </UL> para </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Autor: kwirth <kwirth@perforce.com> Data: Ter, 7 de julho
01:35:51 2009 -0800

Corrija o erro de ortografia na página do documento Jam (cumulativo -> cumulativo).
```

Sua importação está pronta para ser enviada para seu novo servidor Git.

Um importador personalizado

Se o seu sistema não for um dos acima, você deve procurar um importador online – importadores de qualidade estão disponíveis para muitos outros sistemas, incluindo CVS, Clear Case, Visual Source Safe e até mesmo um diretório de arquivos. Se nenhuma dessas ferramentas funcionar para você, se você tiver uma ferramenta mais obscura ou precisar de um processo de importação mais personalizado, use git fast-import. Este comando lê instruções simples do stdin para gravar dados específicos do Git. É muito mais fácil criar objetos Git desta forma do que executar os comandos brutos do Git ou tentar escrever os objetos brutos (veja Git Internals para mais

Machine Translated by Google

Informação). Dessa forma, você pode escrever um script de importação que leia as informações necessárias do sistema do qual você está importando e imprima instruções diretas para stdout. Você pode então executar este programa e canalizar sua saída por meio do git fast-import.

Para demonstrar rapidamente, você escreverá um importador simples. Suponha que você trabalhe no atual, faça backup de seu projeto copiando ocasionalmente o diretório em um diretório de backup back_YYYY_MM_DD com registro de data e hora e deseja importar isso para o Git. Sua estrutura de diretórios se parece com isso:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03 atual
```

Para importar um diretório Git, você precisa revisar como o Git armazena seus dados. Como você deve se lembrar, o Git é fundamentalmente uma lista encadeada de objetos commit que apontam para um instantâneo do conteúdo. Tudo o que você precisa fazer é informar à importação rápida quais são os instantâneos de conteúdo, quais são os pontos de dados de commit para eles e a ordem em que eles aparecem. Sua estratégia será passar pelos instantâneos um de cada vez e criar commits com o conteúdo de cada diretório, ligando cada commit de volta ao anterior.

Como fizemos em Um exemplo de política aplicada pelo Git, escreveremos isso em Ruby, porque é com o que geralmente trabalhamos e tende a ser fácil de ler. Você pode escrever este exemplo facilmente em qualquer coisa com a qual esteja familiarizado - ele só precisa imprimir as informações apropriadas no stdout. E, se você estiver executando no Windows, isso significa que você precisará tomar cuidado especial para não introduzir retornos de carro no final de suas linhas - git fast-import é muito específico sobre querer apenas feeds de linha (LF) e não a linha de retorno de carro feeds (CRLF) que o Windows usa.

Para começar, você mudará para o diretório de destino e identificará cada subdiretório, cada um dos quais é um instantâneo que você deseja importar como um commit. Você mudará para cada subdiretório e imprimirá os comandos necessários para exportá-lo. Seu loop principal básico se parece com isso:

```
última_marca = nil

# percorre os diretórios
Dir.chdir(ARGV[0]) do
Dir.glob("*").each do |dir| próximo se
Arquivo.arquivo?(dir)

# move para o diretório de destino
Dir.chdir(dir) do last_mark = print_export(dir,
        last_mark) end end end
```

Você executa print_export dentro de cada diretório, que pega o manifesto e a marca do instantâneo anterior e retorna o manifesto e a marca deste; dessa forma, você pode vinculá-los corretamente.

Machine Translated by Google

"Mark" é o termo de importação rápida para um identificador que você dá a um commit; ao criar commits, você dá a cada um uma marca que pode ser usada para vincular a ele a partir de outros commits. Assim, a primeira coisa a fazer em seu método print_export é gerar uma marca a partir do nome do diretório:

```
marca = convert_dir_to_mark(dir)
```

Você fará isso criando uma matriz de diretórios e usando o valor do índice como a marca, porque uma marca deve ser um número inteiro. Seu método se parece com isso:

```
$marks = []
def convert_dir_to_mark(dir) if !
$marks.include?(dir) $marks <<
dir end ($marks.index(dir) +
1).to_s end
```

Agora que você tem uma representação inteira do seu commit, você precisa de uma data para os metadados do commit. Como a data é expressa no nome do diretório, você a analisará. A próxima linha no seu arquivo print_export é:

```
data = convert_dir_to_date(dir)
```

onde convert_dir_to_date é definido como:

```
def convert_dir_to_date(dir) if
dir == 'atual' return Time.now().to_i
else

dir = dir.gsub('back_', '') (ano, mês,
dia) = dir.split('_') return Time.local(ano, mês,
dia).to_i end

fim
```

Isso retorna um valor inteiro para a data de cada diretório. A última meta-informação que você precisa para cada commit são os dados do committer, que você codifica em uma variável global:

```
$autor = 'John Doe <john@example.com>'
```

Agora você está pronto para começar a imprimir os dados de confirmação para o seu importador. As informações iniciais informam que você está definindo um objeto de confirmação e em qual ramificação ele está, seguido pela marca que você gerou, as informações do committer e a mensagem de commit e, em seguida, o commit anterior, se houver. O código fica assim:

Machine Translated by Google

```
# imprime as informações de
importação puts 'commit refs/heads/
master' puts 'mark :' + mark puts
"committer #{$author} #{date} -0700"
export_data('imported from + dir) puts 'from :' + última_marca
se última_marca
```

Você codifica o fuso horário (-0700) porque isso é fácil. Se estiver importando de outro sistema, você deve especificar o fuso horário como um deslocamento. A mensagem de confirmação deve ser expressa em um formato especial:

```
dados (tamanho)\n(conteúdo)
```

O formato consiste na palavra data, o tamanho dos dados a serem lidos, uma nova linha e, finalmente, os dados. Como você precisa usar o mesmo formato para especificar o conteúdo do arquivo posteriormente, crie um método auxiliar, export_data:

```
def export_data(string) print
"data #{string.size}\n#{string}" end
```

Tudo o que resta é especificar o conteúdo do arquivo para cada instantâneo. Isso é fácil, porque você tem cada um em um diretório – você pode imprimir o comando deleteall seguido pelo conteúdo de cada arquivo no diretório. O Git então gravará cada snapshot apropriadamente:

```
coloca 'excluir tudo'
Dir.glob("**/*").each do |arquivo| next if !
Arquivo.arquivo?(arquivo)
inline_data(arquivo) end
```

Nota: Como muitos sistemas consideram suas revisões como alterações de um commit para outro, a importação rápida também pode receber comandos com cada commit para especificar quais arquivos foram adicionados, removidos ou modificados e quais são os novos conteúdos. Você poderia calcular as diferenças entre os instantâneos e fornecer apenas esses dados, mas fazer isso é mais complexo – você também pode fornecer ao Git todos os dados e deixá-lo descobrir. Se isso for mais adequado para seus dados, verifique a página do manual de importação rápida para obter detalhes sobre como fornecer seus dados dessa maneira.

O formato para listar o novo conteúdo do arquivo ou especificar um arquivo modificado com o novo conteúdo é o seguinte:

```
M 644 caminho inline/para/dados de
arquivo (tamanho) (conteúdo do
arquivo)
```

Machine Translated by Google

Aqui, 644 é o modo (se você tiver arquivos executáveis, precisará detectar e especificar 755) e o inline diz que você listará o conteúdo imediatamente após esta linha. Seu método inline_data se parece com isto:

```
def inline_data(file, code = 'M', mode = '644') content = File.read(file)
puts "#{code} #{mode} inline #{file}" export_data(content) end
```

Você reutiliza o método export_data definido anteriormente, porque é o mesmo que você especificou para os dados da mensagem de confirmação.

A última coisa que você precisa fazer é retornar a marca atual para que ela possa ser passada para a próxima iteração:

```
marca de retorno
```

ÿ

Se você estiver executando no Windows, precisará certificar-se de adicionar uma etapa extra. Como mencionado anteriormente, o Windows usa CRLF para novos caracteres de linha, enquanto git fast-import espera apenas LF. Para contornar esse problema e tornar o git fast-import feliz, você precisa dizer ao ruby para usar LF em vez de CRLF:

```
$stdout.binmode
```

É isso. Aqui está o roteiro na íntegra:

```
#!/usr/bin/env rubi

$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 == 'atual'


                return Time.now().to_i else dir =
dir.gsub('back_', '') (ano, mês, dia) = dir.split('_')
                return Time.local(ano, mês, dia). to_i
```

Machine Translated by Google

```
fim fim



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) data =
convert_dir_to_date(dir) marca = 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 :#{ última_marca}" se
última_marca



coloca 'excluir tudo'
Dir.glob("**/*").each do |arquivo| next if !Arquivo.arquivo?
                (arquivo) inline_data(arquivo)

marca final

fim


# Percorra os diretórios last_mark = nil
Dir.chdir(ARGV[0]) do Dir.glob("*").each do |
dir| próximo se Arquivo.arquivo?(dir)




                # move para o diretório de destino Dir.chdir(dir)
                do last_mark = print_export(dir, last_mark)

                fim
fim fim
```

Se você executar este script, obterá um conteúdo semelhante a este:

```
$ ruby import.rb /opt/import_from commit refs/heads/
master mark:1 committer John Doe
<john@example.com> 1388649600 -0700
```

Machine Translated by Google

```
data 29
imported from back_2014_01_02deleteall M 644 inline
README.md data 28 # Hello




Este é o meu leia-me.
commit refs/heads/master mark:2


committer John Doe <john@example.com> 1388822400 -0700 dados 29
importados de back_2014_01_04from:1 deleteall



M 644 em linha main.rb
dados 34
#!/bin/env rubi

coloca "Olá"
M 644 README.md embutido
(...)
```

Para executar o importador, canalize essa saída por meio do git fast-import enquanto estiver no diretório Git para o qual deseja importar. Você pode criar um novo diretório e, em seguida, executar git init nele como ponto de partida e, em seguida, executar seu script:

```
$ git init
Repositório Git vazio inicializado em /opt/import_to/.git/ $ ruby import.rb /opt/
import_from | git fast-import estatísticas git-fast-import:

---------------------------------------------------------------------------
Objetos alocados:       5000
Total de objetos:         13    6 duplicatas 4
        blobs:             ( 5 (    duplicatas      ) 3 deltas de      5
tentativas)
        árvores:           4 (    1 duplicatas      0 deltas de      4
tentativas)
        commits:           4 (    1 duplicatas      0 deltas de      0
tentativas) tags:
        tentativas)        0 (    0 duplicatas      0 deltas de      0

Total de filiais:          1    1 carrega
        marcas:         ( 1024    5 único      ) )
        átomos:            ( 2
Total de memória:       2344 KiB
          pools:        2110 KiB
        objetos:         234 KiB
---------------------------------------------------------------------------
pack_report: getpagesize() 4096 pack_report: core.packedGitWindowSize
= 1073741824 pack_report: core.packedGitLimit = 8589934592
```

Machine Translated by Google

```
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
```

Como você pode ver, quando ele é concluído com sucesso, ele fornece várias estatísticas sobre o que foi realizado. Neste caso, você importou 13 objetos no total para 4 commits em 1 branch. Agora, você pode executar git log para ver seu novo histórico:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Autor: John Doe <john@example.com> Data:
terça-feira, 29 de julho 19:39:04 2014 -0700

importado do atual

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Autor: John Doe <john@example.com> Data: segunda-feira,
3 de fevereiro 01:00:00 2014 -0700

importado de back_2014_02_03
```

Pronto, um repositório Git limpo e agradável. É importante observar que nada é verificado - você não tem nenhum arquivo em seu diretório de trabalho a princípio. Para obtê-los, você deve redefinir sua ramificação para onde o mestre está agora:

```
$ ls
$ git reset --hard master HEAD
está agora em 3caa046 importado do atual $ ls
README.md main.rb
```

Você pode fazer muito mais com a ferramenta de importação rápida – lidar com modos diferentes, dados binários, ramificações múltiplas e mesclagem, tags, indicadores de progresso e muito mais. Vários exemplos de cenários mais complexos estão disponíveis no diretório contrib/fast-import do código-fonte do Git.

Resumo

Você deve se sentir confortável usando o Git como um cliente para outros sistemas de controle de versão ou importando praticamente qualquer repositório existente para o Git sem perder dados. No próximo capítulo, abordaremos as partes internas brutas do Git para que você possa criar cada byte, se necessário.

Machine Translated by Google

Internos do Git

Você pode ter pulado para este capítulo de um capítulo muito anterior, ou pode ter chegado aqui depois de ler sequencialmente o livro inteiro até este ponto — em ambos os casos, é aqui que examinaremos o funcionamento interno e a implementação do Git. Descobrimos que entender essas informações era fundamentalmente importante para avaliar o quão útil e poderoso o Git é, mas outros argumentaram que pode ser confuso e desnecessariamente complexo para iniciantes. Portanto, fizemos dessa discussão o último capítulo do livro para que você possa lê-la no início ou no final de seu processo de aprendizado. Deixamos para você decidir.

Agora que você está aqui, vamos começar. Primeiro, se ainda não está claro, o Git é fundamentalmente um sistema de arquivos endereçável por conteúdo com uma interface de usuário VCS escrita sobre ele. Você aprenderá mais sobre o que isso significa daqui a pouco.

Nos primeiros dias do Git (principalmente antes da versão 1.5), a interface do usuário era muito mais complexa porque enfatizava esse sistema de arquivos em vez de um VCS polido. Nos últimos anos, a interface do usuário foi refinada até ficar tão limpa e fácil de usar quanto qualquer sistema existente; no entanto, o estereótipo persiste sobre a IU inicial do Git, que era complexa e difícil de aprender.

A camada do sistema de arquivos endereçável por conteúdo é incrivelmente legal, então vamos cobrir isso primeiro neste capítulo; em seguida, você aprenderá sobre os mecanismos de transporte e as tarefas de manutenção do repositório com as quais eventualmente terá que lidar.

Encanador e Porcelanato

Este livro aborda principalmente como usar o Git com cerca de 30 subcomandos, como checkout, branch, remote e assim por diante. Mas como o Git era inicialmente um kit de ferramentas para um sistema de controle de versão em vez de um VCS totalmente amigável, ele possui vários subcomandos que executam trabalhos de baixo nível e foram projetados para serem encadeados no estilo UNIX ou chamados de scripts. Esses comandos são geralmente referidos como comandos de "encanamento" do Git, enquanto os comandos mais amigáveis são chamados de comandos de "porcelana".

Como você já deve ter notado, os primeiros nove capítulos deste livro tratam quase exclusivamente de comandos de porcelana. Mas neste capítulo, você lidará principalmente com os comandos de nível inferior, porque eles fornecem acesso ao funcionamento interno do Git e ajudam a demonstrar como e por que o Git faz o que faz. Muitos desses comandos não devem ser usados manualmente na linha de comando, mas sim como blocos de construção para novas ferramentas e scripts personalizados.

Quando você executa git init em um diretório novo ou existente, o Git cria o diretório .git , que é onde está localizado quase tudo o que o Git armazena e manipula. Se você deseja fazer backup ou clonar seu repositório, copiar esse único diretório em outro lugar fornece quase tudo o que você precisa. Todo este capítulo trata basicamente do que você pode ver neste diretório. Aqui está a aparência de um diretório .git recém-inicializado:

```
$ ls -F1
descrição da
configuração
CABEÇA
```

Machine Translated by Google

```
ganchos/
info/ objetos/ refs/
```

Dependendo da sua versão do Git, você pode ver algum conteúdo adicional lá, mas este é um novo repositório git init — é o que você vê por padrão. O arquivo de descrição é usado apenas pelo programa GitWeb, então não se preocupe com isso. O arquivo de configuração contém as opções de configuração específicas do projeto e o diretório info mantém um arquivo de exclusão global para padrões ignorados que você não deseja rastrear em um arquivo .gitignore . O diretório hooks contém seus scripts de hook do lado do cliente ou do servidor, que são discutidos em detalhes em Git Hooks.

Isso deixa quatro entradas importantes: o HEAD e os arquivos de índice (ainda a serem criados) , e os diretórios de objetos e refs . Estas são as partes principais do Git. O diretório de objetos armazena todo o conteúdo do seu banco de dados, o diretório refs armazena ponteiros em objetos de confirmação nesses dados (branches, tags, remotos e mais), o arquivo HEAD aponta para o branch que você fez check-out no momento e o arquivo de índice é onde o Git armazena as informações da sua área de preparação. Agora você examinará cada uma dessas seções em detalhes para ver como o Git opera.

Objetos Git

Git é um sistema de arquivos endereçável por conteúdo. Ótimo. O que isso significa? Isso significa que no núcleo do Git está um armazenamento de dados chave-valor simples. O que isso significa é que você pode inserir qualquer tipo de conteúdo em um repositório Git, para o qual o Git lhe devolverá uma chave exclusiva que você poderá usar posteriormente para recuperá-lo contente.

Como demonstração, vejamos o comando de encanamento git hash-object, que pega alguns dados, armazena-os em seu diretório .git/objects (o *banco de dados do objeto)* e devolve a chave exclusiva que agora se refere a esse objeto de dados.

Primeiro, você inicializa um novo repositório Git e verifica se não há (previsivelmente) nada no diretório de objetos :

```
$ git init test
Inicializado repositório Git vazio em /tmp/test/.git/ $ cd test $
find .git/objects .git/objects .git/objects/info .git/objects/pack
$ find .git/objects -type f
```

O Git inicializou o diretório de objetos e criou os subdiretórios pack e info nele, mas não há arquivos regulares. Agora, vamos usar git hash-object para criar um novo objeto de dados e armazená-lo manualmente em seu novo banco de dados Git:

```
$ echo 'teste de conteúdo' | git hash-object -w --stdin
```

Machine Translated by Google

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Em sua forma mais simples, git hash-object pegaria o conteúdo que você entregou a ele e apenas retornaria a chave exclusiva que *seria* usada para armazená-lo em seu banco de dados Git. A opção -w informa ao comando não apenas para retornar a chave, mas para gravar esse objeto no banco de dados. Finalmente, a opção --stdin diz ao git hash-object para obter o conteúdo a ser processado do stdin; caso contrário, o comando esperaria um argumento de nome de arquivo no final do comando contendo o conteúdo a ser usado.

A saída do comando acima é um hash de soma de verificação de 40 caracteres. Este é o hash SHA-1 — uma soma de verificação do conteúdo que você está armazenando mais um cabeçalho, sobre o qual você aprenderá daqui a pouco. Agora você pode ver como o Git armazenou seus dados:

```
$ find .git/objects -type f .git/
objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Se você examinar novamente seu diretório de objetos , verá que agora ele contém um arquivo para esse novo conteúdo. É assim que o Git armazena o conteúdo inicialmente — como um único arquivo por parte do conteúdo, nomeado com a soma de verificação SHA-1 do conteúdo e seu cabeçalho. O subdiretório é nomeado com os 2 primeiros caracteres do SHA-1 e o nome do arquivo são os 38 caracteres restantes.

Depois de ter o conteúdo em seu banco de dados de objetos, você pode examinar esse conteúdo com o comando git cat-file . Este comando é uma espécie de canivete suíço para inspecionar objetos Git. Passar -p para o arquivo cat instrui o comando a primeiro descobrir o tipo de conteúdo e, em seguida, exibi-lo adequadamente:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
conteúdo de teste
```

Agora, você pode adicionar conteúdo ao Git e retirá-lo novamente. Você também pode fazer isso com conteúdo em arquivos. Por exemplo, você pode fazer algum controle de versão simples em um arquivo. Primeiro, crie um novo arquivo e salve seu conteúdo em seu banco de dados:

```
$ echo 'versão 1' > test.txt $ git
hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Em seguida, escreva algum novo conteúdo no arquivo e salve-o novamente:

```
$ echo 'versão 2' > test.txt $ git
hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Seu banco de dados de objetos agora contém ambas as versões deste novo arquivo (assim como o primeiro conteúdo que você armazenou lá):

Machine Translated by Google

```
$ find .git/objects -type f .git/
objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a .git/
objects/83/baae61804e65cc73a7201a7252750c76066a30 .git/
objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Neste ponto, você pode excluir sua cópia local desse arquivo test.txt e, em seguida, usar o Git para recuperar, do banco de dados de objetos, a primeira versão que você salvou:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt $ cat
test.txt versão 1
```

ou a segunda versão:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt $ cat
test.txt versão 2
```

Mas lembrar a chave SHA-1 para cada versão do seu arquivo não é prático; além disso, você não está armazenando o nome do arquivo em seu sistema - apenas o conteúdo. Esse tipo de objeto é chamado de *blob.* Você pode fazer com que o Git informe o tipo de objeto de qualquer objeto no Git, dada sua chave SHA-1, com git cat-file -t:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob
```

Objetos de árvore

O próximo tipo de objeto Git que examinaremos é a *árvore,* que resolve o problema de armazenar o nome do arquivo e também permite armazenar um grupo de arquivos juntos. O Git armazena conteúdo de maneira semelhante a um sistema de arquivos UNIX, mas um pouco simplificado. Todo o conteúdo é armazenado como objetos de árvore e blob, com árvores correspondendo a entradas de diretório UNIX e blobs correspondendo mais ou menos a inodes ou conteúdo de arquivo. Um único objeto de árvore contém uma ou mais entradas, cada uma das quais é o hash SHA-1 de um blob ou subárvore com seu modo, tipo e nome de arquivo associados. Por exemplo, digamos que você tenha um projeto em que a árvore mais recente se parece com:

```
$ git cat-file -p master^{tree} 100644
blob a906cb2a4a904a152e80877d4088654daad0c859 100644    Leia-me
blob 8f94139338f9404f26296befa88755fc2598c289            Rakefile
040000 árvore 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0   lib
```

A sintaxe master^{tree} especifica o objeto tree que é apontado pelo último commit em seu branch master . Observe que o subdiretório lib não é um blob, mas um ponteiro para outra árvore:

Machine Translated by Google

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b simplegit.rb
```

ÿ

Dependendo de qual shell você usa, você pode encontrar erros ao usar a sintaxe master^{tree} .

No CMD no Windows, o caractere ^ é usado para escapar, então você deve dobrá-lo para evitar isso: git cat-file -p master^^{tree}. Ao usar o PowerShell, os parâmetros que usam caracteres {} devem ser citados para evitar que o parâmetro seja analisado incorretamente: git cat-file -p 'master^{tree}'.

Se você estiver usando ZSH, o caractere ^ é usado para globbing, então você deve colocar toda a expressão entre aspas: git cat-file -p "master^{tree}".

Conceitualmente, os dados que o Git está armazenando são mais ou menos assim:

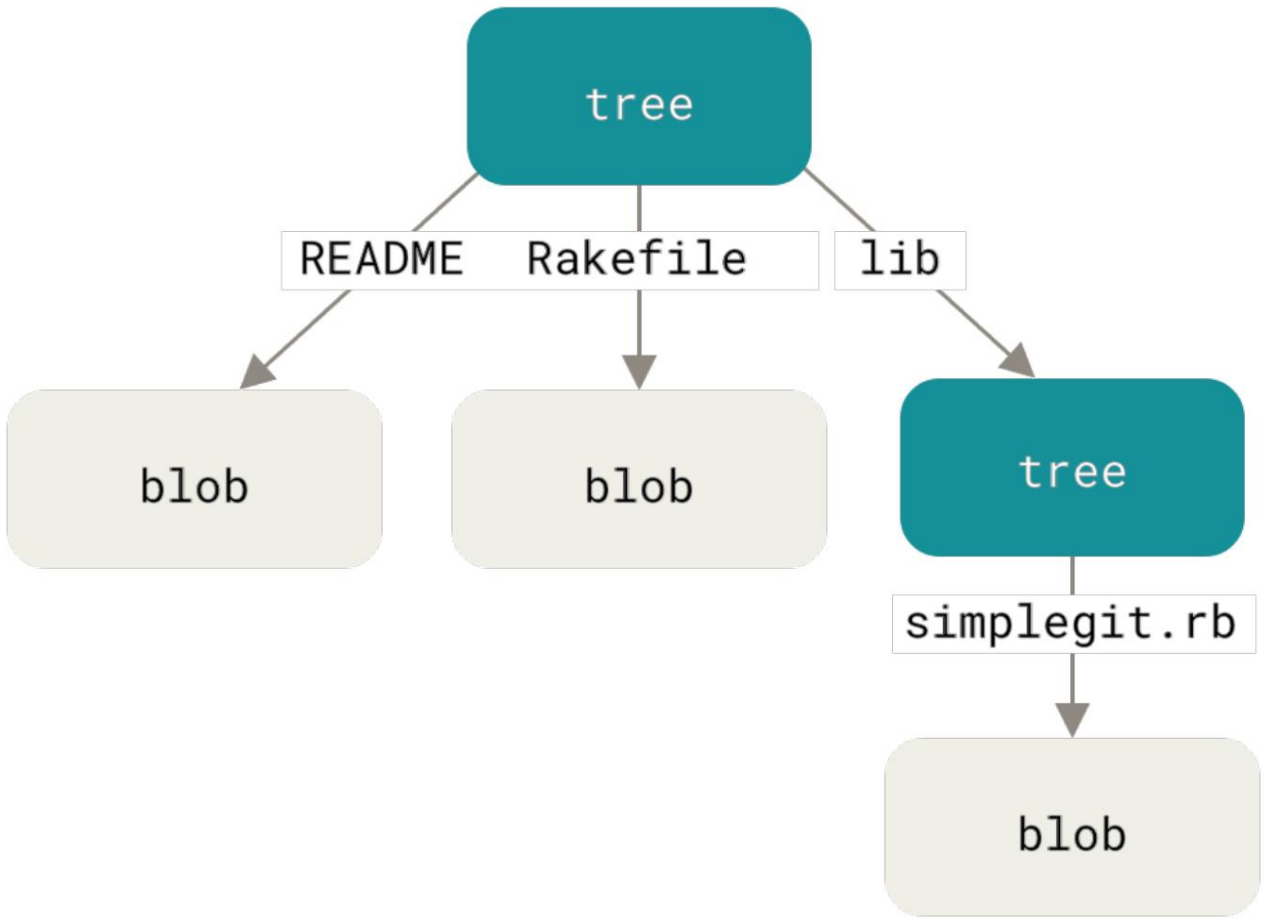


Figura 147. Versão simples do modelo de dados do Git

Você pode facilmente criar sua própria árvore. O Git normalmente cria uma árvore pegando o estado de sua área de teste ou índice e escrevendo uma série de objetos de árvore a partir dele. Portanto, para criar um objeto de árvore, primeiro você precisa configurar um índice preparando alguns arquivos. Para criar um índice com uma única entrada — a primeira versão do seu arquivo test.txt — você pode usar o comando de encanamento git update-index. Você usa esse comando para adicionar artificialmente a versão anterior do arquivo test.txt a uma nova área de preparação. Você deve passar a opção --add porque o arquivo ainda não existe em sua área de staging (você nem mesmo tem um

Machine Translated by Google

área de preparação ainda configurada) e --cacheinfo porque o arquivo que você está adicionando não está em seu diretório, mas está em seu banco de dados. Em seguida, você especifica o modo, SHA-1 e o nome do arquivo:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Nesse caso, você está especificando um modo de 100644, o que significa que é um arquivo normal. Outras opções são 100755, o que significa que é um arquivo executável; e 120000, que especifica um link simbólico. O modo é retirado dos modos normais do UNIX, mas é muito menos flexível — esses três modos são os únicos válidos para arquivos (blobs) no Git (embora outros modos sejam usados para diretórios e submódulos).

Agora, você pode usar git write-tree para gravar a área de preparação em um objeto de árvore. Nenhuma opção -w é necessária — chamar este comando cria automaticamente um objeto de árvore do estado do índice se essa árvore ainda não existir:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a72607650c test.
```

Você também pode verificar se este é um objeto de árvore usando o mesmo comando git cat-file que você viu anteriormente:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
árvore
```

Agora você criará uma nova árvore com a segunda versão do test.txt e também um novo arquivo:

```
$ echo 'novo arquivo' > new.txt
$ git update-index --cacheinfo 100644 \
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt $
git update-index --add new.txt
```

Sua área de preparação agora tem a nova versão de test.txt , bem como o novo arquivo new.txt. Escreva essa árvore (registrando o estado da área de preparação ou índice para um objeto de árvore) e veja como ela se parece:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a        teste.txt
```

Observe que esta árvore possui entradas de arquivo e também que o SHA-1 do test.txt é o SHA-1 da “versão 2” de antes (1f7a7a). Apenas por diversão, você adicionará a primeira árvore como um subdiretório a este. Você pode

Machine Translated by Google

leia as árvores em sua área de preparação chamando git read-tree. Nesse caso, você pode ler uma árvore existente em sua área de preparação como uma subárvore usando a opção --prefix com este comando:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579 $ git
write-tree 3c4e9cd789d88d8d89c1073707c3585e41b0e614 $ git cat-file -p
3c4e9cd789d88d8d89c1073707c3585e41b0e614 040000 tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak 100644 blob
fa49b077972391ad58037050f2a75f74e3671e92 100644 blob
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a                    novo.txt
                                                            teste.txt
```

Se você criou um diretório de trabalho a partir da nova árvore que acabou de escrever, obteria os dois arquivos no nível superior do diretório de trabalho e um subdiretório chamado bak que continha a primeira versão do arquivo test.txt . Você pode pensar nos dados que o Git contém para essas estruturas como sendo assim:

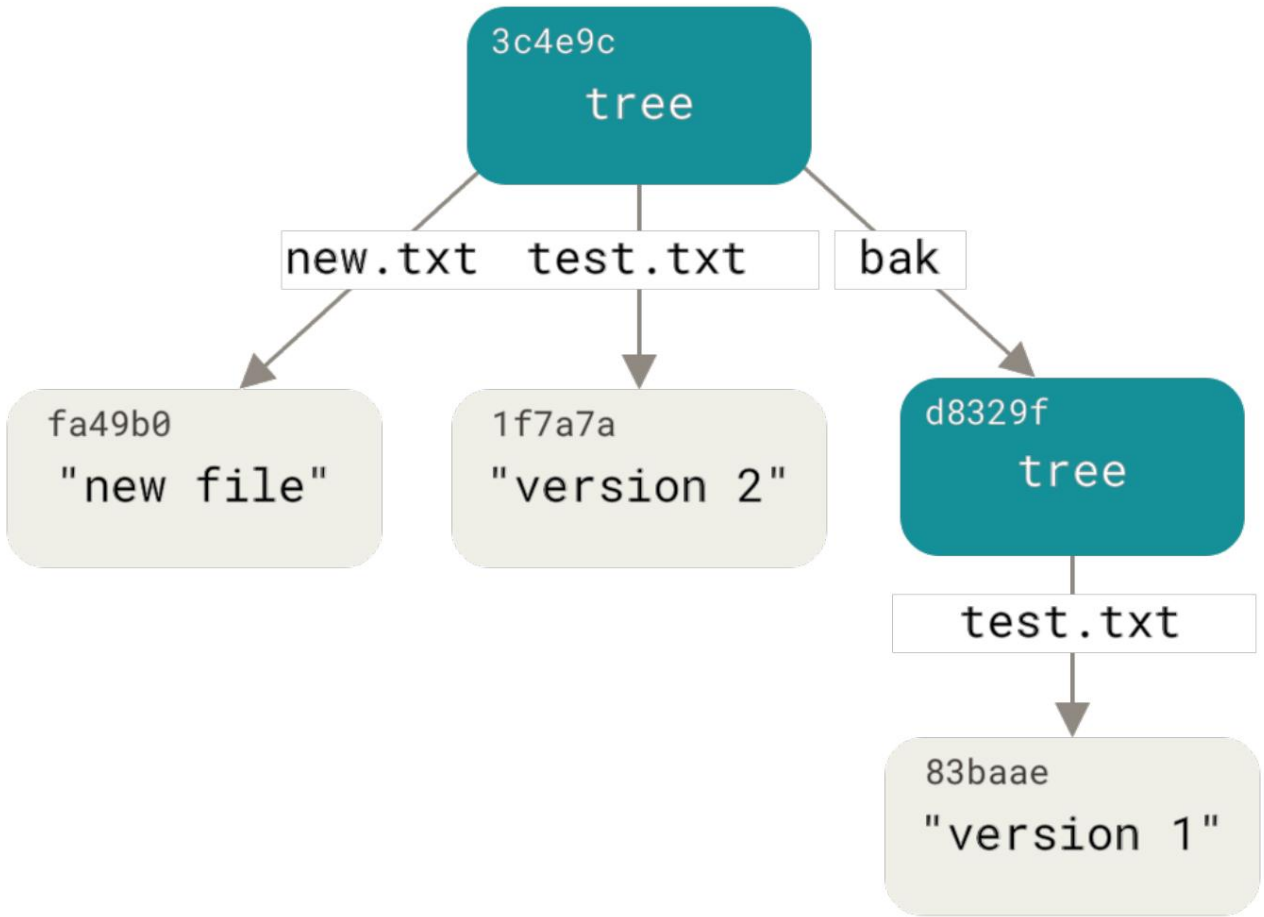


Figura 148. A estrutura de conteúdo de seus dados Git atuais

Confirmar objetos

Se você fez todos os itens acima, agora você tem três árvores que representam os diferentes instantâneos do seu projeto que deseja rastrear, mas o problema anterior permanece: você deve lembrar todos os três valores SHA-1 para recuperar os instantâneos . Você também não tem nenhuma informação sobre quem salvou os instantâneos, quando eles foram salvos ou por que foram salvos. Esta é a informação básica que o objeto commit armazena para você.

Machine Translated by Google

Para criar um objeto commit, você chama commit-tree e especifica uma única árvore SHA-1 e quais objetos commit, se houver, o precedem diretamente. Comece com a primeira árvore que você escreveu:

```
$ echo 'Primeiro commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

ÿ

Você obterá um valor de hash diferente devido ao tempo de criação e aos dados do autor diferentes. Além disso, enquanto em princípio qualquer objeto de commit pode ser reproduzido com precisão dados esses dados, os detalhes históricos da construção deste livro significam que os hashes de commit impressos podem não corresponder aos commits fornecidos. Substitua hashes de commit e tag por suas próprias somas de verificação mais adiante neste capítulo.

Agora você pode ver seu novo objeto commit com git cat-file:

```
$ git cat-file -p fdf4fc3 tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 autor Scott
Chacon <schacon@gmail.com> 1243040974 -0700 committer Scott Chacon
<schacon@gmail.com> 1243040974 -0700

Primeiro commit
```

O formato de um objeto commit é simples: ele especifica a árvore de nível superior para o instantâneo do projeto naquele ponto; o pai confirma se houver (o objeto commit descrito acima não tem nenhum pai); as informações do autor/committer (que usam suas definições de configuração user.name e user.email e um carimbo de data/hora); uma linha em branco e, em seguida, a mensagem de confirmação.

Em seguida, você escreverá os outros dois objetos commit, cada um referenciando o commit que veio diretamente antes dele:

```
$ echo 'Segundo commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d $ echo 'Terceiro commit' | git
commit-tree 3c4e9c -p cac0cab 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Cada um dos três objetos commit aponta para uma das três árvores de snapshot que você criou. Curiosamente, agora você tem um histórico real do Git que pode ser visualizado com o comando git log , se executá-lo no último commit SHA-1:

```
$ git log --stat 1a410e commit
1a410efbd13591db07496601ebc7a059dd55cfe9
Autor: Scott Chacon <schacon@gmail.com> Data:
sexta-feira, 22 de maio 18:15:24 2009 -0700

Terceiro commit

bak/teste.txt |          1 +
```

Machine Translated by Google

```
1 arquivo alterado, 1 inserção(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Autor: Scott Chacon <schacon@gmail.com> Data: Sex
22 de maio 18:14:29 2009 -0700

Segunda confirmação

novo.txt |        1 +
teste.txt | 2 +- 2
arquivos alterados, 2 inserções (+), 1 exclusão (-)

cometer fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Autor: Scott Chacon <schacon@gmail.com>
Data: sexta-feira, 22 de maio 18:09:34 2009 -0700

Primeiro commit

teste.txt | 1 + 1
arquivo alterado, 1 inserção(+)
```

Incrível. Você acabou de fazer as operações de baixo nível para criar um histórico do Git sem usar nenhum dos comandos de front-end. Isso é basicamente o que o Git faz quando você executa os comandos git add e git commit — ele armazena blobs para os arquivos que foram alterados, atualiza o índice, grava árvores e grava objetos de confirmação que fazem referência às árvores de nível superior e aos commits que veio imediatamente antes deles. Esses três objetos Git principais — o blob, a árvore e o commit — são inicialmente armazenados como arquivos separados em seu diretório .git/objects . Aqui estão todos os objetos no diretório de exemplo agora, comentados com o que eles armazenam:

```
$ find .git/objects -type f .git/
objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2 .git/objects/
1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3 .git/objects/1f/
7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2 .git/objects/3c /
4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3 .git/objects/83/
baae61804e65cc73a7201a7252750c76066a30 # test.txt v1 .git/objects/ca/
c0cab538b970a37ea1e769cbbde608743bc96d # commit 2 .git/objects/
d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content' .git/objects/
d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1 .git/objects/fa/
49b077972391ad58037050f2a75f74e3671e92 # new.txt .git/objects/fd/
f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Se você seguir todos os ponteiros internos, obterá um gráfico de objeto mais ou menos assim:

Machine Translated by Google

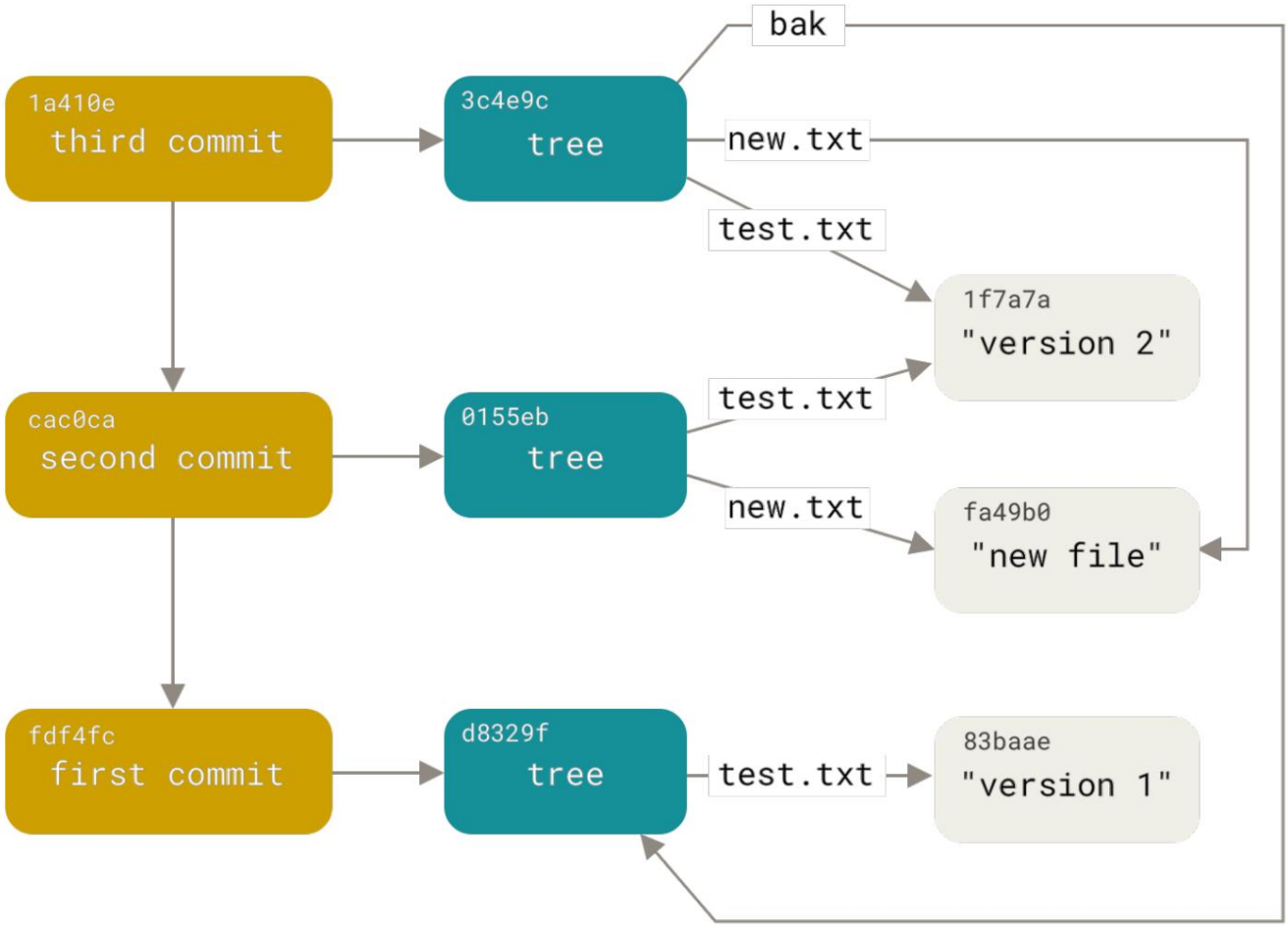


Figura 149. Todos os objetos acessíveis em seu diretório Git

Armazenamento de objetos

Mencionamos anteriormente que há um cabeçalho armazenado com cada objeto que você envia para seu banco de dados de objetos Git. Vamos dedicar um minuto para ver como o Git armazena seus objetos. Você verá como armazenar um objeto blob — neste caso, a string “what is up, doc?” — interativamente na linguagem de script Ruby.

Você pode iniciar o modo Ruby interativo com o comando irb :

```
$ irb
>> content = "o que houve, doc?"
=> "o que foi, doutor?"
```

O Git primeiro constrói um cabeçalho que começa identificando o tipo de objeto — neste caso, um blob. Para essa primeira parte do cabeçalho, o Git adiciona um espaço seguido pelo tamanho em bytes do conteúdo e adiciona um byte nulo final:

```
>> header = "blob #{content.bytesize}\0" =>
"blob 16\u0000"
```

O Git concatena o cabeçalho e o conteúdo original e, em seguida, calcula a soma de verificação SHA-1 desse novo conteúdo. Você pode calcular o valor SHA-1 de uma string em Ruby incluindo o SHA1

Machine Translated by Google

digest library com o comando require e então chamando Digest::SHA1.hexdigest() com a string:

```
>> store = header + content
=> "blob 16\u0000what's up, doc?" >>
require 'digest/sha1' => true >> sha1
= Digest::SHA1.hexdigest(store) =>
"bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Vamos comparar isso com a saída do git hash-object. Aqui usamos echo -n para evitar adicionar uma nova linha à entrada.

```
$ echo -n "o que houve, doutor?" | git hash-object --stdin
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Git comprime o novo conteúdo com zlib, o que você pode fazer em Ruby com a biblioteca zlib. Primeiro, você precisa requerer a biblioteca e depois executar Zlib::Deflate.deflate() no conteúdo:

```
>> require 'zlib' =>
true >> zlib_content
= Zlib::Deflate.deflate(store) =>
"x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\ a\x00_\x1C\a\x9D"
```

Por fim, você gravará seu conteúdo esvaziado por zlib em um objeto no disco. Você determinará o caminho do objeto que deseja gravar (os dois primeiros caracteres do valor SHA-1 sendo o nome do subdiretório e os últimos 38 caracteres sendo o nome do arquivo nesse diretório). Em Ruby, você pode usar a função FileUtils.mkdir_p() para criar o subdiretório se ele não existir. Em seguida, abra o arquivo com File.open() e grave o conteúdo previamente compactado com zlib no arquivo com uma chamada write() no identificador de arquivo resultante:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38] => ".git/
objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37" >>
requerem 'fileutils'
=> verdadeiro
>> FileUtils.mkdir_p(File.dirname(path)) =>
".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content } => 32
```

Vamos verificar o conteúdo do objeto usando git cat-file:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
e aí, doc?
```

Machine Translated by Google

```
---
```

É isso – você criou um objeto Git blob válido.

Todos os objetos Git são armazenados da mesma forma, apenas com tipos diferentes – em vez do blob de string, o cabeçalho começará com commit ou tree. Além disso, embora o conteúdo do blob possa ser quase qualquer coisa, o conteúdo da confirmação e da árvore são formatados de maneira muito específica.

Referências do Git

Se você estiver interessado em ver o histórico de seu repositório acessível a partir do commit, digamos, 1a410e, você pode executar algo como git log 1a410e para exibir esse histórico, mas você ainda teria que lembrar que 1a410e é o commit que você deseja usar como o ponto de partida para essa história. Em vez disso, seria mais fácil se você tivesse um arquivo no qual pudesse armazenar esse valor SHA-1 com um nome simples para poder usar esse nome simples em vez do valor bruto SHA-1.

No Git, esses nomes simples são chamados de “referências” ou “refs”; você pode encontrar os arquivos que contêm esses valores SHA-1 no diretório .git/refs . No projeto atual, este diretório não contém arquivos, mas contém uma estrutura simples:

```
$ find .git/refs .git/
 refs .git/refs/
 heads .git/refs/
 tags $ find .git/refs
-type f
```

Para criar uma nova referência que o ajudará a lembrar onde está seu último commit, você pode tecnicamente fazer algo simples como isto:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

Agora, você pode usar a referência head que acabou de criar em vez do valor SHA-1 em seus comandos Git:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Terceiro
commit cac0cab538b970a37ea1e769cbbde608743bc96d
Segundo commit fdf4fc3344e67ab068f836878b6c4951e3b Primeiro commit
```

Você não é encorajado a editar diretamente os arquivos de referência; em vez disso, o Git fornece o comando mais seguro git update-ref para fazer isso se você quiser atualizar uma referência:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Machine Translated by Google

Isso é basicamente o que é um branch no Git: um simples ponteiro ou referência ao início de uma linha de trabalho. Para criar um branch no segundo commit, você pode fazer isso:

```
$ git update-ref refs/heads/test cac0ca
```

Sua ramificação conterá apenas o trabalho desse commit abaixo:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d Segundo commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d Primeiro commit
```

Agora, seu banco de dados Git é conceitualmente mais ou menos assim:

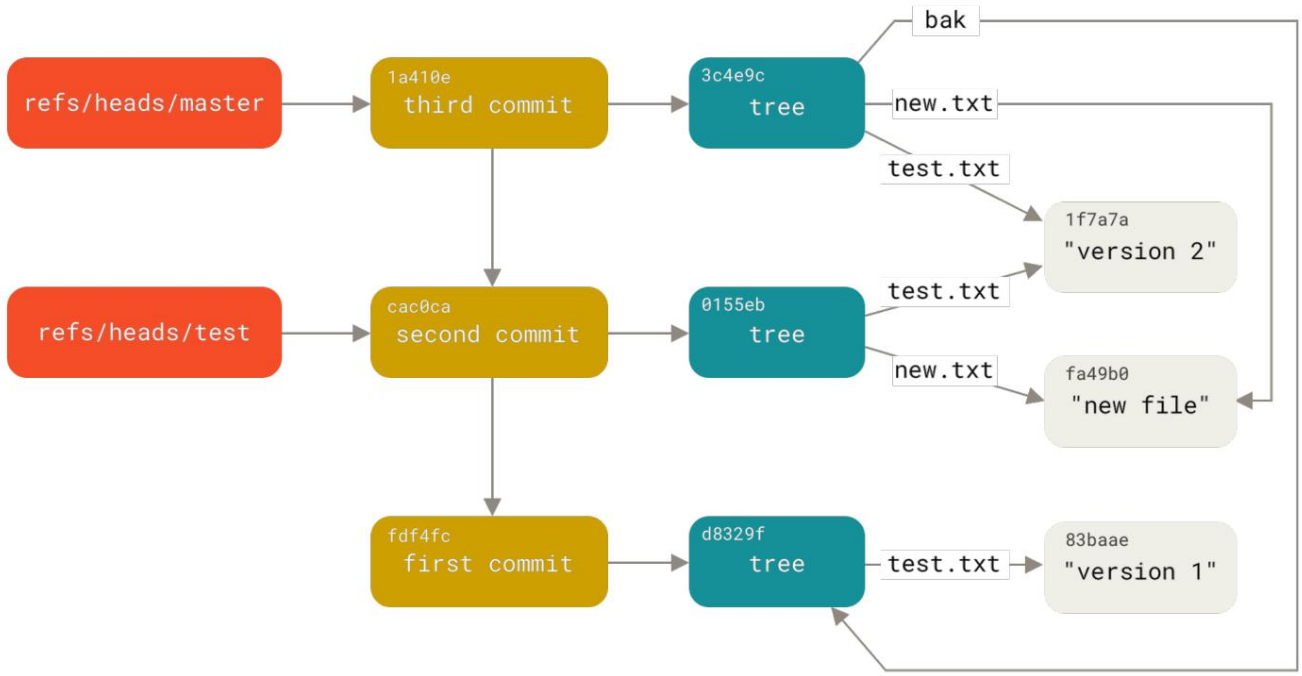


Figura 150. Objetos de diretório Git com referências de branch head incluídas

Quando você executa comandos como git branch <branch>, o Git basicamente executa o comando update-ref para adicionar o SHA-1 do último commit do branch em que você está em qualquer nova referência que você deseja criar.

A cabeça

A questão agora é, quando você executa git branch <branch>, como o Git sabe o SHA-1 do último commit? A resposta é o arquivo HEAD.

Normalmente, o arquivo HEAD é uma referência simbólica ao branch em que você está atualmente. Por referência simbólica, queremos dizer que, ao contrário de uma referência normal, ela contém um ponteiro para outra referência.

No entanto, em alguns casos raros, o arquivo HEAD pode conter o valor SHA-1 de um objeto git. Isso acontece quando você faz checkout de uma tag, commit ou branch remoto, o que coloca seu repositório em "HEAD desanexado " estado.

Machine Translated by Google

Se você olhar o arquivo, normalmente verá algo assim:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Se você executar git checkout test, o Git atualizará o arquivo para ficar assim:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Quando você executa git commit, ele cria o objeto commit, especificando o pai desse objeto commit para ser qualquer valor SHA-1 para o qual a referência em HEAD aponta.

Você também pode editar manualmente este arquivo, mas, novamente, existe um comando mais seguro para fazer isso: git simbólico-ref. Você pode ler o valor do seu HEAD através deste comando:

```
$ git Symbolic-ref HEAD
refs/heads/master
```

Você também pode definir o valor de HEAD usando o mesmo comando:

```
$ git symbol-ref HEAD refs/heads/test $
cat .git/HEAD ref: refs/heads/test
```

Você não pode definir uma referência simbólica fora do estilo refs:

```
$ git Symbolic-ref HEAD test
fatal: recusando-se a apontar HEAD fora de refs/
```

Tag

Acabamos de discutir os três principais tipos de objeto do Git *(blobs, tree* e *commits),* mas há um quarto. O objeto *tag* é muito parecido com um objeto commit — ele contém um tagger, uma data, uma mensagem e um ponteiro. A principal diferença é que um objeto tag geralmente aponta para um commit em vez de uma árvore. É como uma referência de ramificação, mas nunca se move — sempre aponta para o mesmo commit, mas dá a ele um nome mais amigável.

Conforme discutido no Git Basics, existem dois tipos de tags: anotadas e leves. Você pode criar uma tag leve executando algo assim:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Machine Translated by Google

Isso é tudo que uma tag leve é - uma referência que nunca se move. Uma tag anotada é mais complexa, no entanto. Se você criar uma tag anotada, o Git criará um objeto de tag e, em seguida, gravará uma referência para apontar para ela em vez de diretamente para o commit. Você pode ver isso criando uma tag anotada (usando a opção -a):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Tag de teste'
```

Aqui está o valor SHA-1 do objeto que ele criou:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Agora, execute git cat-file -p nesse valor SHA-1:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2 object
1a410efbd13591db07496601ebc7a059dd55cfe9 type commit tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sáb 23 de maio 16:48:58
-07 2009


dia de teste
```

Observe que a entrada do objeto aponta para o valor commit SHA-1 que você marcou. Observe também que não precisa apontar para um commit; você pode marcar qualquer objeto Git. No código-fonte do Git, por exemplo, o mantenedor adicionou sua chave pública GPG como um objeto blob e o marcou. Você pode visualizar a chave pública executando isso em um clone do repositório Git:

```
$ git cat-file blob junio-gpg-pub
```

O repositório do kernel Linux também possui um objeto tag não apontador de commit — o primeiro tag criado aponta para a árvore inicial da importação do código-fonte.

Remotos

O terceiro tipo de referência que você verá é uma referência remota. Se você adicionar um remoto e enviar para ele, o Git armazenará o último valor que você enviou para esse remoto para cada ramificação no diretório refs/remotes . Por exemplo, você pode adicionar um remoto chamado origin e enviar seu branch master para ele:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git $ git push
origin master Contagem de objetos: 11, concluído.

Comprimindo objetos: 100% (5/5), concluído.
Escrevendo objetos: 100% (7/7), 716 bytes, concluído.
Total 7 (delta 2), reutilizado 4 (delta 1)
Para git@github.com:schacon/simplegit-progit.git
```

Machine Translated by Google

```
a11bef0..ca82a6d mestre -> mestre
```

Então, você pode ver qual era o branch master no remoto de origem na última vez que você se comunicou com o servidor, verificando o arquivo refs/remotes/origin/master :

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

As referências remotas diferem das ramificações (referências refs/heads) principalmente porque são consideradas somente leitura. Você pode git checkout para um, mas o Git não irá referenciar HEAD simbolicamente para um, então você nunca irá atualizá-lo com um comando commit . O Git os gerencia como marcadores para o último estado conhecido de onde essas ramificações estavam nesses servidores.

Packfiles

Se você seguiu todas as instruções do exemplo da seção anterior, agora deve ter um repositório Git de teste com 11 objetos — quatro blobs, três árvores, três commits e uma tag:

```
$ find .git/objects -type f .git/
objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2 .git/
objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit
3 .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt
v2 .git/objects/3c /4e9cd789d88d8d89c1073707c3585e41b0e614 # tree
3 .git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt
v1 .git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag .git/
objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2 .git/
objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content' .git/
objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1 .git/objects/fa/
49b077972391ad58037050f2a75f74e3671e92 # new.txt .git/objects/fd/
f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git comprime o conteúdo desses arquivos com zlib, e você não está armazenando muito, então todos esses arquivos coletivamente ocupam apenas 925 bytes. Agora você adicionará algum conteúdo mais considerável ao repositório para demonstrar um recurso interessante do Git. Para demonstrar, adicionaremos o arquivo repo.rb da biblioteca Grit — trata-se de um arquivo de código-fonte de 22K:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git checkout master $ git add repo.rb $ git commit -m 'Criar repo.rb' [mestre 484a592]
Criar repo.rb 3 arquivos alterados, 709 inserções (+), 2 exclusões (-) modo de exclusão
100644 bak/test.txt modo de criação 100644 repo.rb reescrever test.txt (100%)
```

Machine Translated by Google

Se você observar a árvore resultante, poderá ver o valor SHA-1 que foi calculado para seu novo objeto blob repo.rb :

```
$ git cat-file -p master^{tree} 100644
blob fa49b077972391ad58037050f2a75f74e3671e92 100644        novo.txt
blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 100644        repo.rb
blob e3f094f522629ae358806b17daf78246c27c007b               teste.txt
```

Você pode então usar git cat-file para ver o tamanho desse objeto:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Neste ponto, modifique um pouco esse arquivo e veja o que acontece:

```
$ echo '# testing' >> repo.rb $ git
commit -am 'Modificar repo.rb um
pouco' [master 2431da6] Modificar repo.rb
um pouco 1 arquivo alterado, 1 inserção(+)
```

Verifique a árvore criada por esse último commit e você verá algo interessante:

```
$ git cat-file -p master^{tree} 100644
blob fa49b077972391ad58037050f2a75f74e3671e92 100644        novo.txt
blob b042a60ef7dff760008df33cee372b945b6e884e               repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b        teste.txt
```

O blob agora é um blob diferente, o que significa que, embora você tenha adicionado apenas uma única linha ao final de um arquivo de 400 linhas, o Git armazenou esse novo conteúdo como um objeto completamente novo:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e 22054
```

Você tem dois objetos de 22K quase idênticos em seu disco (cada um compactado em aproximadamente 7K).
Não seria bom se o Git pudesse armazenar um deles por completo, mas o segundo objeto apenas como o delta entre ele e o primeiro?

Acontece que pode. O formato inicial no qual o Git salva objetos em disco é chamado de formato de objeto “solto”. No entanto, ocasionalmente o Git empacota vários desses objetos em um único arquivo binário chamado “packfile” para economizar espaço e ser mais eficiente. O Git faz isso se você tiver muitos objetos soltos por perto, se executar o comando git gc manualmente ou se enviar para um servidor remoto. Para ver o que acontece, você pode pedir manualmente ao Git para empacotar os objetos chamando o comando git gc :

```
$ git gc
```

Machine Translated by Google

```
Contando objetos: 18, feito.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (14/14), concluído.
Objetos de escrita: 100% (18/18), concluído.
Total 18 (delta 3), reutilizado 0 (delta 0)
```

Se você olhar em seu diretório de objetos , descobrirá que a maioria de seus objetos se foi e um novo par de arquivos apareceu:

```
$ 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
```

Os objetos que permanecem são os blobs que não são apontados por nenhum commit — neste caso, o “what is up, doc?” exemplo e os blobs de exemplo de “conteúdo de teste” que você criou anteriormente. Como você nunca os adicionou a nenhum commit, eles são considerados pendentes e não são incluídos em seu novo packfile.

Os outros arquivos são seu novo packfile e um index. O packfile é um único arquivo contendo o conteúdo de todos os objetos que foram removidos do seu sistema de arquivos. O índice é um arquivo que contém deslocamentos nesse packfile para que você possa buscar rapidamente um objeto específico. O que é legal é que, embora os objetos no disco antes de você executar o comando gc tivessem coletivamente cerca de 15K de tamanho, o novo packfile tem apenas 7K. Você reduziu o uso do disco pela metade empacotando seus objetos.

Como o Git faz isso? Quando o Git empacota objetos, ele procura arquivos com nomes e tamanhos semelhantes e armazena apenas os deltas de uma versão do arquivo para a seguinte. Você pode examinar o packfile e ver o que o Git fez para economizar espaço. O comando git verify-pack plumbing permite que você veja o que foi empacotado:

```
$ git verify-pack -v .git/objects/pack/pack
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
deef2e1b793907545e50a2ea2ddb5ba6c58c4506
```

Machine Translated by Google

```
0155eb4229851634a0f03eb265b69f5a2d56f341 árvore 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
não delta: 15 objetos comprimento da cadeia = 1: 3 objetos .git/
objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack:
ok
```

Aqui, o blob 033b4 , que se você se lembra era a primeira versão do seu arquivo repo.rb , está referenciando o blob b042a , que era a segunda versão do arquivo. A terceira coluna na saída é o tamanho do objeto no pacote, então você pode ver que b042a ocupa 22K do arquivo, mas que 033b4 ocupa apenas 9 bytes. O que também é interessante é que a segunda versão do arquivo é aquela que está armazenada intacta, enquanto a versão original é armazenada como um delta — isso ocorre porque é mais provável que você precise de acesso mais rápido à versão mais recente do arquivo .

A coisa realmente legal sobre isso é que ele pode ser reembalado a qualquer momento. O Git irá ocasionalmente reempacotar seu banco de dados automaticamente, sempre tentando economizar mais espaço, mas você também pode reempacotar manualmente a qualquer momento executando git gc manualmente.

o Refspec

Ao longo deste livro, usamos mapeamentos simples de ramificações remotas para referências locais, mas eles podem ser mais complexos. Suponha que você esteja acompanhando as últimas duas seções e tenha criado um pequeno repositório Git local e agora queira adicionar um *controle remoto* a ele:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

A execução do comando acima adiciona uma seção ao arquivo .git/config do seu repositório , especificando o nome do remoto (origem), a URL do repositório remoto e a *refspec* a ser usada para buscar:

```
[remote "origin"] url
= https://github.com/schacon/simplegit-progit fetch = +refs/
heads/*:refs/remotes/origin/*
```

O formato do refspec é, primeiro, um + opcional, seguido por <src>:<dst>, onde <src> é o padrão para referências no lado remoto e <dst> é onde essas referências serão rastreadas localmente. O + diz ao Git para atualizar a referência mesmo que não seja um avanço rápido.

No caso padrão que é gravado automaticamente por um comando git remote add origin , o Git busca todas as referências em refs/heads/ no servidor e as grava em refs/remotes/origin/ localmente. Portanto, se houver uma ramificação principal no servidor, você poderá acessar o log dessa ramificação localmente por meio de qualquer um dos seguintes:

Machine Translated by Google

```
$ git log origin/master $ git
log remotes/origin/master $ git log refs/
remotes/origin/master
```

Eles são todos equivalentes, porque o Git expande cada um deles para refs/remotes/origin/master.

Se você quiser que o Git, em vez disso, baixe apenas o branch master a cada vez, e não todos os outros branches no servidor remoto, você pode alterar a linha de busca para se referir apenas a esse branch:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Este é apenas o refspec padrão para git fetch para esse controle remoto. Se você deseja fazer uma busca única, também pode especificar a refspec específica na linha de comando. Para puxar a ramificação master no controle remoto para origin/mymaster localmente, você pode executar:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Você também pode especificar várias refspecs. Na linha de comando, você pode baixar vários ramos da seguinte forma:

```
$ git fetch origin master:refs/remotes/origin/mymaster \ topic:refs/
        remotes/origin/topic De git@github.com:schacon/simplegit !
[rejeitado] * [novo ramo]
                                tópico        -> origem/meumestre (sem avanço rápido) ->
                                mestre        origem/tópico
```

Nesse caso, o pull do branch master foi rejeitado porque não foi listado como uma referência de avanço rápido.
Você pode substituir isso especificando o + na frente do refspec.

Você também pode especificar várias refspecs para buscar em seu arquivo de configuração. Se você quiser sempre buscar as ramificações master e experimental do controle remoto de origem , adicione duas linhas:

```
[remote "origin"] url
= https://github.com/schacon/simplegit-progit fetch = +refs/heads/
master:refs/remotes/origin/master fetch = +refs/heads/experiment:refs/
remotes/ origem/experiência
```

Desde o Git 2.6.0, você pode usar globs parciais no padrão para corresponder a várias ramificações, então isso funciona:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Melhor ainda, você pode usar namespaces (ou diretórios) para realizar o mesmo com mais estrutura.
Se você tem uma equipe de controle de qualidade que envia uma série de branches e deseja obter o branch master e

Machine Translated by Google

qualquer uma das ramificações da equipe de controle de qualidade, mas nada mais, você pode usar uma seção de configuração como esta:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit fetch =
+refs/heads/master:refs/remotes/origin/master fetch = +refs/
heads/qa/*:refs/ controles remotos/origem/qa/*
```

Se você tiver um processo de fluxo de trabalho complexo que tenha uma equipe de controle de qualidade enviando ramificações, desenvolvedores enviando ramificações e equipes de integração pressionando e colaborando em ramificações remotas, você pode nomeá-los facilmente dessa maneira.

Empurrando Refspecs

É bom que você possa buscar referências de namespace dessa maneira, mas como a equipe de controle de qualidade coloca suas ramificações em um qa/ namespace em primeiro lugar? Você consegue isso usando refspecs para empurrar.

Se a equipe de QA quiser enviar sua ramificação master para qa/master no servidor remoto, eles podem executar:

```
$ git push origin master:refs/heads/qa/master
```

Se eles quiserem que o Git faça isso automaticamente toda vez que executarem o git push origin, eles podem adicionar um valor de push ao arquivo de configuração:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit fetch =
+refs/heads/*:refs/remotes/origin/* push = refs/heads/
master:refs/heads/qa /mestre
```

Novamente, isso fará com que um git push origin envie o branch master local para o branch qa/master remoto por padrão.

ÿ Você não pode usar o refspec para buscar de um repositório e enviar para outro. Para obter um exemplo de como fazer isso, consulte Mantenha seu repositório público do GitHub atualizado.

Excluindo referências

Você também pode usar o refspec para excluir referências do servidor remoto executando algo assim:

```
$ git push origem:tópico
```

Como o refspec é <src>:<dst>, deixando de fora a parte <src> , isso basicamente diz para fazer o tópico ramificar no nada remoto, o que o exclui.

Ou você pode usar a sintaxe mais recente (disponível desde Git v1.7.0):

Machine Translated by Google

```
$ git push origin --delete tópico
```

Protocolos de transferência

O Git pode transferir dados entre dois repositórios de duas maneiras principais: o protocolo “burro” e o protocolo “inteligente”. Esta seção abordará rapidamente como esses dois protocolos principais operam.

O protocolo estúpido

Se você estiver configurando um repositório para ser servido somente leitura por HTTP, o protocolo burro provavelmente será usado. Esse protocolo é chamado de “burro” porque não requer nenhum código específico do Git no lado do servidor durante o processo de transporte; o processo de busca é uma série de requisições HTTP GET , onde o cliente pode assumir o layout do repositório Git no servidor.

ÿ

O protocolo burro raramente é usado hoje em dia. É difícil proteger ou tornar privado, então a maioria dos hosts Git (tanto baseados em nuvem quanto no local) se recusará a usá-lo.
Geralmente é aconselhável usar o protocolo inteligente, que descrevemos um pouco mais adiante.

Vamos seguir o processo http-fetch para a biblioteca simplegit:

```
$ git clone http://servidor/simplegit-progit.git
```

A primeira coisa que esse comando faz é baixar o arquivo info/refs . Este arquivo é escrito pelo comando update server-info , e é por isso que você precisa ativá-lo como um gancho pós-recebimento para que o transporte HTTP funcione corretamente:

```
=> OBTER info/refs
ca82a6dff817ec66f44342007202690a93763949                refs/cabeças/mestre
```

Agora você tem uma lista de referências remotas e SHA-1s. Em seguida, você procura qual é a referência HEAD para saber o que verificar quando terminar:

```
=> GET HEAD
ref: refs/heads/master
```

Você precisa verificar a ramificação master quando concluir o processo. Neste ponto, você está pronto para iniciar o processo de caminhada. Como seu ponto de partida é o objeto de commit ca82a6 que você viu no arquivo info/refs , você começa buscando isso:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949 (179 bytes
de dados binários)
```

Você recebe um objeto de volta – esse objeto está em formato solto no servidor e você o buscou por meio de um estático

Machine Translated by Google

Solicitação HTTP GET. Você pode descompactá-lo com zlib, remover o cabeçalho e examinar o conteúdo do commit:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949 tree
cfda3bf379e4f8dba8717dee55aab78aef7f4daf parent
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 author Scott Chacon
<schacon@gmail.com> 1205815931 -0700 committer Scott Chacon
<schacon@gmail.com> 1240030591 -0700

Alterar número da versão
```

Em seguida, você tem mais dois objetos para recuperar – cfda3b, que é a árvore de conteúdo para a qual o commit que acabamos de recuperar aponta; e 085bb3, que é o commit pai:

```
=> OBTER objetos/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 (179
bytes de dados)
```

Isso fornece seu próximo objeto de confirmação. Pegue o objeto árvore:

```
=> OBTER objetos/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 não encontrado)
```

Ops – parece que aquele objeto de árvore não está em formato solto no servidor, então você recebe uma resposta 404 de volta. Existem algumas razões para isso – o objeto pode estar em um repositório alternativo ou pode estar em um packfile neste repositório. O Git verifica primeiro as alternativas listadas:

```
=> GET objects/info/http-alternates (arquivo
vazio)
```

Se isso retornar com uma lista de URLs alternativos, o Git verifica se há arquivos soltos e packfiles lá – esse é um bom mecanismo para projetos que são bifurcações um do outro para compartilhar objetos no disco. No entanto, como não há alternativas listadas nesse caso, seu objeto deve estar em um packfile. Para ver quais packfiles estão disponíveis neste servidor, você precisa obter o arquivo objects/info/packs , que contém uma listagem deles (também gerado por update-server-info):

```
=> OBTER objetos/informações/
pacotes P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Há apenas um arquivo de pacote no servidor, então seu objeto obviamente está lá, mas você verificará o arquivo de índice para ter certeza. Isso também é útil se você tiver vários packfiles no servidor, para poder ver qual packfile contém o objeto que você precisa:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx (4k de
dados binários)
```

Machine Translated by Google

Agora que você tem o índice do packfile, pode ver se o seu objeto está nele – porque o índice lista os SHA-1s dos objetos contidos no packfile e os deslocamentos para esses objetos. Seu objeto está lá, então vá em frente e pegue todo o packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k de dados binários)
```

Você tem seu objeto de árvore, então você continua andando com seus commits. Eles também estão dentro do packfile que você acabou de baixar, então você não precisa fazer mais solicitações ao seu servidor. O Git verifica uma cópia de trabalho do branch master que foi apontado pela referência HEAD que você baixou no início.

O Protocolo Inteligente

O protocolo burro é simples, mas um pouco ineficiente, e não pode lidar com a gravação de dados do cliente para o servidor. O protocolo inteligente é um método mais comum de transferência de dados, mas requer um processo na extremidade remota que é inteligente sobre o Git – ele pode ler dados locais, descobrir o que o cliente tem e precisa e gerar um arquivo de pacote personalizado para ele. Existem dois conjuntos de processos para transferência de dados: um par para upload de dados e um par para download de dados.

Carregando dados

Para fazer upload de dados para um processo remoto, o Git usa os processos send-pack e receive-pack . O processo send pack é executado no cliente e se conecta a um processo receive-pack no lado remoto.

SSH

Por exemplo, digamos que você execute git push origin master em seu projeto e a origem seja definida como uma URL que usa o protocolo SSH. O Git inicia o processo send-pack , que inicia uma conexão via SSH com seu servidor. Ele tenta executar um comando no servidor remoto por meio de uma chamada SSH que se parece com isto:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/masterÿreport-status \
delete-refs side-band-64k quiet ofs-delta \ agent=git/2 :2.1.1+github-607-gfba4028
delete-refs 0000
```

O comando git-receive-pack responde imediatamente com uma linha para cada referência que possui atualmente – neste caso, apenas o branch master e seu SHA-1. A primeira linha também contém uma lista dos recursos do servidor (aqui, report-status, delete-refs e alguns outros, incluindo o identificador do cliente).

Os dados são transmitidos em blocos. Cada pedaço começa com um valor hexadecimal de 4 caracteres especificando quanto tempo o pedaço é (incluindo os 4 bytes do próprio comprimento). Os pedaços geralmente contêm uma única linha de dados e uma alimentação de linha à direita. Seu primeiro bloco começa com 00a5, que é hexadecimal para 165, o que significa que o bloco tem 165 bytes de comprimento. A próxima parte é 0000, o que significa que o servidor concluiu sua listagem de referências.

Machine Translated by Google

Agora que conhece o estado do servidor, seu processo send-pack determina quais commits ele possui que o servidor não possui. Para cada referência que este push atualizará, o processo send-pack informa ao processo receive-pack essa informação. Por exemplo, se você estiver atualizando o branch master e adicionando um branch experimental , a resposta do send-pack pode ser mais ou menos assim:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
\  refs/heads/master report-status 006c0000000000000000000000000000000000000000
cdfdb42577e2506715f8cfeacdbabc092bf63e8d \  refs/heads/experiment 0000
```

O Git envia uma linha para cada referência que você está atualizando com o comprimento da linha, o antigo SHA-1, o novo SHA-1 e a referência que está sendo atualizada. A primeira linha também tem as capacidades do cliente. O valor SHA-1 de todos os '0's significa que não havia nada antes - porque você está adicionando a referência do experimento. Se você estivesse excluindo uma referência, veria o oposto: todos os '0's no lado direito.

Em seguida, o cliente envia um packfile com todos os objetos que o servidor ainda não possui. Finalmente, o servidor responde com uma indicação de sucesso (ou falha):

```
000eunpack ok
```

HTTP(S)

Esse processo é basicamente o mesmo em HTTP, embora o handshake seja um pouco diferente. A conexão é iniciada com esta solicitação:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack 001f# service=git-
receive-pack 00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/masterÿreport-
status \ delete-refs side -band-64k quiet ofs-delta \ agent=git/2:2.1.1~vmg-bitmaps-
bugaloo-608-g116744e 0000
```

Esse é o fim da primeira troca cliente-servidor. O cliente então faz outra requisição, desta vez um POST, com os dados que o send-pack fornece.

```
=> POST http://servidor/simplegit-progit.git/git-receive-pack
```

A solicitação POST inclui a saída do send-pack e o packfile como sua carga útil. O servidor então indica sucesso ou falha com sua resposta HTTP.

Lembre-se de que o protocolo HTTP pode envolver ainda mais esses dados em uma codificação de transferência em partes.

Machine Translated by Google

Baixando dados

Quando você baixa dados, os processos fetch-pack e upload-pack estão envolvidos. O cliente inicia um processo fetch-pack que se conecta a um processo upload-pack no lado remoto para negociar quais dados serão transferidos.

SSH

Se você está fazendo a busca por SSH, o fetch-pack executa algo assim:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Após a conexão do fetch-pack , o upload-pack envia de volta algo como isto:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADÿmulti_ack thin-pack \ side-band
side-band-64k ofs-delta raso sem progresso include-tag \ multi_ack_detailed
symref=HEAD:refs/heads/master \ agent=git/2:2.1.1+github- 607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master

0000
```

Isso é muito semelhante ao que o receive-pack responde, mas os recursos são diferentes. Além disso, ele envia de volta o que o HEAD aponta (symref=HEAD:refs/heads/master) para que o cliente saiba o que verificar se for um clone.

Nesse ponto, o processo fetch-pack verifica quais objetos possui e responde com os objetos de que precisa enviando “want” e, em seguida, o SHA-1 que deseja. Ele envia todos os objetos que já possui com “have” e depois o SHA-1. No final desta lista, escreve “done” para iniciar o processo de upload-pack para começar a enviar o packfile dos dados de que necessita:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 0009concluído

0000
```

HTTP(S)

O handshake para uma operação de busca requer duas solicitações HTTP. O primeiro é um GET para o mesmo endpoint usado no protocolo burro:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack 001e#
service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEADÿmulti_ack thin-pack \ banda
lateral banda lateral-64k ofs-delta rasa sem progresso include -tag \ multi_ack_detailed no-
done symref=HEAD:refs/heads/master \ agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

Machine Translated by Google

```
0000
```

Isso é muito semelhante a invocar o git-upload-pack em uma conexão SSH, mas a segunda troca é realizada como uma solicitação separada:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Novamente, este é o mesmo formato acima. A resposta a esta solicitação indica sucesso ou falha e inclui o packfile.

Resumo dos Protocolos

Esta seção contém uma visão geral muito básica dos protocolos de transferência. O protocolo inclui muitos outros recursos, como multi_ack ou recursos de banda lateral , mas abordá-los está fora do escopo deste livro. Tentamos dar a você uma ideia do vai-e-vem geral entre cliente e servidor; se você precisar de mais conhecimento do que isso, provavelmente desejará dar uma olhada no código-fonte do Git.

Manutenção e Recuperação de Dados

Ocasionalmente, pode ser necessário fazer alguma limpeza – tornar um repositório mais compacto, limpar um repositório importado ou recuperar o trabalho perdido. Esta seção abordará alguns desses cenários.

Manutenção

Ocasionalmente, o Git executa automaticamente um comando chamado “auto gc”. Na maioria das vezes, esse comando não faz nada. No entanto, se houver muitos objetos soltos (objetos que não estão em um packfile) ou muitos packfiles, o Git inicia um comando git gc completo . O “gc” significa coleta de lixo, e o comando faz várias coisas: ele reúne todos os objetos soltos e os coloca em packfiles, consolida packfiles em um grande packfile e remove objetos que não podem ser acessados de nenhum cometem e têm alguns meses de idade.

Você pode executar o auto gc manualmente da seguinte maneira:

```
$ git gc --auto
```

Novamente, isso geralmente não faz nada. Você deve ter cerca de 7.000 objetos soltos ou mais de 50 packfiles para o Git disparar um comando gc real. Você pode modificar esses limites com as configurações gc.auto e gc.autopacklimit , respectivamente.

A outra coisa que o gc fará é empacotar suas referências em um único arquivo. Suponha que seu repositório contenha as seguintes ramificações e tags:

```
$ find .git/refs -type f
```

Machine Translated by Google

```
.git/refs/heads/experiment .git/
refs/heads/master .git/refs/tags/
v1.0 .git/refs/tags/v1.1
```

Se você executar git gc, não terá mais esses arquivos no diretório refs . Git irá movê-los para uma questão de eficiência em um arquivo chamado .git/packed-refs que se parece com este:

```
$ 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
```

Se você atualizar uma referência, o Git não edita esse arquivo, mas grava um novo arquivo em refs/heads. Para obter o SHA-1 apropriado para uma determinada referência, o Git verifica essa referência no diretório refs e, em seguida, verifica o arquivo pack-refs como substituto. Portanto, se você não conseguir encontrar uma referência no diretório refs , provavelmente está no seu arquivo pack-refs .

Observe a última linha do arquivo, que começa com um ^. Isso significa que a tag diretamente acima é uma tag anotada e essa linha é o commit para o qual a tag anotada aponta.

Recuperação de dados

Em algum momento de sua jornada Git, você pode acidentalmente perder um commit. Geralmente, isso acontece porque você força a exclusão de um branch que tinha trabalho nele e, afinal, você queria o branch; ou você reinicializa um branch, abandonando assim os commits dos quais você queria algo.
Supondo que isso aconteça, como você pode recuperar seus commits?

Aqui está um exemplo que redefine o branch master em seu repositório de teste para um commit mais antigo e então recupera os commits perdidos. Primeiro, vamos revisar onde seu repositório está neste momento:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Agora, mova o branch master de volta para o commit do meio:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD está agora em 1a410ef Terceiro
commit $ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Terceiro commit
```

Machine Translated by Google

```
cac0cab538b970a37ea1e769cbbde608743bc96d Segundo commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d Primeiro commit
```

Você efetivamente perdeu os dois principais commits – você não tem branch a partir do qual esses commits possam ser acessados. Você precisa encontrar o commit SHA-1 mais recente e adicionar uma ramificação que aponte para ele. O truque é encontrar o último commit SHA-1 – não é como se você o tivesse memorizado, certo?

Muitas vezes, a maneira mais rápida é usar uma ferramenta chamada git reflog. Enquanto você trabalha, o Git registra silenciosamente qual é o seu HEAD toda vez que você o altera. Cada vez que você confirma ou altera branches, o reflog é atualizado. O reflog também é atualizado pelo comando git update-ref , que é outro motivo para usá-lo em vez de apenas escrever o valor SHA-1 em seus arquivos ref, conforme abordamos em Referências do Git. Você pode ver onde esteve a qualquer momento executando git reflog:

```
$ git reflog
1a410ef HEAD@{0}: reset: movendo para 1a410ef ab1afef
HEAD@{1}: commit: Modifique repo.rb um pouco 484a592
HEAD@{2}: commit: Crie repo.rb
```

Aqui podemos ver os dois commits que verificamos, porém não há muita informação aqui. Para ver as mesmas informações de uma maneira muito mais útil, podemos executar git log -g, que fornecerá uma saída de log normal para o seu reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9 Reflog:
HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Mensagem de reflog: atualizando o
HEAD Autor: Scott Chacon <schacon@gmail.com>
Data: Sex May 22 18:22:37 2009 -0700

        Terceiro commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b Reflog:
HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Mensagem de reflog: atualizando o
HEAD Autor: Scott Chacon <schacon@gmail.com>
Data: Sex May 22 18:15:24 2009 -0700

        Modifique repo.rb um pouco
```

Parece que o commit inferior é aquele que você perdeu, então você pode recuperá-lo criando um novo branch nesse commit. Por exemplo, você pode iniciar um branch chamado recovery-branch naquele commit (ab1afef):

```
$ git branch recover-branch ab1afef $ git log --
pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
```

Machine Translated by Google

```
cac0cab538b970a37ea1e769cbbde608743bc96d Segundo commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d Primeiro commit
```

Legal – agora você tem um branch chamado recovery-branch que é onde seu branch master costumava estar, tornando os dois primeiros commits acessíveis novamente. Em seguida, suponha que sua perda não esteja, por algum motivo, no reflog - você pode simular isso removendo a ramificação de recuperação e excluindo o reflog. Agora os dois primeiros commits não são acessíveis por nada:

```
$ git branch -D recuperar-branch $ rm
-Rf .git/logs/
```

Como os dados do reflog são mantidos no diretório .git/logs/ , você efetivamente não tem reflog. Como você pode recuperar esse commit neste ponto? Uma maneira é usar o utilitário git fsck , que verifica a integridade do seu banco de dados. Se você executá-lo com a opção --full , ele mostra todos os objetos que não são apontados por outro objeto:

```
$ git fsck --full
Verificando diretórios de objetos: 100% (256/256), concluído.
Verificação de objetos: 100% (18/18), concluído.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4 dangling
commit ab1afef80fac8e34258ff41fc1b867c702daa24b dangling tree
aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9 dangling blob
7108f7ecb345ee9d0084193f147cdad4d2998293
```

Nesse caso, você pode ver o commit que falta após a string “dangling commit”. Você pode recuperá-lo da mesma forma, adicionando uma ramificação que aponta para esse SHA-1.

Removendo objetos

Há muitas coisas boas sobre o Git, mas um recurso que pode causar problemas é o fato de que um git clone baixa todo o histórico do projeto, incluindo todas as versões de todos os arquivos. Isso é bom se tudo for código-fonte, porque o Git é altamente otimizado para compactar esses dados com eficiência.
No entanto, se alguém em qualquer ponto da história do seu projeto adicionar um único arquivo enorme, todos os clones de todos os tempos serão forçados a baixar esse arquivo grande, mesmo que tenha sido removido do projeto no próximo commit. Por ser alcançável a partir da história, sempre estará lá.

Isso pode ser um grande problema quando você está convertendo repositórios Subversion ou Perforce em Git. Como você não baixa todo o histórico nesses sistemas, esse tipo de adição traz poucas consequências. Se você importou de outro sistema ou descobriu que seu repositório é muito maior do que deveria, veja como localizar e remover objetos grandes.

Esteja avisado: esta técnica é destrutiva para o seu histórico de commits. Ele reescreve cada objeto de confirmação desde a árvore mais antiga que você precisa modificar para remover uma referência de arquivo grande. Se você fizer isso imediatamente após uma importação, antes que alguém comece a basear o trabalho no commit, tudo bem – caso contrário, você deve notificar todos os contribuidores que eles devem rebasear seu trabalho em seus novos commits.

Machine Translated by Google

Para demonstrar, você irá adicionar um arquivo grande em seu repositório de teste, removê-lo no próximo commit, localizá-lo e removê-lo permanentemente do repositório. Primeiro, adicione um objeto grande ao seu histórico:

```
$ 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 'Adicionar git tarball ' [master 7b30847] Adicionar git tarball 1 arquivo alterado, 0
inserções (+), 0 exclusões (-) modo de criação 100644 git.tgz
```

Ops – você não queria adicionar um tarball enorme ao seu projeto. Melhor se livrar disso:

```
$ git rm git.tgz rm
'git.tgz' $ git commit
-m 'Oops - remover tarball grande' [master dadf725]
Ops - remover tarball grande 1 arquivo alterado, 0
inserções(+), 0 exclusões(-) modo de exclusão 100644
git.tgz
```

Agora, gc seu banco de dados e veja quanto espaço você está usando:

```
$ git gc
Contando objetos: 17, concluído.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (13/13), concluído.
Objetos de escrita: 100% (17/17), concluído.
Total 17 (delta 1), reutilizado 10 (delta 0)
```

Você pode executar o comando count-objects para ver rapidamente quanto espaço está usando:

```
$ git count-objects -v count:
7
tamanho: 32
embalagem: 17
pacotes: 1
embalagem de
tamanho: 4868
embalagem de ameixa:
0 lixo: 0 tamanho de lixo: 0
```

A entrada size-pack é o tamanho dos seus packfiles em kilobytes, então você está usando quase 5 MB. Antes do último commit, você estava usando perto de 2K – claramente, remover o arquivo do commit anterior não o removeu do seu histórico. Toda vez que alguém clonar este repositório, terá que clonar todos os 5 MB apenas para obter este pequeno projeto, porque você acidentalmente adicionou um arquivo grande. Vamos nos livrar disso.

Primeiro você tem que encontrá-lo. Nesse caso, você já sabe qual é o arquivo. Mas suponha que não; como

Machine Translated by Google

você identificaria qual arquivo ou arquivos estavam ocupando tanto espaço? Se você executar git gc, todos os objetos estarão em um packfile; você pode identificar os objetos grandes executando outro comando de encanamento chamado git Verify-Pack e classificando no terceiro campo na saída, que é o tamanho do arquivo. Você também pode canalizá-lo através do comando tail porque está interessado apenas nos últimos arquivos maiores:

```
$ git Verify-pack -v .git/objects/pack/pack-29…69.idx \ | classificar -k
3 -n \ | tail -3 dadf7258d699da2c8d89b09ef6670edb7d5f91b4
commit 229 159 12 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
blob 22044 5792 4977696 82c99a3e86bb1267b236a4b6eff7868d97489af1
blob 4975916 4976258 1438
```

O objeto grande está na parte inferior: 5 MB. Para descobrir qual é o arquivo, você usará o comando rev-list , que você usou rapidamente em Impondo um formato específico de mensagem de confirmação. Se você passar --objects para rev-list, ele listará todos os commit SHA-1s e também os blob SHA-1s com os caminhos de arquivo associados a eles. Você pode usar isso para encontrar o nome do seu blob:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Agora, você precisa remover este arquivo de todas as árvores do seu passado. Você pode ver facilmente quais commits modificaram este arquivo:

```
$ git log --oneline --branches -- git.tgz dadf725
Ops - remover tarball grande 7b30847 Adicionar
git tarball
```

Você deve reescrever todos os commits downstream de 7b30847 para remover completamente este arquivo do seu histórico do Git. Para fazer isso, você usa filter-branch, que você usou em Rewrite History:

```
$ git filter-branch --index-filter \ 'git rm --
ignore-unmatch --cached git.tgz' -- 7b30847^..
Reescrever 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Reescrever dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' foi reescrito
```

A opção --index-filter é semelhante à opção --tree-filter usada em Reescrever o histórico, exceto que, em vez de passar um comando que modifica arquivos com check-out no disco, você está modificando sua área de preparação ou índice a cada vez.

Em vez de remover um arquivo específico com algo como arquivo rm, você deve removê-lo com git rm --cached – você deve removê-lo do índice, não do disco. A razão para fazer dessa forma é a velocidade – como o Git não precisa verificar cada revisão no disco antes de executar seu filtro, o processo pode ser muito, muito mais rápido. Você pode realizar a mesma tarefa com --tree-filter se quiser. A opção --ignore-unmatch para git rm diz para não errar se o padrão que você está tentando remover não for

Machine Translated by Google

lá. Por fim, você pede ao filter-branch para reescrever seu histórico apenas a partir do commit 7b30847 , porque você sabe que foi aí que o problema começou. Caso contrário, ele começará do início e levará mais tempo desnecessariamente.

Seu histórico não contém mais uma referência a esse arquivo. No entanto, seu reflog e um novo conjunto de refs que o Git adicionou quando você fez a ramificação do filtro em .git/refs/original ainda o fazem, então você deve removê-los e reempacotar o banco de dados. Você precisa se livrar de qualquer coisa que tenha um ponteiro para esses commits antigos antes de reempacotar:

```
$ rm -Rf .git/refs/original $ rm
-Rf .git/logs/ $ git gc Contando
objetos: 15, concluído.

Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (11/11), concluído.
Objetos de escrita: 100% (15/15), concluído.
Total 15 (delta 1), reutilizado 12 (delta 0)
```

Vamos ver quanto espaço você economizou.

```
$ git count-objects -v count:
11
tamanho: 4904
embalagem: 15
embalagens: 1
embalagem de
tamanho: 8 embalagem
de ameixa: 0 lixo: 0
tamanho de lixo: 0
```

O tamanho do repositório compactado caiu para 8K, o que é muito melhor do que 5MB. Você pode ver pelo valor do tamanho que o objeto grande ainda está em seus objetos soltos, então não se foi; mas não será transferido em um push ou clone subseqüente, o que é importante. Se você realmente quiser, pode remover o objeto completamente executando git prune com a opção --expire :

```
$ git prune --expire now $ git
count-objects -v count: 0 size:
0 in-pack: 15 packs: 1 size-
pack: 8 prune-packable: 0
lixo: 0 tamanho-lixo: 0
```

Machine Translated by Google

variáveis ambientais

O Git sempre é executado dentro de um shell bash e usa várias variáveis de ambiente do shell para determinar como ele se comporta. Ocasionalmente, é útil saber o que são e como eles podem ser usados para fazer o Git se comportar da maneira que você deseja. Esta não é uma lista exaustiva de todas as variáveis de ambiente nas quais o Git presta atenção, mas abordaremos as mais úteis.

Comportamento global

Parte do comportamento geral do Git como um programa de computador depende de variáveis de ambiente.

GIT_EXEC_PATH determina onde o Git procura por seus subprogramas (como git-commit, git-diff e outros). Você pode verificar a configuração atual executando git --exec-path.

HOME geralmente não é considerado personalizável (muitas outras coisas dependem disso), mas é onde o Git procura o arquivo de configuração global. Se você deseja uma instalação do Git verdadeiramente portátil, completa com configuração global, você pode substituir HOME no perfil de shell do Git portátil.

PREFIX é semelhante, mas para a configuração de todo o sistema. O Git procura esse arquivo em $PREFIX/etc/gitconfig.

GIT_CONFIG_NOSYSTEM, se definido, desabilita o uso do arquivo de configuração de todo o sistema. Isso é útil se a configuração do seu sistema estiver interferindo em seus comandos, mas você não tem acesso para alterá-la ou removê-la.

GIT_PAGER controla o programa usado para exibir a saída de várias páginas na linha de comando. Se isso não for definido, o PAGER será usado como substituto.

GIT_EDITOR é o editor que o Git abrirá quando o usuário precisar editar algum texto (uma mensagem de commit, por exemplo). Se não for definido, o EDITOR será usado.

Locais de repositório

O Git usa várias variáveis de ambiente para determinar como ele interage com o repositório atual.

GIT_DIR é o local da pasta .git . Se isso não for especificado, o Git sobe na árvore de diretórios até chegar a ~ ou /, procurando por um diretório .git a cada etapa.

GIT_CEILING_DIRECTORIES controla o comportamento de pesquisa de um diretório .git . Se você acessar diretórios que são lentos para carregar (como aqueles em uma unidade de fita ou em uma conexão de rede lenta), você pode querer que o Git pare de tentar mais cedo do que poderia, especialmente se o Git for invocado ao criar seu prompt de shell .

GIT_WORK_TREE é o local da raiz do diretório de trabalho para um repositório não vazio. Se --git -dir ou GIT_DIR for especificado, mas nenhum --work-tree, GIT_WORK_TREE ou core.worktree for especificado, o diretório de trabalho atual será considerado o nível superior de sua árvore de trabalho.

GIT_INDEX_FILE é o caminho para o arquivo de índice (somente repositórios não vazios).

GIT_OBJECT_DIRECTORY pode ser usado para especificar a localização do diretório que geralmente reside em

Machine Translated by Google

.git/objetos.

GIT_ALTERNATE_OBJECT_DIRECTORIES é uma lista separada por dois-pontos (formatada como /dir/um:/dir/dois:…) que informa ao Git onde procurar objetos se eles não estiverem em GIT_OBJECT_DIRECTORY. Se você tiver muitos projetos com arquivos grandes com exatamente o mesmo conteúdo, isso pode ser usado para evitar o armazenamento de muitas cópias deles.

Pathspecs

Um "pathspec" refere-se a como você especifica caminhos para coisas no Git, incluindo o uso de curingas. Eles são usados no arquivo .gitignore , mas também na linha de comando (git add *.c).

GIT_GLOB_PATHSPECS e **GIT_NOGLOB_PATHSPECS** controlam o comportamento padrão de curingas em pathspecs. Se GIT_GLOB_PATHSPECS for definido como 1, os caracteres curinga agem como curingas (que é o padrão); se GIT_NOGLOB_PATHSPECS for definido como 1, os caracteres curinga correspondem apenas a si mesmos, o que significa que algo como *.c corresponderia apenas a um arquivo *chamado* "*.c", em vez de qualquer arquivo cujo nome termine com .c. Você pode substituir isso em casos individuais iniciando o pathspec com :(glob) ou :(literal), como em :(glob)*.c.

GIT_LITERAL_PATHSPECS desabilita ambos os comportamentos acima; nenhum caractere curinga funcionará e os prefixos de substituição também serão desativados.

GIT_ICASE_PATHSPECS define todos os pathspecs para funcionar de maneira que não diferencie maiúsculas de minúsculas.

Comprometendo

A criação final de um objeto Git commit geralmente é feita por git-commit-tree, que usa essas variáveis de ambiente como sua principal fonte de informações, retornando aos valores de configuração somente se eles não estiverem presentes.

GIT_AUTHOR_NAME é o nome legível por humanos no campo "autor".

GIT_AUTHOR_EMAIL é o e-mail para o campo "autor".

GIT_AUTHOR_DATE é o timestamp usado para o campo "autor".

GIT_COMMITTER_NAME define o nome humano para o campo "committer".

GIT_COMMITTER_EMAIL é o endereço de e-mail do campo "committer".

GIT_COMMITTER_DATE é usado para o timestamp no campo "committer".

EMAIL é o endereço de e-mail alternativo caso o valor de configuração user.email não esteja definido. Se *isso* não for definido, o Git retornará aos nomes de usuário e host do sistema.

rede

O Git usa a biblioteca curl para fazer operações de rede em HTTP, então **GIT_CURL_VERBOSE** diz ao Git para emitir todas as mensagens geradas por essa biblioteca. Isso é semelhante a fazer curl -v na linha de comando.

GIT_SSL_NO_VERIFY diz ao Git para não verificar os certificados SSL. Às vezes, isso pode ser necessário se você estiver

Machine Translated by Google

usando um certificado autoassinado para servir repositórios Git em HTTPS, ou você está no meio da configuração de um servidor Git, mas ainda não instalou um certificado completo.

Se a taxa de dados de uma operação HTTP for inferior a **GIT_HTTP_LOW_SPEED_LIMIT** bytes por segundo por mais de **GIT_HTTP_LOW_SPEED_TIME** segundos, o Git interromperá essa operação. Esses valores substituem os valores de configuração http.lowSpeedLimit e http.lowSpeedTime .

GIT_HTTP_USER_AGENT define a string do agente do usuário usada pelo Git ao se comunicar por HTTP. O padrão é um valor como git/2.0.0.

Diferenciar e Mesclar

GIT_DIFF_OPTS é um pouco impróprio. Os únicos valores válidos são -u<n> ou --unified=<n>, que controla o número de linhas de contexto mostradas em um comando git diff .

GIT_EXTERNAL_DIFF é usado como uma substituição para o valor de configuração diff.external . Se estiver definido, o Git invocará este programa quando git diff for invocado.

GIT_DIFF_PATH_COUNTER e **GIT_DIFF_PATH_TOTAL** são úteis dentro do programa especificado por GIT_EXTERNAL_DIFF ou diff.external. O primeiro representa qual arquivo em uma série está sendo diferenciado (começando com 1) e o último é o número total de arquivos no lote.

GIT_MERGE_VERBOSITY controla a saída para a estratégia de mesclagem recursiva. Os valores permitidos são os seguintes:

- 0 não produz nada, exceto possivelmente uma única mensagem de erro.
- 1 mostra apenas conflitos.
- 2 também mostra alterações de arquivo.
- 3 mostra quando os arquivos são ignorados porque não foram alterados.
- 4 mostra todos os caminhos à medida que são processados.
- 5 e acima mostram informações detalhadas de depuração.

O valor padrão é 2.

Depurando

Quer *realmente* saber o que o Git está fazendo? O Git possui um conjunto bastante completo de rastreamentos incorporados e tudo o que você precisa fazer é ativá-los. Os possíveis valores dessas variáveis são os seguintes:

- “true”, “1” ou “2” – a categoria de rastreamento é gravada em stderr.
- Um caminho absoluto começando com / – a saída do rastreamento será gravada nesse arquivo.

GIT_TRACE controla rastreamentos gerais, que não se enquadram em nenhuma categoria específica. Isso inclui a expansão de aliases e delegação a outros subprogramas.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554                    rastreamento: exec: 'git-lga'
```

Machine Translated by Google

```
20:12:49.878369 run-command.c:341                    trace: run_command: 'git-lga' trace:
20:12:49.879529 git.c:282 '--pretty=oneline'         alias expansion: lga => 'log' '--graph'
'--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' trace: exec: 'less'

20:12:49.899675 executar-comando.c:192
```

GIT_TRACE_PACK_ACCESS controla o rastreamento do acesso ao packfile. O primeiro campo é o packfile que está sendo acessado, o segundo é o deslocamento dentro desse arquivo:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088 20:10:12.081886 .git/objects/pack/pack-c3fa...291e.pack 12 .git/objects/
sha1_file.c:2088 20:10:12.082115 sha1_file.c:2088 pack/pack-c3fa...291e.pack 34662 .git/objects/pack/pack-
# […] 20:10:12.087398 sha1_file.c:208.c:          c3fa...291e.pack 35175
56914983 20:10:12.087419 sha1_file.c:2088
14303666                                            .git/objects/pack/pack-e80e...e3d2.pack

                                                    .git/objects/pack/pack-e80e...e3d2.pack

No mestre da filial
Sua filial está atualizada com 'origem/mestre'. nada para
confirmar, diretório de trabalho limpo
```

GIT_TRACE_PACKET permite rastreamento em nível de pacote para operações de rede.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46 pacote: pack 20:15:14.867071          git< # service=git-upload
pkt-line.c:46 pacote: git< 0000 20:15:14.867079 pkt-line.
c:46 git< pacote: 97b8860c071898d9e162678ea1035ec0c0f8b11 multi_ack thin-pack side-
band side band-64k ofs-delta raso no-progress incluir-tag multi_ack_detailed symref=HEAD:refs/heads/master
agent=git/2.0. 4 20:15:14.867088 pkt-line.c:46 pacote: 0f20ae29889d61f2e93ae00fd34f1cdb53285702
refs/heads/ab/add-interactive-show-diff-func

                                                    git<

nome
20:15:14.867094 pkt-line.c:46                       pacote:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config # […]
```

GIT_TRACE_PERFORMANCE controla o registro de dados de desempenho. A saída mostra quanto tempo leva cada invocação específica do git .

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414 'pack-refs' '--all'     desempenho: 0,374835000 s: comando git: 'git'
'--prune'
```

Machine Translated by Google

```
20:18:19.845585 trace.c:414                    desempenho: 0,343020000 s: comando git: 'git'
'reflog' 'expirar' '--all'
Contando objetos: 170994, concluído.
Compressão Delta usando até 8 threads.
Comprimindo objetos: 100% (43413/43413), concluído.
Objetos de escrita: 100% (170994/170994), concluído.
Total 170994 (delta 126176), reutilizado 170524 (delta 125706)
20:18:23.567927 trace.c:414 desempenho: 3,715349000 s: git                    command: 'git' 'pack-objects' '--keep-true-parents' '-
-honor-pack-keep' '--non-vazio' '--all' '-- reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-
offset ' '.git/objects/pack/.tmp-49190-pack' 20:18:23.584728 trace.c:414 'prune-packed' 20:18:23.605218
trace.c:414 'update-server-info' 20: 18:23.606342 trace.c:414 'repack' '-d' '-l' '-A' '--unpack-
unreachable=2.weeks.ago'                       desempenho: 0,000910000 s: comando git: 'git'

                                               desempenho: 0,017972000 s: comando git: 'git'

                                               desempenho: 3,756312000 s: comando git: 'git'

Verificando a conectividade: 170994,
concluído. 20:18:25.225424 trace.c:414         desempenho: 1,616423000 s: comando git: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414 'rerere' 'gc'      desempenho: 0,001051000 s: comando git: 'git'
20:18:25.233159 trace.c: 414 'gc'
                                               desempenho: 6,112217000 s: git comando: 'git'
```

GIT_TRACE_SETUP mostra informações sobre o que o Git está descobrindo sobre o repositório e o ambiente com o qual está interagindo.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315                    setup: git_dir: .git setup:
20:19:47.087184 trace.c:316                    worktree: /Users/ben/src/git setup: cwd: /
20:19:47.087191 trace.c:317                    Users/ben/src/git setup: prefix: (null)
20:19:47.087194 trace.c:318 No
branch master Seu branch está
atualizado com 'origin/master'. nada para confirmar,
diretório de trabalho limpo
```

Diversos

GIT_SSH, se especificado, é um programa chamado em vez de ssh quando o Git tenta se conectar a um host SSH. Ele é invocado como $GIT_SSH [username@]host [-p <port>] <command>. Observe que esta não é a maneira mais fácil de personalizar como o ssh é invocado; ele não suportará parâmetros extras de linha de comando, então você teria que escrever um script wrapper e definir GIT_SSH para apontar para ele. Provavelmente é mais fácil usar o arquivo ~/.ssh/config para isso.

GIT_ASKPASS é uma substituição para o valor de configuração core.askpass . Este é o programa invocado sempre que o Git precisa pedir credenciais ao usuário, que pode esperar um prompt de texto como um argumento de linha de comando e deve retornar a resposta no stdout (consulte Armazenamento de Credenciais para saber mais sobre isso

Machine Translated by Google

subsistema).

GIT_NAMESPACE controla o acesso a referências de namespace e é equivalente ao sinalizador --namespace . Isso é útil principalmente no lado do servidor, onde você pode querer armazenar várias bifurcações de um único repositório em um único repositório, mantendo apenas as referências separadas.

GIT_FLUSH pode ser usado para forçar o Git a usar E/S sem buffer ao gravar incrementalmente em stdout. Um valor de 1 faz com que o Git seja liberado com mais frequência, um valor de 0 faz com que toda a saída seja armazenada em buffer. O valor padrão (se esta variável não estiver definida) é escolher um esquema de buffer apropriado, dependendo da atividade e do modo de saída.

GIT_REFLOG_ACTION permite que você especifique o texto descritivo escrito no reflog. Aqui está um exemplo:

```
$ GIT_REFLOG_ACTION="minha ação" git commit --allow-empty -m 'Minha
mensagem' [mestre 9e3d55a] Minha mensagem $ git reflog -1 9e3d55a HEAD@{0}: minha
ação: Minha mensagem
```

Resumo

Neste ponto, você deve ter um bom entendimento do que o Git faz em segundo plano e, até certo ponto, como ele é implementado. Este capítulo cobriu vários comandos de encanamento — comandos que são de nível inferior e mais simples do que os comandos de porcelana que você aprendeu no resto do livro. Entender como o Git funciona em um nível mais baixo deve facilitar a compreensão de por que ele está fazendo o que está fazendo e também para escrever suas próprias ferramentas e scripts auxiliares para fazer seu fluxo de trabalho específico funcionar para você.

O Git como um sistema de arquivos endereçável por conteúdo é uma ferramenta muito poderosa que você pode usar facilmente como mais do que apenas um VCS. Esperamos que você possa usar seu novo conhecimento interno do Git para implementar sua própria aplicação legal dessa tecnologia e se sentir mais confortável usando o Git de maneiras mais avançadas.

Machine Translated by Google

Apêndice A: Git em outros ambientes

Se você leu todo o livro, aprendeu muito sobre como usar o Git na linha de comando.
Você pode trabalhar com arquivos locais, conectar seu repositório a outros em uma rede e trabalhar efetivamente com outros. Mas a história não termina aí; O Git geralmente é usado como parte de um ecossistema maior, e o terminal nem sempre é a melhor maneira de trabalhar com ele. Agora vamos dar uma olhada em alguns dos outros tipos de ambientes em que o Git pode ser útil e como outros aplicativos (incluindo o seu) funcionam junto com o Git.

Interfaces Gráficas

O ambiente nativo do Git está no terminal. Novos recursos aparecem lá primeiro, e apenas na linha de comando é que todo o poder do Git está à sua disposição. Mas o texto simples não é a melhor escolha para todas as tarefas; às vezes, uma representação visual é o que você precisa, e alguns usuários se sentem muito mais confortáveis com uma interface de apontar e clicar.

É importante observar que diferentes interfaces são adaptadas para diferentes fluxos de trabalho. Alguns clientes expõem apenas um subconjunto cuidadosamente selecionado da funcionalidade do Git, a fim de oferecer suporte a uma forma específica de trabalho que o autor considera eficaz. Quando vistas sob esta luz, nenhuma dessas ferramentas pode ser chamada de “melhor” do que qualquer uma das outras, elas simplesmente são mais adequadas para o propósito a que se destinam. Observe também que não há nada que esses clientes gráficos possam fazer que o cliente de linha de comando não possa; a linha de comando ainda é onde você terá mais poder e controle ao trabalhar com seus repositórios.

gitk e git-gui

Ao instalar o Git, você também obtém suas ferramentas visuais, gitk e git-gui.

gitk é um visualizador de histórico gráfico. Pense nisso como um poderoso shell de GUI sobre git log e git grep.
Esta é a ferramenta a ser usada quando você está tentando encontrar algo que aconteceu no passado ou visualizar o histórico do seu projeto.

O Gitk é mais fácil de invocar a partir da linha de comando. Apenas cd em um repositório Git e digite:

```
$ gitk [opções de log do git]
```

O Gitk aceita muitas opções de linha de comando, a maioria das quais são passadas para a ação git log subjacente. Provavelmente uma das mais úteis é a flag --all , que diz ao gitk para mostrar os commits acessíveis a partir de *qualquer* ref, não apenas do HEAD. A interface do Gitk fica assim:

Machine Translated by Google

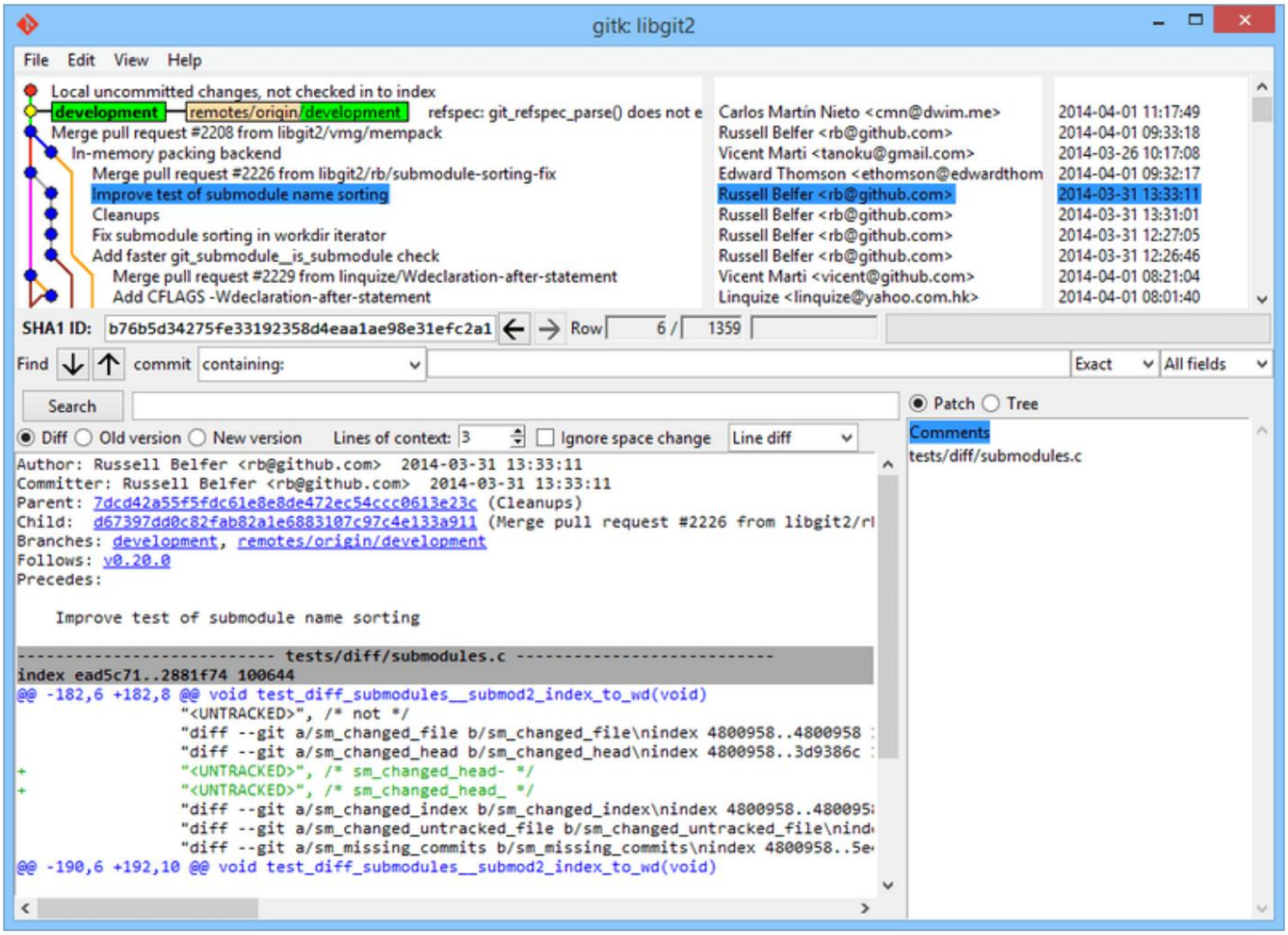


Figura 151. O visualizador de histórico gitk

No topo está algo que se parece um pouco com a saída de git log --graph; cada ponto representa um commit, as linhas representam os relacionamentos pai e as referências são mostradas como caixas coloridas. O ponto amarelo representa o HEAD e o ponto vermelho representa as alterações que ainda não foram confirmadas. Na parte inferior está uma visualização do commit selecionado; os comentários e o patch à esquerda e uma visão resumida à direita. No meio está uma coleção de controles usados para pesquisar o histórico.

O git-gui, por outro lado, é principalmente uma ferramenta para criar commits. Também é mais fácil invocar a partir da linha de comando:

```
$ git gui
```

E parece algo assim:

Machine Translated by Google

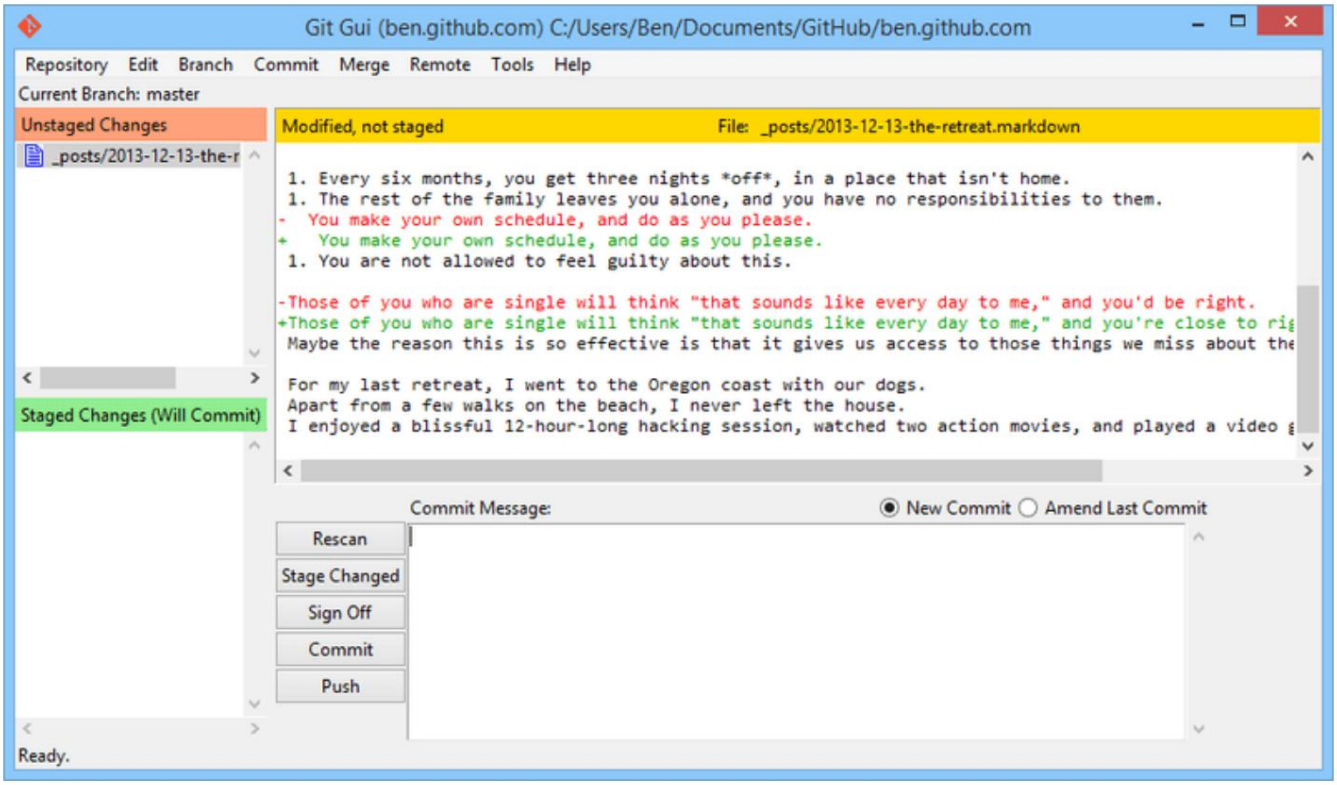


Figura 152. A ferramenta git-gui *commit*

À esquerda está o índice; as mudanças não preparadas estão na parte superior, as mudanças preparadas na parte inferior. Você pode mover arquivos inteiros entre os dois estados clicando em seus ícones ou pode selecionar um arquivo para visualização clicando em seu nome.

No canto superior direito está a exibição diff, que mostra as alterações para o arquivo atualmente selecionado. Você pode preparar pedaços individuais (ou linhas individuais) clicando com o botão direito nesta área.

No canto inferior direito está a área de mensagem e ação. Digite sua mensagem na caixa de texto e clique em “Commit” para fazer algo semelhante ao git commit. Você também pode optar por corrigir o último commit selecionando o botão de opção “Amend”, que atualizará a área “Staged Changes” com o conteúdo do último commit. Em seguida, você pode simplesmente preparar ou remover algumas alterações, alterar a mensagem do commit e clicar em “Commit” novamente para substituir o commit antigo por um novo.

gitk e git-gui são exemplos de ferramentas orientadas a tarefas. Cada um deles é adaptado para uma finalidade específica (visualizar histórico e criar commits, respectivamente) e omitir os recursos desnecessários para essa tarefa.

GitHub para macOS e Windows

O GitHub criou dois clientes Git orientados ao fluxo de trabalho: um para Windows e outro para macOS. Esses clientes são um bom exemplo de ferramentas orientadas ao fluxo de trabalho – em vez de expor *toda* a funcionalidade do Git, eles se concentram em um conjunto curado de recursos comumente usados que funcionam bem juntos. Eles se parecem com isso:

Machine Translated by Google

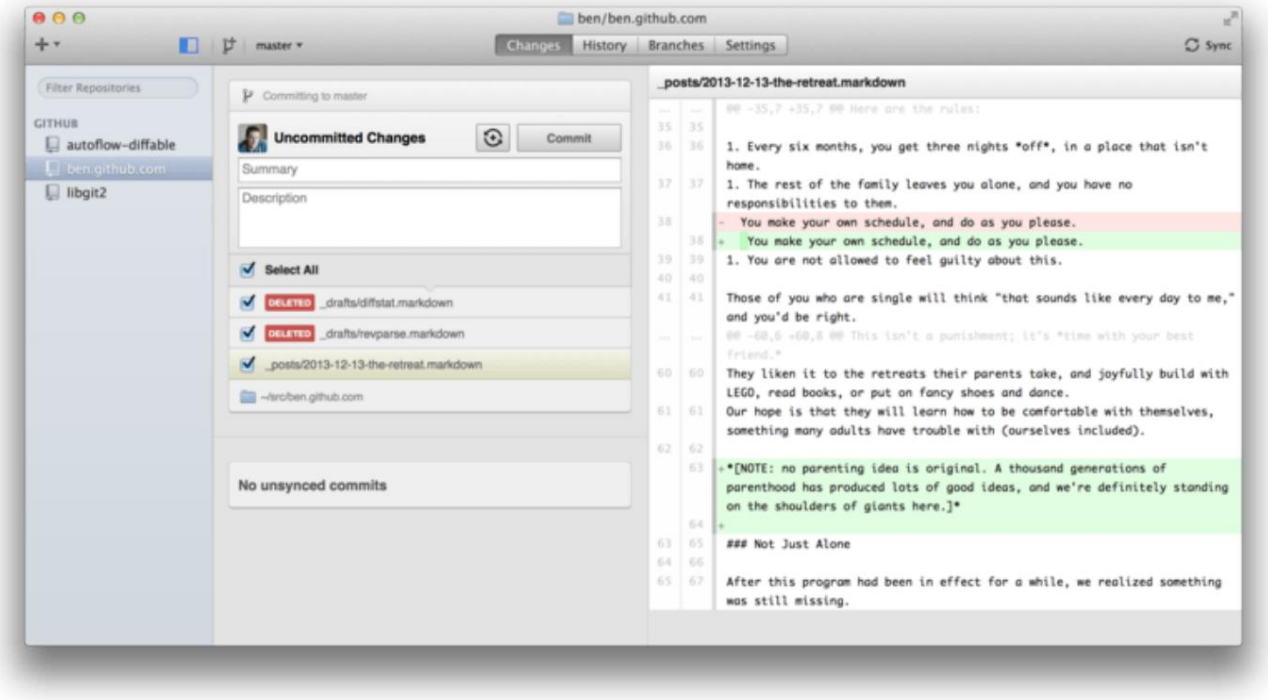


Figura 153. GitHub para macOS

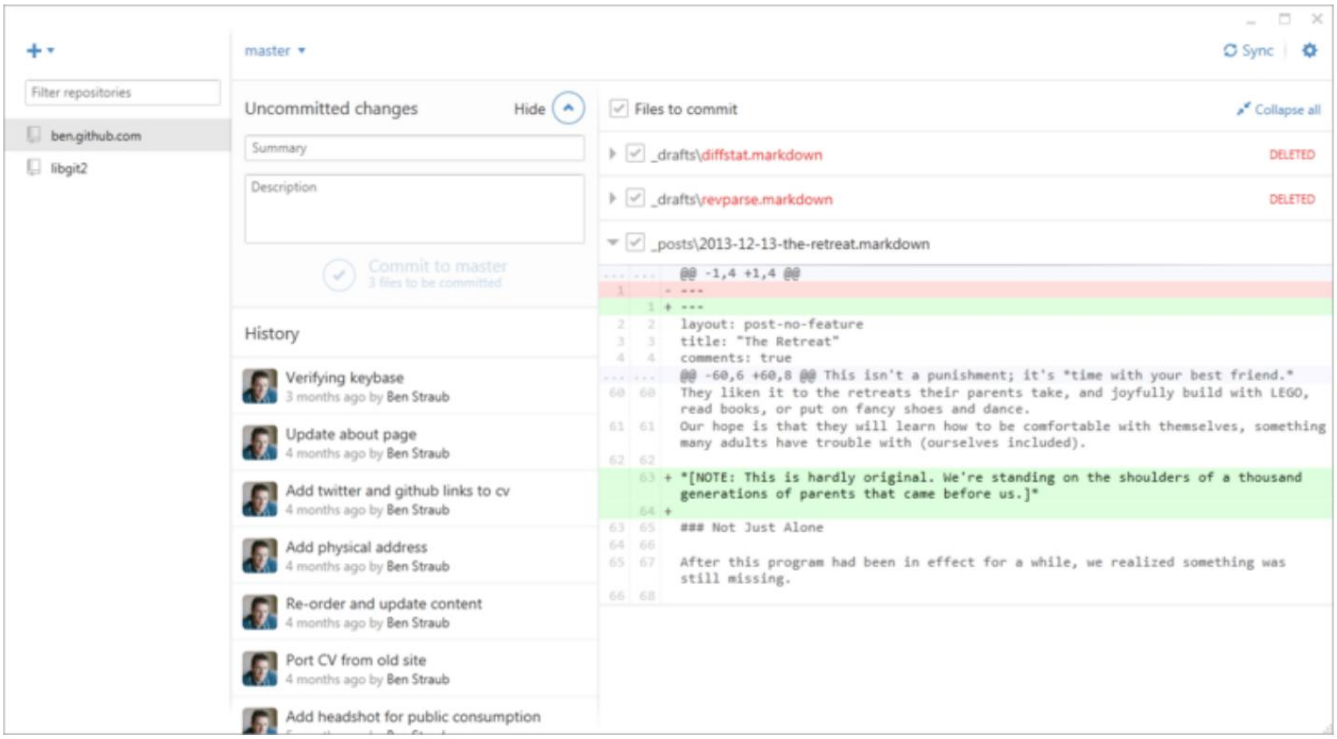


Figura 154. GitHub para Windows

Eles são projetados para parecerem e funcionarem de forma muito parecida, então vamos tratá-los como um único produto neste capítulo. Não faremos um resumo detalhado dessas ferramentas (elas têm sua própria documentação), mas um rápido tour pela visualização de “alterações” (que é onde você passará a maior parte do tempo) é necessário.

- À esquerda está a lista de repositórios que o cliente está rastreando; você pode adicionar um repositório (por clonagem ou anexação local) clicando no ícone “+” na parte superior desta área.
- No centro há uma área de entrada de confirmação, que permite inserir uma mensagem de confirmação e selecionar quais arquivos devem ser incluídos. No Windows, o histórico de confirmação é exibido logo abaixo disso; sobre

Machine Translated by Google

macOS, está em uma guia separada.

- À direita está uma visualização diff, que mostra o que foi alterado em seu diretório de trabalho ou quais alterações foram incluídas no commit selecionado.
- A última coisa a observar é o botão "Sincronizar" no canto superior direito, que é a principal forma de interagir pela rede.

ÿ

Você não precisa de uma conta GitHub para usar essas ferramentas. Embora sejam projetados para destacar o serviço do GitHub e o fluxo de trabalho recomendado, eles funcionarão com prazer com qualquer repositório e farão operações de rede com qualquer host Git.

Instalação

O GitHub para Windows pode ser baixado em https://windows.github.com, e GitHub para macOS em https://mac.github.com. Quando os aplicativos são executados pela primeira vez, eles orientam você em toda a configuração inicial do Git, como configurar seu nome e endereço de e-mail, e ambos definem padrões sensatos para muitas opções de configuração comuns, como caches de credenciais e comportamento CRLF.

Ambos são “evergreen” – as atualizações são baixadas e instaladas em segundo plano enquanto os aplicativos estão abertos. Isso inclui uma versão integrada do Git, o que significa que você provavelmente não terá que se preocupar em atualizá-lo manualmente novamente. No Windows, o cliente inclui um atalho para iniciar o PowerShell com o Posh-git, sobre o qual falaremos mais adiante neste capítulo.

A próxima etapa é fornecer à ferramenta alguns repositórios para trabalhar. O cliente mostra uma lista dos repositórios aos quais você tem acesso no GitHub e pode cloná-los em uma única etapa. Se você já possui um repositório local, basta arrastar seu diretório do Finder ou do Windows Explorer para a janela do cliente GitHub e ele será incluído na lista de repositórios à esquerda.

Fluxo de trabalho recomendado

Depois de instalado e configurado, você pode usar o cliente GitHub para muitas tarefas comuns do Git. O fluxo de trabalho pretendido para essa ferramenta às vezes é chamado de “GitHub Flow”. Cobrimos isso com mais detalhes no The GitHub Flow, mas a essência geral é que (a) você estará se comprometendo com uma ramificação e (b) estará sincronizando com um repositório remoto regularmente.

O gerenciamento de filiais é uma das áreas em que as duas ferramentas divergem. No macOS, há um botão na parte superior da janela para criar uma nova ramificação:

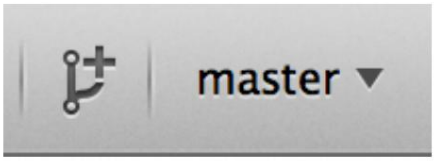


Figura 155. Botão “Create Branch” no macOS

No Windows, isso é feito digitando o nome da nova ramificação no widget de troca de ramificação:

Machine Translated by Google

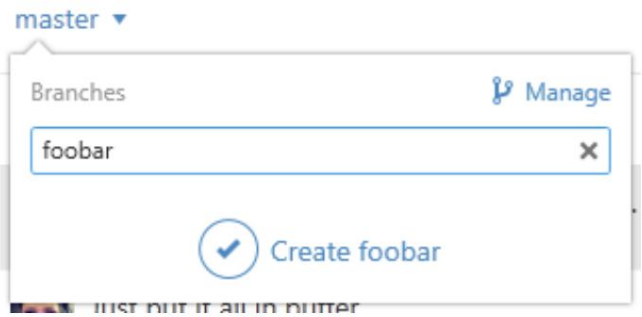


Figura 156. Criando uma ramificação no Windows

Depois que sua ramificação é criada, fazer novos commits é bastante simples. Faça algumas alterações em seu diretório de trabalho e, ao alternar para a janela do cliente GitHub, ele mostrará quais arquivos foram alterados. Digite uma mensagem de confirmação, selecione os arquivos que deseja incluir e clique no botão “Commit” (ctrl-enter ou ÿ-enter).

A principal maneira de interagir com outros repositórios na rede é por meio do recurso “Sincronizar”. O Git internamente tem operações separadas para push, fetching, merge e rebase, mas os clientes do GitHub agrupam tudo isso em um recurso de várias etapas. Veja o que acontece quando você clica no botão Sincronizar:

1. git pull --rebase. Se isso falhar devido a um conflito de mesclagem, volte para git pull --no-rebase.
2. git push.

Essa é a sequência mais comum de comandos de rede ao trabalhar nesse estilo, portanto, esmagá-los em um único comando economiza muito tempo.

Resumo

Essas ferramentas são muito adequadas para o fluxo de trabalho para o qual foram projetadas. Desenvolvedores e não desenvolvedores podem colaborar em um projeto em minutos, e muitas das práticas recomendadas para esse tipo de fluxo de trabalho são incorporadas às ferramentas. No entanto, se seu fluxo de trabalho for diferente ou se você quiser mais controle sobre como e quando as operações de rede são realizadas, recomendamos usar outro cliente ou a linha de comando.

Outras GUIs

Existem vários outros clientes gráficos do Git, e eles vão desde ferramentas especializadas e de propósito único até aplicativos que tentam expor tudo o que o Git pode fazer. O site oficial do Git tem uma lista selecionada dos clientes mais populares em https://git-scm.com/downloads/guis. Uma lista mais abrangente está disponível no site Git wiki, em https://git.wiki.kernel.org/index.php/ Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git no Visual Studio

O Visual Studio tem ferramentas Git incorporadas diretamente no IDE, começando com o Visual Studio 2019 versão 16.8.

As ferramentas suportam a seguinte funcionalidade do Git:

Machine Translated by Google

- Criar ou clonar um repositório.
- Abra e navegue pelo histórico de um repositório.
- Criar e verificar filiais e tags.
- Armazenar, preparar e confirmar alterações.
- Busca, pull, push ou sincronização de commits.
- Mesclar e rebasear ramificações.
- Resolver conflitos de mesclagem.
- Exibir diferenças.
- … e mais!

Leia a documentação oficial aprender mais.

Git no Visual Studio Code

O Visual Studio Code possui suporte integrado ao git. Você precisará ter o git versão 2.0.0 (ou mais recente) instalado.

As principais características são:

- Veja o diff do arquivo que você está editando na sarjeta.
- A barra de status do Git (canto inferior esquerdo) mostra a ramificação atual, indicadores sujos, entrada e confirmações de saída.
- Você pode fazer as operações git mais comuns de dentro do editor:
  - ÿ Inicializar um repositório.
  - ÿ Clone um repositório.
  - ÿ Crie ramificações e tags.
  - ÿ Preparar e confirmar alterações.
  - ÿ Push/pull/sync com uma ramificação remota.
  - ÿ Resolver conflitos de mesclagem.
  - ÿ Exibir diferenças.
- Com uma extensão, você pode também lidar com o GitHub Pull Pedidos: https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github.

A documentação oficial pode ser encontrada aqui: https://code.visualstudio.com/Docs/editor/versioncontrol .

Git em IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine

Os IDEs JetBrains (como IntelliJ IDEA, PyCharm, WebStorm, PhpStorm, RubyMine e outros) são fornecidos com um plug-in Git Integration. Ele fornece uma exibição dedicada no IDE para trabalhar com Git e GitHub

Machine Translated by Google

Solicitações de pull.

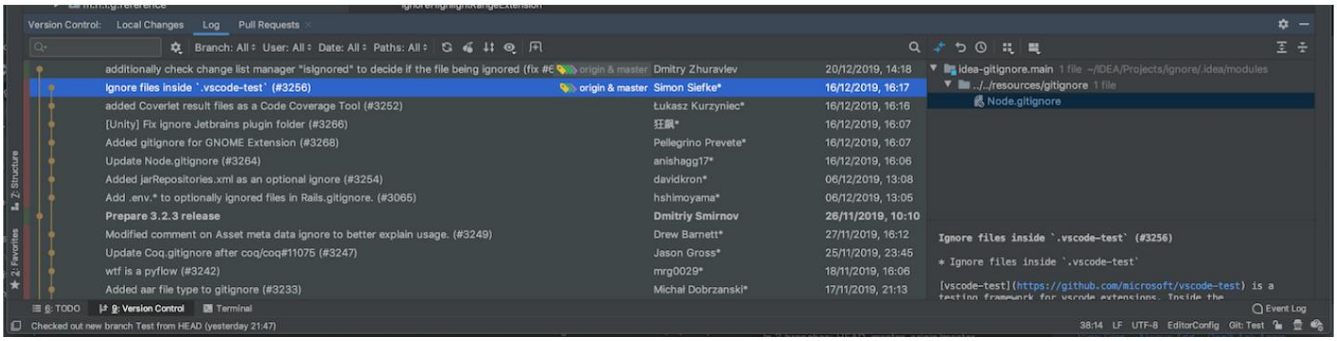


Figura 157. Version Control ToolWindow em JetBrains IDEs

A integração depende do cliente git de linha de comando e requer a instalação de um. A documentação oficial está disponível em https://www.jetbrains.com/help/idea/using-git-integration.html.

Git em Texto Sublime

A partir da versão 3.2, o Sublime Text possui integração com o git no editor.

As características são:

- A barra lateral mostrará o status git de arquivos e pastas com um distintivo/ícone.
- Arquivos e pastas que estão em seu arquivo .gitignore ficarão esmaecidos na barra lateral.
- Na barra de status, você pode ver a ramificação git atual e quantas modificações você tem feito.
- Todas as alterações em um arquivo agora são visíveis por meio de marcadores na medianiz.
- Você pode usar parte da funcionalidade do cliente Sublime Merge git de dentro do Sublime Text. Esse requer que o Sublime Merge esteja instalado. Veja: https://www.sublimemerge.com/.

A documentação oficial do Sublime Text pode ser encontrada aqui: https://www.sublimetext.com/docs/3/git_integration.html .

Git no Bash

Se você for um usuário do Bash, poderá explorar alguns dos recursos do seu shell para tornar sua experiência com o Git muito mais amigável. Na verdade, o Git vem com plug-ins para vários shells, mas não é ativado por padrão.

Primeiro, você precisa obter uma cópia do arquivo de conclusões do código-fonte da versão do Git que está usando. Verifique sua versão digitando git version e use git checkout tags/vX.YZ, onde vX.YZ corresponde à versão do Git que você está usando. Copie o arquivo contrib/completion/git-completion.bash em algum lugar acessível, como seu diretório pessoal, e adicione-o ao seu .bashrc:

```
. ~/git-completion.bash
```

Feito isso, altere seu diretório para um repositório Git e digite:

Machine Translated by Google

```
$ git check<tab>
```

…e o Bash irá completar automaticamente para git checkout. Isso funciona com todos os subcomandos do Git, parâmetros de linha de comando e remotos e nomes de referência quando apropriado.

Também é útil personalizar seu prompt para mostrar informações sobre o repositório Git do diretório atual. Isso pode ser tão simples ou complexo quanto você quiser, mas geralmente há algumas informações importantes que a maioria das pessoas deseja, como o branch atual e o status do diretório de trabalho. Para adicioná-los ao seu prompt, apenas copie o arquivo contrib/completion/git-prompt.sh do repositório de origem do Git para o seu diretório pessoal, adicione algo assim ao seu .bashrc:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

O \w significa imprimir o diretório de trabalho atual, o \$ imprime a parte $ do prompt e __git_ps1 " (%s)" chama a função fornecida pelo arquivo git-prompt.sh com um argumento de formatação. Com isso, seu prompt Bash ficará assim quando você estiver em qualquer diretório de um projeto Git:

Figura 158. Prompt bash personalizado

Ambos os scripts vêm com documentação útil; dê uma olhada no conteúdo de git length.bash e git-prompt.sh para mais informações.

Git em zsh

O Zsh também vem com uma biblioteca de preenchimento de tabulação para Git. Para usá-lo, simplesmente execute autoload -Uz compinit && compinit em seu .zshrc. A interface do Zsh é um pouco mais poderosa que a do Bash:

```
$ git che<tab>
check-attr          -- exibir informações gitattributes
check-ref-format -- garante que um nome de referência esteja bem formado checkout
checkout-index      -- ramificação de check-out ou caminhos para a árvore
                    de trabalho -- copia arquivos do índice para o diretório de
palito              trabalho -- encontra commits não mesclados upstream --
de cereja           aplica alterações introduzidas por alguns commits existentes
```

Machine Translated by Google

Preenchimentos de tabulação ambíguos não são apenas listados; eles têm descrições úteis e você pode navegar graficamente na lista pressionando repetidamente a guia. Isso funciona com comandos do Git, seus argumentos e nomes de coisas dentro do repositório (como refs e remotos), bem como nomes de arquivos e todas as outras coisas que o Zsh sabe como completar com tabulação.

O Zsh vem com uma estrutura para obter informações de sistemas de controle de versão, chamada vcs_info. Para incluir o nome da ramificação no prompt à direita, adicione estas linhas ao seu arquivo ~/.zshrc :

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT='${vcs_info_msg_0_}'
# PROMPT='${vcs_info_msg_0_}%# '
zstyle ':vcs_info:git:*' formatos '%b'
```

Isso resulta em uma exibição da ramificação atual no lado direito da janela do terminal, sempre que seu shell estiver dentro de um repositório Git. O lado esquerdo também é suportado, é claro; apenas descomente a atribuição para PROMPT. Parece um pouco com isso:

Figura 159. Prompt zsh personalizado

Para obter mais informações sobre vcs_info, verifique sua documentação na página de manual zshcontrib(1) ou on-line em http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control Information.

Em vez de vcs_info, você pode preferir o script de personalização de prompt que acompanha o Git, chamado git-prompt.sh; consulte https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh para

Machine Translated by Google

detalhes. git-prompt.sh é compatível com Bash e Zsh.

O Zsh é poderoso o suficiente para que existam estruturas inteiras dedicadas a torná-lo melhor. Um deles se chama "oh-my-zsh" e pode ser encontrado em https://github.com/robbyrussell/oh-my-zsh. O sistema de plug-in do oh-my-zsh vem com um poderoso preenchimento de guia git e possui uma variedade de "temas" de prompt, muitos dos quais exibem dados de controle de versão. Um exemplo de tema oh-my-zsh é apenas um exemplo do que pode ser feito com este sistema.

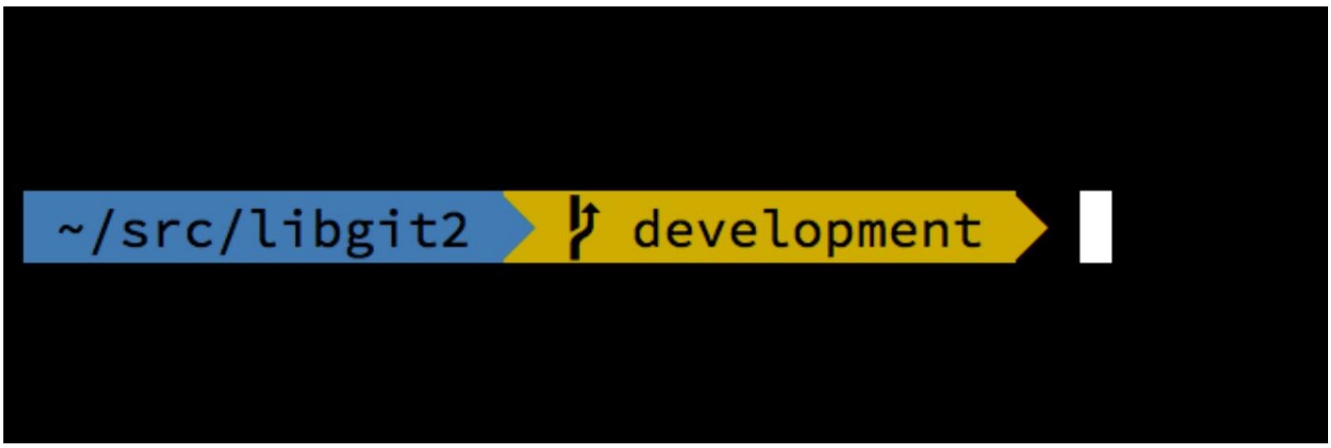


Figura 160. Um exemplo de um tema oh-my-zsh

Git no PowerShell

O terminal de linha de comando legado no Windows (cmd.exe) não é realmente capaz de uma experiência Git personalizada, mas se você estiver usando o PowerShell, está com sorte. Isso também funciona se você estiver executando o PowerShell Core no Linux ou macOS. Um pacote chamado posh-git (https://github.com/dahlbyk/posh-git) fornece recursos avançados de preenchimento de guias, bem como um prompt aprimorado para ajudá-lo a ficar por dentro do status do seu repositório. Se parece com isso:

Figura 161. PowerShell com Posh-git

Instalação

Pré-requisitos (somente Windows)

Antes de poder executar scripts do PowerShell em sua máquina, você precisa definir sua ExecutionPolicy local como RemoteSigned (basicamente, qualquer coisa, exceto Undefined e Restricted). Se você escolher

Machine Translated by Google

AllSigned em vez de RemoteSigned, também scripts locais (seus próprios) precisam ser assinados digitalmente para serem executados. Com RemoteSigned, apenas os scripts com o ZoneIdentifier configurado para Internet (foram baixados da web) precisam ser assinados, outros não. Se você for um administrador e quiser configurá-lo para todos os usuários dessa máquina, use -Scope LocalMachine. Se você é um usuário normal, sem direitos administrativos, pode usar -Scope CurrentUser para configurá-lo apenas para você.

Mais sobre https://docs.microsoft.com/en-us/powershell/module/ microsoft.powershell.core/about/about_scopes.

Escopos do PowerShell

Mais sobre PowerShell ExecutionPolicy: https://docs.microsoft.com/en-us/powershell/module/ microsoft.powershell.security/set-executionpolicy.

Para definir o valor de ExecutionPolicy para RemoteSigned para todos os usuários, use o próximo comando:

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

Galeria do PowerShell

Se você tiver pelo menos o PowerShell 5 ou PowerShell 4 com PackageManagement instalado, poderá usar o gerenciador de pacotes para instalar o posh-git para você.

Mais informações sobre a Galeria do PowerShell: https://docs.microsoft.com/en-us/powershell/scripting/galeria/overview .

```
> Install-Module posh-git -Scope CurrentUser -Force
> Install-Module posh-git -Scope CurrentUser -AllowPrerelease -Force # Versão beta mais recente com suporte ao PowerShell Core
```

Se você deseja instalar o posh-git para todos os usuários, use -Scope AllUsers e execute o comando em um console elevado do PowerShell. Se o segundo comando falhar com um erro como Módulo 'PowerShellGet' não foi instalado usando Install-Module, você precisará executar outro comando primeiro:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

Então você pode voltar e tentar novamente. Isso acontece porque os módulos fornecidos com o Windows PowerShell são assinados com um certificado de publicação diferente.

Atualizar prompt do PowerShell

Para incluir informações do git em seu prompt, o módulo posh-git precisa ser importado. Para importar o posh git sempre que o PowerShell for iniciado, execute o comando Add-PoshGitToProfile , que adicionará a instrução de importação ao seu script $profile . Este script é executado sempre que você abre um novo console do PowerShell. Lembre-se de que existem vários scripts $profile . Por exemplo. um para o console e outro separado para o ISE.

Machine Translated by Google

```
> Módulo de importação posh-git
> Add-PoshGitToProfile -AllHosts
```

Da fonte

Basta baixar uma versão posh-git de https://github.com/dahlbyk/posh-git/releases, e descompacte-o. Em seguida, importe o módulo usando o caminho completo para o arquivo posh-git.psd1 :

```
> Import-Module <path-to-uncompress-folder>\src\posh-git.psd1 > Add-
PoshGitToProfile -AllHosts
```

Isso adicionará a linha adequada ao seu arquivo profile.ps1 e o posh-git estará ativo na próxima vez que você abrir o PowerShell.

Para obter uma descrição das informações resumidas do status do Git exibidas no prompt, consulte: https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-summary-information Para obter mais detalhes sobre como personalizar seu prompt posh-git, consulte: https://github.com/dahlbyk/posh-git/blob/master/README.md#customization-variables .

Resumo

Você aprendeu como aproveitar o poder do Git de dentro das ferramentas que usa durante seu trabalho diário e também como acessar repositórios Git de seus próprios programas.

Machine Translated by Google

Apêndice B: Incorporando o Git em seu Formulários

Se seu aplicativo for para desenvolvedores, é provável que ele se beneficie da integração com o controle do código-fonte. Mesmo aplicativos não-desenvolvedores, como editores de documentos, podem se beneficiar dos recursos de controle de versão, e o modelo do Git funciona muito bem para muitos cenários diferentes.

Se precisar integrar o Git ao seu aplicativo, você tem basicamente duas opções: gerar um shell e chamar o programa de linha de comando git ou incorporar uma biblioteca Git ao seu aplicativo. Aqui abordaremos a integração de linha de comando e várias das bibliotecas Git incorporáveis mais populares.

Git de linha de comando

Uma opção é gerar um processo de shell e usar a ferramenta de linha de comando Git para fazer o trabalho. Isso tem a vantagem de ser canônico e todos os recursos do Git são suportados. Isso também é bastante fácil, já que a maioria dos ambientes de tempo de execução possui um recurso relativamente simples para invocar um processo com argumentos de linha de comando. No entanto, esta abordagem tem algumas desvantagens.

Uma delas é que toda a saída está em texto simples. Isso significa que você terá que analisar o formato de saída ocasionalmente alterado do Git para ler o progresso e as informações de resultado, o que pode ser ineficiente e sujeito a erros.

Outra é a falta de recuperação de erros. Se um repositório estiver corrompido de alguma forma, ou o usuário tiver um valor de configuração malformado, o Git simplesmente se recusará a executar muitas operações.

Ainda outro é o gerenciamento de processos. O Git exige que você mantenha um ambiente de shell em um processo separado, o que pode adicionar complexidade indesejada. Tentar coordenar muitos desses processos (especialmente ao acessar potencialmente o mesmo repositório de vários processos) pode ser um grande desafio.

Libgit2

Outra opção à sua disposição é usar o Libgit2. Libgit2 é uma implementação livre de dependência do Git, com foco em ter uma boa API para uso em outros programas. Você pode encontrá-lo em https://libgit2.org.

Primeiro, vamos dar uma olhada na aparência da API C. Aqui está um tour rápido:

```
// Abre um repositório
git_repository *repo;
erro int = git_repository_open(&repo, "/caminho/para/repositório");

// Desreferencia HEAD para um
commit git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;
```

Machine Translated by Google

```
// Imprime algumas das propriedades do
commit printf("%s", git_commit_message(commit));
const git_signature *autor = git_commit_author(commit);
printf("%s <%s>\n", autor->nome, autor->e-mail); const git_oid
*tree_id = git_commit_tree_id(commit);

// Limpeza
git_commit_free(commit);
git_repository_free(repo);
```

As primeiras linhas abrem um repositório Git. O tipo git_repository representa um identificador para um repositório com um cache na memória. Este é o método mais simples, para quando você sabe o caminho exato para o diretório de trabalho de um repositório ou pasta .git . Há também o git_repository_open_ext que inclui opções para pesquisa, git_clone e amigos para fazer um clone local de um repositório remoto e git_repository_init para criar um repositório totalmente novo.

O segundo pedaço de código usa a sintaxe rev-parse (consulte Referências de ramificação para saber mais sobre isso) para obter o commit para o qual o HEAD eventualmente aponta. O tipo retornado é um ponteiro git_object , que representa algo que existe no banco de dados de objetos Git para um repositório. git_object é na verdade um tipo “pai” para vários tipos diferentes de objetos; o layout de memória para cada um dos tipos “filhos” é o mesmo que para git_object, então você pode converter com segurança para o correto. Nesse caso, git_object_type(commit) retornaria GIT_OBJ_COMMIT, então é seguro converter para um ponteiro git_commit .

A próxima parte mostra como acessar as propriedades do commit. A última linha aqui usa um tipo git_oid ; esta é a representação de Libgit2 para um hash SHA-1.

A partir desta amostra, alguns padrões começaram a surgir:

- Se você declarar um ponteiro e passar uma referência a ele em uma chamada Libgit2, essa chamada provavelmente será retornar um código de erro inteiro. Um valor 0 indica sucesso; qualquer coisa menos é um erro.
- Se Libgit2 preencher um ponteiro para você, você é responsável por liberá-lo.
- Se Libgit2 retornar um ponteiro const de uma chamada, você não precisa liberá-lo, mas ele se tornará inválido quando o objeto a que pertence é liberado.
- Escrever C é um pouco doloroso.

Esse último significa que não é muito provável que você esteja escrevendo C ao usar Libgit2. Felizmente, há várias ligações específicas de linguagem disponíveis que facilitam bastante o trabalho com repositórios Git de sua linguagem e ambiente específicos. Vamos dar uma olhada no exemplo acima escrito usando as ligações Ruby para Libgit2, que são chamadas de Rugged e podem ser encontradas em 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] }>"
árvore = commit.árvore
```

Machine Translated by Google

Como você pode ver, o código é muito menos confuso. Em primeiro lugar, o Rugged usa exceções; ele pode levantar coisas como ConfigError ou ObjectError para sinalizar condições de erro. Em segundo lugar, não há liberação explícita de recursos, já que Ruby é coletado como lixo. Vamos dar uma olhada em um exemplo um pouco mais complicado: criar um commit do zero

```
blob_id = repo.write("Conteúdo do blob", :blob) ÿ

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ÿ

sig =
{ :email => "bob@example.com", :name
=> "Bob User", :time => Time.now, }



commit_id = Rugged::Commit.create(repo, :tree
=> index.write_tree(repo), ÿ :author =>
sig, :committer => sig, ÿ :message =>
"Adicionar novo arquivo.txt", ÿ :parents =>
repo.empty? ? [] : [ repo.head.target ].compact,
ÿ :update_ref => 'HEAD', ÿ ) commit = repo.lookup(commit_id) ÿ
```

ÿ Crie um novo blob, que contém o conteúdo de um novo arquivo.

ÿ Preencha o índice com a árvore do commit principal e adicione o novo arquivo no caminho newfile.txt.

ÿ Isso cria uma nova árvore no ODB e a usa para o novo commit.

ÿ Usamos a mesma assinatura para os campos author e committer.

ÿ A mensagem de confirmação.

ÿ Ao criar um commit, você deve especificar os pais do novo commit. Isso usa a ponta de HEAD para o pai solteiro.

ÿ Rugged (e Libgit2) pode, opcionalmente, atualizar uma referência ao fazer um commit.

ÿ O valor de retorno é o hash SHA-1 de um novo objeto de confirmação, que você pode usar para obter um Confirmar objeto.

O código Ruby é bom e limpo, mas como Libgit2 está fazendo o trabalho pesado, esse código também será executado bem rápido. Se você não for um rubista, abordaremos algumas outras vinculações em Other Bindings.

Funcionalidade avançada

Libgit2 tem alguns recursos que estão fora do escopo do Git principal. Um exemplo é a capacidade de conexão: o Libgit2 permite que você forneça “back-ends” personalizados para vários tipos de operação, para que você possa armazenar coisas de uma maneira diferente do Git padrão. Libgit2 permite back-ends personalizados para

Machine Translated by Google

configuração, armazenamento de referências e banco de dados de objetos, entre outras coisas.

Vamos dar uma olhada em como isso funciona. O código abaixo é emprestado do conjunto de exemplos de back-end fornecidos pela equipe Libgit2 (que pode ser encontrado em https://github.com/libgit2/libgit2-backends).
Veja como um back-end personalizado para o banco de dados de objetos é configurado:

```
git_odb *odb;
erro int = git_odb_new(&odb); ÿ

git_odb_backend
*meu_backend; error = git_odb_backend_mine(&my_backend, /*…*/); ÿ

erro = git_odb_add_backend(odb, my_backend, 1); ÿ

git_repository *repo;
error = git_repository_open(&repo, "some-path");
erro = git_repository_set_odb(repo, odb); ÿ
```

Observe que os erros são capturados, mas não manipulados. Esperamos que seu código seja melhor que o nosso.

ÿ Inicialize um “front-end” de banco de dados de objeto (ODB) vazio, que atuará como um contêiner para o “back-ends” que são os que fazem o trabalho real.

ÿ Inicialize um back-end ODB personalizado.

ÿ Adicione o back-end ao front-end.

ÿ Abra um repositório e configure-o para usar nosso ODB para procurar objetos.

Mas o que é essa coisa de git_odb_backend_mine ? Bem, esse é o construtor para sua própria implementação ODB, e você pode fazer o que quiser nele, desde que preencha a estrutura git_odb_backend corretamente. Aqui está o que *poderia* parecer:

```
typedef struct
{ git_odb_backend pai;

// Algumas outras coisas
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));

back-end->custom_context = …;

backend->parent.read = &my_backend__read;
backend->parent.read_prefix = &my_backend__read_prefix;
backend->parent.read_header = &my_backend__read_header;
```

Machine Translated by Google

```
// …

*backend_out = (git_odb_backend *) backend;

retornar GIT_SUCCESS; }
```

A restrição mais sutil aqui é que o primeiro membro de my_backend_struct deve ser uma estrutura git_odb_backend ; isso garante que o layout da memória seja o que o código Libgit2 espera que seja. O resto é arbitrário; essa estrutura pode ser tão grande ou pequena quanto você precisar.

A função de inicialização aloca alguma memória para a estrutura, configura o contexto personalizado e, em seguida, preenche os membros da estrutura pai que ela oferece suporte. Dê uma olhada no arquivo include/git2/sys/odb_backend.h na fonte Libgit2 para um conjunto completo de assinaturas de chamada; seu caso de uso específico ajudará a determinar qual deles você deseja oferecer suporte.

Outras vinculações

Libgit2 tem ligações para muitos idiomas. Aqui mostramos um pequeno exemplo usando alguns dos pacotes de ligações mais completos até o momento; existem bibliotecas para muitas outras linguagens, incluindo C++, Go, Node.js, Erlang e JVM, todas em vários estágios de maturidade. A coleção oficial de ligações pode ser encontrada navegando nos repositórios em https://github.com/libgit2. O código que escreveremos retornará a mensagem de commit do commit eventualmente apontado por HEAD (mais ou menos como git log -1).

LibGit2Sharp

Se você estiver escrevendo um aplicativo .NET ou Mono, LibGit2Sharp (https://github.com/libgit2/libgit2sharp) é o que você está procurando. As ligações são escritas em C#, e muito cuidado foi tomado para agrupar as chamadas Libgit2 brutas com APIs CLR nativas. Aqui está a aparência do nosso programa de exemplo:

```
new Repository(@"C:\caminho\para\repo").Head.Tip.Message;
```

Para aplicativos de área de trabalho do Windows, há até mesmo um pacote NuGet que o ajudará a começar rapidamente.

objetivo-git

Se seu aplicativo estiver sendo executado em uma plataforma Apple, você provavelmente está usando Objective-C como sua linguagem de implementação. Objective-Git (https://github.com/libgit2/objective-git) é o nome das ligações Libgit2 para esse ambiente. O programa de exemplo se parece com isso:

```
GTRepository *repo =
[[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] error:NULL];

NSString *msg = [[[repo headReferenceWithError:NULL] resolvidoTarget] mensagem];
```

Machine Translated by Google

O Objective-git é totalmente interoperável com o Swift, então não tenha medo se você deixou o Objective-C para trás.

pigit2

As ligações para Libgit2 em Python são chamadas de Pygit2 e podem ser encontradas em https://www.pygit2.org.
Nosso programa de exemplo:

```
pygit2.Repository("/path/to/repo") # abre o repositório .head #
obtém o branch atual # desce até o commit
.peel(pygit2.Commit) .mensagem
                                                    # leia a mensagem
```

Leitura adicional

Obviamente, um tratamento completo dos recursos do Libgit2 está fora do escopo deste livro. Se você quiser mais informações sobre o próprio Libgit2, há documentação da API em https://libgit2.github.com/libgit2, e um conjunto de guias em https://libgit2.github.com/docs. Para as outras ligações, verifique o README e os testes incluídos; geralmente existem pequenos tutoriais e dicas para leituras adicionais.

JGitGenericName

Se você quiser usar o Git de dentro de um programa Java, existe uma biblioteca Git completa chamada JGit.
JGit é uma implementação relativamente completa do Git escrita nativamente em Java e é amplamente usada na comunidade Java. O projeto JGit está sob o guarda-chuva do Eclipse e sua página inicial pode ser encontrada em https://www.eclipse.org/jgit/.

Preparando-se

Existem várias maneiras de conectar seu projeto com o JGit e começar a escrever código nele.
Provavelmente, o mais fácil é usar o Maven – a integração é realizada adicionando o seguinte trecho à tag <dependencies> em seu arquivo pom.xml:

```
<dependency>
<groupId>org.eclipse.jgit</groupId>
<artifactId>org.eclipse.jgit</artifactId>
<version>3.5.0.201409260305-r</version>
</dependência>
```

A versão provavelmente já terá avançado quando você ler isto; verifique https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit informações do repositório. Depois que essa etapa for concluída, o Maven adquirirá automaticamente as bibliotecas JGit necessárias.

Se você preferir gerenciar as dependências binárias por conta própria, os binários JGit pré-construídos estão disponíveis em https://www.eclipse.org/jgit/download. Você pode incorporá-los ao seu projeto executando um comando como este:

Machine Translated by Google

```
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
```

Encanamento

O JGit possui dois níveis básicos de API: encanamento e porcelana. A terminologia para isso vem do próprio Git, e o JGit é dividido aproximadamente nos mesmos tipos de áreas: as APIs de porcelana são um front-end amigável para ações comuns em nível de usuário (o tipo de coisa que um usuário normal usaria a ferramenta de linha de comando Git para), enquanto as APIs de encanamento são para interagir diretamente com objetos de repositório de baixo nível.

O ponto de partida para a maioria das sessões JGit é a classe Repository , e a primeira coisa que você deseja fazer é criar uma instância dela. Para um repositório baseado em sistema de arquivos (sim, JGit permite outros modelos de armazenamento), isso é feito usando FileRepositoryBuilder:

```
// Criar um novo repositório
Repository newCreatedRepo = FileRepositoryBuilder.create( new
File("/tmp/new_repo/.git")); newCreatedRepo.create();


// Abre um repositório existente
Repository existRepo = new FileRepositoryBuilder() .setGitDir(new
File("meu_repo/.git")) .build();
```

O construtor tem uma API fluente para fornecer tudo o que precisa para encontrar um repositório Git, independentemente de seu programa saber exatamente onde está localizado. Ele pode usar variáveis de ambiente (.readEnvironment()), iniciar de um local no diretório de trabalho e pesquisar (.setWorkTree(…).findGitDir()) ou apenas abrir um diretório .git conhecido como acima.

Depois de ter uma instância de Repositório , você pode fazer todo tipo de coisa com ela. Aqui está uma amostra rápida:

```
// Obtém uma
referência Ref master = repo.getRef("master");

// Obtém os pontos de referência do objeto para
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Carrega o conteúdo bruto do
objeto ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Cria uma
ramificação RefUpdate createBranch1 = repo.updateRef("refs/heads/
branch1"); createBranch1.setNewObjectId(masterTip);
```

Machine Translated by Google

```
createBranch1.update();

// Excluir uma
ramificação RefUpdate deleteBranch1 = repo.updateRef("refs/heads/
branch1"); deleteBranch1.setForceUpdate(true); deleteBranch1.delete();


//
Configuração Configuração cfg =
repo.getConfig(); String nome = cfg.getString("usuário", null, "nome");
```

Há muita coisa acontecendo aqui, então vamos passar por uma seção de cada vez.

A primeira linha obtém um ponteiro para a referência mestre . O JGit pega automaticamente a referência master *atual* , que fica em refs/heads/master, e retorna um objeto que permite buscar informações sobre a referência. Você pode obter o nome (.getName()) e o objeto de destino de uma referência direta (.getObjectId()) ou a referência apontada por uma referência simbólica (.getTarget()). Objetos ref também são usados para representar refs e objetos de tag, então você pode perguntar se a tag está “descascada”, o que significa que ela aponta para o destino final de uma sequência (potencialmente longa) de objetos de tag.

A segunda linha obtém o destino da referência mestre , que é retornada como uma instância de ObjectId. ObjectId representa o hash SHA-1 de um objeto, que pode ou não existir no banco de dados de objetos do Git. A terceira linha é semelhante, mas mostra como o JGit lida com a sintaxe rev-parse (para saber mais sobre isso, consulte Referências de ramificação); você pode passar qualquer especificador de objeto que o Git entenda, e o JGit retornará um ObjectId válido para esse objeto ou nulo.

As próximas duas linhas mostram como carregar o conteúdo bruto de um objeto. Neste exemplo, chamamos ObjectLoader.copyTo() para transmitir o conteúdo do objeto diretamente para stdout, mas ObjectLoader também possui métodos para ler o tipo e o tamanho de um objeto, bem como retorná-lo como uma matriz de bytes. Para objetos grandes (onde .isLarge() retorna true), você pode chamar .openStream() para obter um objeto semelhante a InputStream que pode ler os dados brutos do objeto sem colocá-los todos na memória de uma vez.

As próximas linhas mostram o que é necessário para criar uma nova ramificação. Criamos uma instância RefUpdate, configuramos alguns parâmetros e chamamos .update() para acionar a alteração. Diretamente a seguir está o código para excluir esse mesmo ramo. Observe que .setForceUpdate(true) é necessário para que isso funcione; caso contrário, a chamada .delete() retornará REJECTED e nada acontecerá.

O último exemplo mostra como buscar o valor user.name dos arquivos de configuração do Git. Esta instância do Config usa o repositório que abrimos anteriormente para configuração local, mas detectará automaticamente os arquivos de configuração global e do sistema e também lerá os valores deles.

Esta é apenas uma pequena amostra da API de encanamento completa; existem muitos outros métodos e classes disponíveis. Também não é mostrado aqui o modo como o JGit lida com erros, que é por meio do uso de exceções. As APIs JGit às vezes lançam exceções Java padrão (como IOException), mas há uma série de tipos de exceção específicos do JGit que também são fornecidos (como NoRemoteRepositoryException, CorruptObjectException e NoMergeBaseException).

Machine Translated by Google

Porcelana

As APIs de encanamento são bastante completas, mas pode ser complicado reuni-las para atingir objetivos comuns, como adicionar um arquivo ao índice ou fazer um novo commit. O JGit fornece um conjunto de APIs de alto nível para ajudar nisso, e o ponto de entrada para essas APIs é a classe Git :

```
Repositório
repositório; // constrói repositório...
Git git = new Git(repo);
```

A classe Git tem um bom conjunto de métodos *de estilo de construtor* de alto nível que podem ser usados para construir algum comportamento bastante complexo. Vamos dar uma olhada em um exemplo — fazendo algo como git ls-remote:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",
"p4ssw0rd"); Collection<Ref> remoteRefs =
git.lsRemote() .setCredentialsProvider(cp) .setRemote("origem") .setTags(true) .setHeads(false) .call();
for (Ref ref : remoteRefs) { System.out.println(ref.getName() + }
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
" -> " + ref.getObjectId().name());
```

Este é um padrão comum com a classe Git; os métodos retornam um objeto de comando que permite encadear chamadas de método para definir parâmetros, que são executados quando você chama .call(). Neste caso, estamos pedindo tags ao remoto de origem , mas não cabeças. Observe também o uso de um objeto CredentialsProvider para autenticação.

Muitos outros comandos estão disponíveis por meio da classe Git, incluindo, entre outros , adicionar, culpar, confirmar, limpar, enviar, rebasear, reverter e redefinir.

Leitura adicional

Esta é apenas uma pequena amostra dos recursos completos do JGit. Se você estiver interessado e quiser saber mais, veja onde procurar informações e inspiração:

- A documentação oficial da API JGit pode ser encontrada em https://www.eclipse.org/jgit/documentation. Eles são Javadoc padrão, portanto, seu JVM IDE favorito também poderá instalá-los localmente.
- O JGit Cookbook em https://github.com/centic9/jgit-cookbook tem muitos exemplos de como fazer tarefas específicas com JGit.

go-git

Caso você queira integrar o Git a um serviço escrito em Golang, também existe uma implementação de biblioteca Go pura. Esta implementação não possui nenhuma dependência nativa e, portanto, não é propensa

Machine Translated by Google

para erros de gerenciamento manual de memória. Também é transparente para as ferramentas de análise de desempenho Golang padrão, como CPU, criadores de perfil de memória, detector de corrida, etc.

go-git é focado em extensibilidade, compatibilidade e suporta a maioria das APIs de encanamento, documentadas em https://github.com/go-git/go-git/blob/master/COMPATIBILITY.md.

Aqui está um exemplo básico de uso de APIs Go:

```
importar "github.com/go-git/go-git/v5"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{ URL:
"https://github.com/go-git/go-git", Progresso: os.Stdout, })
```

Assim que você tiver uma instância do Repository , poderá acessar informações e realizar mutações nela:

```
// recupera o ramo apontado por HEAD
ref, err := r.Head()

// obtém o objeto commit, apontado por ref
commit, err := r.CommitObject(ref.Hash())

// recupera o histórico de commits,
err := commit.History()

// itera sobre os commits e imprime cada c :=
      _,  range history { for fmt.Println(c) }
```

Funcionalidade avançada

go-git tem poucos recursos avançados notáveis, um dos quais é um sistema de armazenamento conectável, que é semelhante aos back-ends Libgit2. A implementação padrão é o armazenamento na memória, que é muito rápido.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{ URL:
"https://github.com/go-git/go-git", })
```

O armazenamento conectável oferece muitas opções interessantes. Por exemplo, https://github.com/go-git/go-git/tree/master/_examples/storage permite armazenar referências, objetos e configurações em um banco de dados Aerospike.

Outro recurso é uma abstração flexível do sistema de arquivos. Usando https://pkg.go.dev/github.com/go-git/go billy/v5?tab=doc#Filesystem é fácil armazenar todos os arquivos de maneira diferente, ou seja, empacotando todos eles

Machine Translated by Google

para um único arquivo no disco ou mantendo-os todos na memória.

Outro caso de uso avançado inclui um cliente HTTP ajustável, como o encontrado em https://github.com/go-git/go-git/blob/master/_examples/custom_http/main.go.

```
customClient := &http.Client{ Transport:
&http.Transport{ // aceita qualquer certificado (pode ser útil para testes)

            TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
},
Timeout: 15 * time.Second, // 15 segundos de tempo limite
            CheckRedirect: func(req *http.Request, via []*http.Request) error { return
            http.ErrUseLastResponse // não siga o redirecionamento
}, }



// Substitui o protocolo padrão http(s) para usar nosso cliente personalizado
client.InstallProtocol("https", githttp.NewClient(customClient))

// Clone repositório usando o novo cliente se o protocolo for https:// r, err :=
git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})
```

Leitura adicional

Um tratamento completo das capacidades do go-git está fora do escopo deste livro. Se você quiser mais informações sobre o go-git, há documentação da API em https://pkg.go.dev/github.com/go-git/go-git/v5, e um conjunto de exemplos de uso em https://github.com/go-git/go-git/tree/master/_examples.

Dulwich

Há também uma implementação Git em Python puro - Dulwich. O projeto está hospedado em https://www.dulwich.io/. Ele visa fornecer uma interface para repositórios git (tanto locais quanto remotos) que não chamem o git diretamente, mas usem Python puro. No entanto, possui extensões C opcionais, que melhoram significativamente o desempenho.

Dulwich segue o git design e separa dois níveis básicos de API: encanamento e porcelana.

Aqui está um exemplo de uso da API de nível inferior para acessar a mensagem de commit do último commit:

```
from dulwich.repo import Repo r =
Repo('.') r.head() #

'57fbe010446356833a6ad1600059d80b1e731e15'


c = r[r.head()]
c
# <Commit 015fc1267258458901a94d228e39f0a378370466>
```

Machine Translated by Google

```
c.message
# 'Adicionar nota sobre codificação.\n'
```

Para imprimir um log de confirmação usando API de porcelana de alto nível, pode-se usar:

```
de dulwich import porcelana
porcelana.log('.', max_entries=1)

#commit: 57fbe010446356833a6ad1600059d80b1e731e15
#Autor: Jelmer Vernooij <jelmer@jelmer.uk> #Data: sábado,
29 de abril de 2017 23:57:34 +0000
```

Leitura adicional

A documentação da API, o tutorial e muitos exemplos de como realizar tarefas específicas com o Dulwich estão disponíveis no site oficial https://www.dulwich.io.

Machine Translated by Google

Apêndice C: Comandos do Git

Ao longo do livro, apresentamos dezenas de comandos do Git e tentamos arduamente apresentá-los em uma espécie de narrativa, adicionando mais comandos à história lentamente.
No entanto, isso nos deixa com exemplos de uso dos comandos um pouco espalhados por todo o livro.

Neste apêndice, examinaremos todos os comandos do Git que abordamos ao longo do livro, agrupados aproximadamente para o que eles são usados. Falaremos sobre o que cada comando geralmente faz e, em seguida, indicaremos onde no livro você pode nos encontrar usando-o.

ÿ

Você pode abreviar opções longas. Por exemplo, você pode digitar git commit --a, que age como se você digitasse git commit --amend. Isso só funciona quando as letras depois de -- são exclusivas para uma opção. Use a opção completa ao escrever scripts.

Configuração e configuração

Existem dois comandos que são bastante usados, desde as primeiras invocações do Git até os ajustes e referências comuns do dia a dia, os comandos config e help .

git config

O Git tem uma maneira padrão de fazer centenas de coisas. Para muitas dessas coisas, você pode dizer ao Git como padrão para fazê-las de uma maneira diferente ou definir suas preferências. Isso envolve tudo, desde dizer ao Git qual é o seu nome até preferências específicas de cor do terminal ou qual editor você usa. Existem vários arquivos que este comando lerá e gravará para que você possa definir valores globalmente ou para repositórios específicos.

O comando git config foi usado em quase todos os capítulos do livro.

Na configuração inicial do Git, nós o usamos para especificar nosso nome, endereço de e-mail e preferência de editor antes mesmo de começarmos a usar o Git.

Em Git Aliases, mostramos como você pode usá-lo para criar comandos abreviados que se expandem para longas sequências de opções, para que você não precise digitá-los todas as vezes.

Em Rebase, usamos para tornar --rebase o padrão quando você executa git pull.

No Credential Storage, nós o usamos para configurar um armazenamento padrão para suas senhas HTTP.

Em Expansão de palavras-chave , mostramos como configurar filtros de mancha e limpeza no conteúdo que entra e sai do Git.

Por fim, basicamente toda a Configuração do Git é dedicada ao comando.

git config core.editor comandos

Acompanhando as instruções de configuração em Seu Editor, muitos editores podem ser configurados da seguinte forma:

Machine Translated by Google

Tabela 4. Lista exaustiva de comandos de configuração do core.editor

editor	comando de configuração
Átomo	git config --global core.editor "atom --wait"
BBEdit (Mac, com ferramentas de linha de comando)	git config --global core.editor "bbedit -w"
Emacs	git config --global core.editor emacs
Gedit (Linux)	git config --global core.editor "gedit --wait --new-window"
Gvim (Windows 64 bits)	git config --global core.editor "'C:\Program Files\Vim\vim72\gvim.exe' --nofork '%*'" (Veja também a nota abaixo)
Kate (Linux)	git config --global core.editor "kate"
nano	git config --global core.editor "nano -w"
Bloco de notas (Windows 64 bits)	git config core.editor bloco de notas
Notepad++ (Windows 64 bits)	git config --global core.editor "'C:\Program Files\Notepad\notepad.exe' -multiInst -notabbar -nosession -noPlugin" (Veja também a nota abaixo)
Rascunho (Linux)	git config --global core.editor "editor de texto zero"
Texto sublime (macOS)	git config --global core.editor "/Applications/Sublime\ Text.app/Contents/ SharedSupport/bin/subl --new-window --wait"
Sublime Text (Windows 64-bit)	git config --global core.editor "'C:\Arquivos de Programas\Sublime Text 3\sublime_text.exe' -w" (Veja também a nota abaixo)
Editor de texto (macOS)	git config --global core.editor "open --wait-apps --new -e"
companheiro de texto	git config --global core.editor "mate -w"
Bloco de texto (Windows 64 bits)	git config --global core.editor "'C:\Program Files\TextPad 5\TextPad.exe' -m (Veja também a nota abaixo)
UltraEdit (Windows 64 bits)	git config --global core.editor Uedit32
Por que	git config --global core.editor "vim --nofork"
Código do Visual Studio	git config --global core.editor "code --wait"
VSCodium (Gratuito/Libre Aberto Binários de software de origem de VSCode)	git config --global core.editor "codium --wait"
Prancheta	git config --global core.editor "'C:\Program Files\Windows NT\Accessories\wordpad.exe"" git config --global core.editor "xi --wait"
XI	

Se você tiver um editor de 32 bits em um sistema Windows de 64 bits, o programa será instalado em C:\Arquivos de Programas (x86)\ em vez de C:\Arquivos de Programas\ como na tabela acima.

Machine Translated by Google

git ajuda

O comando git help é usado para mostrar toda a documentação fornecida com o Git sobre qualquer comando. Embora estejamos dando uma visão geral aproximada da maioria dos mais populares neste apêndice, para obter uma lista completa de todas as opções e sinalizadores possíveis para cada comando, você sempre pode executar git help <command> .

Apresentamos o comando git help em Obtendo ajuda e mostramos como usá-lo para encontrar mais informações sobre o shell git em Configurando o servidor.

Obtendo e Criando Projetos

Existem duas maneiras de obter um repositório Git. Uma é copiá-lo de um repositório existente na rede ou em outro lugar e a outra é criar um novo em um diretório existente.

Está quente

Para pegar um diretório e transformá-lo em um novo repositório Git para que você possa iniciar o controle de versão, basta executar git init.

Introduzimos isso pela primeira vez em Obtendo um repositório Git, onde mostramos a criação de um novo repositório para começar a trabalhar.

Falamos brevemente sobre como você pode alterar o nome da ramificação padrão de “mestre” nas Filiais Remotas.

Usamos esse comando para criar um repositório simples vazio para um servidor em Colocando o repositório básico em um servidor.

Por fim, passamos por alguns detalhes do que ele realmente faz nos bastidores de Encanadores e Porcelanas.

git clone

O comando git clone é, na verdade, uma espécie de wrapper em torno de vários outros comandos. Ele cria um novo diretório, entra nele e executa git init para torná-lo um repositório Git vazio, adiciona um controle remoto (git remote add) ao URL que você passa (por padrão chamado origin), executa um git fetch desse controle remoto repositório e, em seguida, verifica o commit mais recente em seu diretório de trabalho com git checkout.

O comando git clone é usado em dezenas de lugares ao longo do livro, mas listaremos apenas alguns lugares interessantes.

É basicamente apresentado e explicado em Cloning an Existing Repository, onde passamos por alguns exemplos.

Em Obtendo o Git em um servidor, veremos como usar a opção --bare para criar uma cópia de um repositório Git sem diretório de trabalho.

Em Bundling , usamos para desempacotar um repositório Git empacotado.

Machine Translated by Google

Por fim, em Cloning a Project with Submodules, aprendemos a opção --recurse-submodules para tornar a clonagem de um repositório com submodules um pouco mais simples.

Embora seja usado em muitos outros lugares ao longo do livro, esses são os únicos ou onde é usado de maneiras um pouco diferentes.

Instantâneos Básicos

Para o fluxo de trabalho básico de preparação de conteúdo e sua confirmação em seu histórico, existem apenas alguns comandos básicos.

git add

O comando git add adiciona conteúdo do diretório de trabalho na área de preparação (ou "índice") para o próximo commit. Quando o comando git commit é executado, por padrão, ele apenas olha para esta área de preparação, então git add é usado para criar exatamente o que você gostaria que seu próximo instantâneo de commit parecesse.

Este comando é incrivelmente importante no Git e é mencionado ou usado dezenas de vezes neste livro. Abordaremos rapidamente alguns dos usos exclusivos que podem ser encontrados.

Primeiro, apresentamos e explicamos o git add em detalhes em Rastreamento de novos arquivos.

Mencionamos como usá-lo para resolver conflitos de mesclagem em Basic Merge Conflicts.

Passamos a usá-lo para testar interativamente apenas partes específicas de um arquivo modificado no Interactive Staging.

Finalmente, nós o emulamos em um nível baixo em Tree Objects, para que você possa ter uma ideia do que ele está fazendo nos bastidores.

git status

O comando git status mostrará os diferentes estados dos arquivos em seu diretório de trabalho e área de preparação. Quais arquivos são modificados e não testados e quais são testados, mas ainda não confirmados. Em sua forma normal, ele também mostrará algumas dicas básicas sobre como mover arquivos entre esses estágios.

Abordamos primeiro o status em Verificando o status de seus arquivos, tanto na forma básica quanto na simplificada.
Embora o usemos ao longo do livro, praticamente tudo o que você pode fazer com o comando git status é abordado lá.

git diff

O comando git diff é usado quando você deseja ver as diferenças entre quaisquer duas árvores. Essa pode ser a diferença entre seu ambiente de trabalho e sua área de preparação (git diff por si só), entre sua área de preparação e seu último commit (git diff --staged) ou entre dois commits (git diff master branchB).

Primeiro, examinamos os usos básicos do git diff em Visualizando suas alterações preparadas e não testadas, onde mostramos como ver quais alterações foram testadas e quais ainda não foram testadas.

Nós o usamos para procurar possíveis problemas de espaço em branco antes de confirmar com a opção --check em

Machine Translated by Google

Diretrizes de Compromisso.

Vemos como verificar as diferenças entre as ramificações de forma mais eficaz com a sintaxe git diff A…B em Determinando o que é introduzido.

Nós o usamos para filtrar as diferenças de espaço em branco com -b e como comparar diferentes estágios de arquivos conflitantes com --theirs, --ours e --base em Advanced Merging.

Finalmente, nós o usamos para comparar efetivamente as alterações do submódulo com --submodule em Iniciando com submódulos.

git diftool

O comando git difftool simplesmente inicia uma ferramenta externa para mostrar a diferença entre duas árvores, caso você queira usar algo diferente do comando git diff embutido .

Mencionamos isso apenas brevemente em Visualizando suas alterações preparadas e não preparadas.

git commit

O comando git commit pega todo o conteúdo do arquivo que foi preparado com git add e registra um novo instantâneo permanente no banco de dados e, em seguida, move o ponteiro do branch atual até ele.

Primeiro abordamos os fundamentos do commit em Commit Your Changes. Lá também demonstramos como usar o sinalizador -a para pular a etapa git add em fluxos de trabalho diários e como usar o sinalizador -m para passar uma mensagem de confirmação na linha de comando em vez de iniciar um editor.

Em Desfazendo coisas, abordamos o uso da opção --amend para refazer o commit mais recente.

Em Branches in a Nutshell, entramos em mais detalhes sobre o que o git commit faz e por que ele faz isso.

Vimos como assinar commits criptograficamente com o sinalizador -S em Signing Commits.

Por fim, veremos o que o comando git commit faz em segundo plano e como ele é realmente implementado em objetos de confirmação.

git reset

O comando git reset é usado principalmente para desfazer coisas, como você pode perceber pelo verbo. Ele se move em torno do ponteiro HEAD e, opcionalmente, altera o índice ou a área de preparação e também pode, opcionalmente, alterar o diretório de trabalho se você usar --hard. Esta opção final torna possível que este comando perca seu trabalho se usado incorretamente, portanto, certifique-se de entendê-lo antes de usá-lo.

Primeiro, abordamos efetivamente o uso mais simples de git reset em Unstaging a Staged File, onde o usamos para remover um arquivo que executamos git add on.

Em seguida, cobrimos isso com alguns detalhes em Redefinir desmistificado, que é inteiramente dedicado a explicar esse comando.

Machine Translated by Google

Usamos git reset --hard para abortar uma mesclagem em Aborting a Merge, onde também usamos git merge --abort, que é um pouco como um wrapper para o comando git reset .

git rm

O comando git rm é usado para remover arquivos da área de teste e do diretório de trabalho do Git. É semelhante ao git add , pois ele prepara a remoção de um arquivo para o próximo commit.

Abordamos o comando git rm com alguns detalhes em Removendo arquivos, incluindo a remoção recursiva de arquivos e apenas a remoção de arquivos da área de preparação, mas deixando-os no diretório de trabalho com --cached.

O único outro uso diferente de git rm no livro é em Removendo objetos , onde usamos brevemente e explicamos o --ignore-unmatch ao executar git filter-branch, o que simplesmente faz com que não haja erro quando o arquivo que estamos tentando remover não não existe. Isso pode ser útil para fins de script.

git mv

O comando git mv é um comando fino e conveniente para mover um arquivo e, em seguida, executar git add no novo arquivo e git rm no arquivo antigo.

Apenas mencionamos brevemente este comando em Mover arquivos.

limpe

O comando git clean é usado para remover arquivos indesejados do seu diretório de trabalho. Isso pode incluir a remoção de artefatos de compilação temporários ou mesclar arquivos de conflito.

Abordamos muitas das opções e cenários nos quais você pode usar o comando clean em Limpando seu diretório de trabalho.

Ramificação e fusão

Existem apenas alguns comandos que implementam a maior parte da funcionalidade de ramificação e mesclagem no Git.

ramo git

O comando git branch é, na verdade, uma espécie de ferramenta de gerenciamento de branch. Ele pode listar as filiais que você possui, criar uma nova filial, excluir filiais e renomear filiais.

A maior parte do Git Branching é dedicada ao comando branch e é usado em todo o capítulo. Nós o apresentamos primeiro em Criando uma Nova Filial e examinamos a maioria de seus outros recursos (listando e excluindo) em Gerenciamento de Filiais.

Em Tracking Branches , usamos a opção git branch -u para configurar um tracking branch.

Por fim, examinamos um pouco do que ele faz em segundo plano nas Referências do Git.

Machine Translated by Google

git check-out

O comando git checkout é usado para alternar ramificações e verificar conteúdo em seu diretório de trabalho.

Primeiro encontramos o comando em Switching Branches junto com o comando git branch .

Vemos como usá-lo para iniciar o rastreamento de ramificações com o sinalizador --track em Rastreamento de ramificações.

Nós o usamos para reintroduzir conflitos de arquivo com --conflict=diff3 em Check-out de conflitos.

Entramos em mais detalhes sobre seu relacionamento com git reset em Reset Demystified.

Finalmente, entramos em alguns detalhes de implementação em The HEAD.

git merge

A ferramenta git merge é usada para mesclar uma ou mais ramificações na ramificação que você verificou. Ele então avançará a ramificação atual para o resultado da mesclagem.

O comando git merge foi introduzido pela primeira vez no Basic Branching. Embora seja usado em vários lugares no livro, há muito poucas variações do comando merge — geralmente apenas git merge <branch> com o nome do único branch no qual você deseja mesclar.

Abordamos como fazer uma mesclagem compactada (onde o Git mescla o trabalho, mas finge que é apenas um novo commit sem registrar o histórico da ramificação na qual você está mesclando) bem no final do Forked Public Project .

Examinamos muito sobre o processo e o comando de mesclagem, incluindo o comando -Xignore-space-change e o sinalizador --abort para abortar uma mesclagem com problema no Advanced Merging.

Aprendemos como verificar as assinaturas antes da fusão se o seu projeto estiver usando assinatura GPG em Signing Commits.

Por fim, aprendemos sobre a fusão de subárvores na fusão de subárvores.

git mergetool

O comando git mergetool simplesmente inicia um auxiliar de mesclagem externo caso você tenha problemas com uma mesclagem no Git.

Mencionamos isso rapidamente em Basic Merge Conflicts e detalhamos como implementar sua própria ferramenta de mesclagem externa em External Merge and Diff Tools.

git log

O comando git log é usado para mostrar o histórico gravado alcançável de um projeto desde o instantâneo de commit mais recente para trás. Por padrão, ele mostrará apenas o histórico da ramificação em que você está atualmente, mas pode receber cabeças ou ramificações diferentes ou mesmo múltiplas para percorrer.
Também é frequentemente usado para mostrar diferenças entre duas ou mais ramificações no nível de confirmação.

Machine Translated by Google

Este comando é usado em quase todos os capítulos do livro para demonstrar a história de um projeto.

Apresentamos o comando e o abordamos com mais profundidade em Exibindo o histórico de commits. Lá, examinamos as opções -p e --stat para ter uma ideia do que foi introduzido em cada confirmação e as opções --pretty e --oneline para visualizar o histórico de forma mais concisa, juntamente com algumas opções simples de filtragem de data e autor.

Em Criando uma nova ramificação, usamos com a opção --decorate para visualizar facilmente onde nossos ponteiros de ramificação estão localizados e também usamos a opção --graph para ver como são os históricos divergentes.

Em Private Small Team e Commit Ranges, abordamos a sintaxe branchA..branchB para usar o comando git log para ver quais commits são exclusivos de um branch em relação a outro branch. Em Commit Ranges, passamos por isso bastante extensivamente.

Em Merge Log e Triple Dot , abordamos o uso do formato branchA…branchB e a sintaxe --left-right para ver o que está em um branch ou outro, mas não em ambos. Em Merge Log, também veremos como usar a opção --merge para ajudar na depuração de conflitos de mesclagem, bem como usar a opção --cc para examinar os conflitos de confirmação de mesclagem em seu histórico.

Em RefLog Shortnames , usamos a opção -g para visualizar o Git reflog por meio dessa ferramenta, em vez de fazer o branch traversal.

Em Pesquisa, examinamos o uso das opções -S e -L para fazer pesquisas bastante sofisticadas por algo que aconteceu historicamente no código, como ver o histórico de uma função.

Em Signing Commits , vemos como usar --show-signature para adicionar uma string de validação a cada commit na saída git log com base em se foi assinado validamente ou não.

git stash

O comando git stash é usado para armazenar temporariamente o trabalho não confirmado para limpar seu diretório de trabalho sem ter que confirmar o trabalho inacabado em uma ramificação.

Isso é basicamente totalmente coberto em Armazenamento e Limpeza.

dia git

O comando git tag é usado para dar um marcador permanente a um ponto específico no histórico do código. Geralmente isso é usado para coisas como lançamentos.

Esse comando é apresentado e abordado em detalhes em Tagging e o usamos na prática em Tagging Your Releases.

Também abordamos como criar uma tag assinada GPG com o sinalizador -s e verificar uma com o sinalizador -v em Assinando seu trabalho.

Compartilhando e atualizando projetos

Não existem muitos comandos no Git que acessam a rede, quase todos os comandos operam no banco de dados local. Quando estiver pronto para compartilhar seu trabalho ou obter alterações de

Machine Translated by Google

em outros lugares, há um punhado de comandos que lidam com repositórios remotos.

git busca

O comando git fetch se comunica com um repositório remoto e busca todas as informações que estão naquele repositório que não estão no seu atual e as armazena em seu banco de dados local.

Primeiro, examinamos esse comando em Buscando e puxando de seus remotos e continuamos a ver exemplos de seu uso em Ramificações remotas.

Também o usamos em vários exemplos em Contribuindo para um projeto.

Nós o usamos para buscar uma única referência específica que está fora do espaço padrão em Pull Request Refs e vemos como buscar de um pacote em Bundling.

Configuramos refspecs altamente personalizados para fazer com que git fetch faça algo um pouco diferente do padrão em The Refspec.

git pull

O comando git pull é basicamente uma combinação dos comandos git fetch e git merge , onde o Git buscará do remoto que você especificar e imediatamente tentará mesclá-lo na ramificação em que você está.

Nós o apresentamos rapidamente em Buscando e extraindo de seus controles remotos e mostramos como ver o que ele mesclará se você executá-lo em Inspecionando um controle remoto.

Também vemos como usá-lo para ajudar nas dificuldades de rebase em Rebase When You Rebase.

Mostramos como usá-lo com uma URL para obter alterações de forma única em Check-out de ramificações remotas.

Por fim, mencionamos rapidamente que você pode usar a opção --verify-signatures para verificar se os commits que você está puxando foram assinados pelo GPG em Signing Commits.

git push

O comando git push é usado para se comunicar com outro repositório, calcular o que seu banco de dados local possui que o remoto não possui e, em seguida, enviar a diferença para o outro repositório.
Ele requer acesso de gravação ao outro repositório e, portanto, normalmente é autenticado de alguma forma.

Primeiro, examinamos o comando git push em Como enviar para seus controles remotos. Aqui, abordamos os fundamentos do push de uma ramificação para um repositório remoto. Em Pushing, nos aprofundamos um pouco mais no push de branches específicos e em Tracking Branches , vemos como configurar tracking branches para pushar automaticamente. Em Excluindo ramificações remotas, usamos o sinalizador --delete para excluir uma ramificação no servidor com git push.

Ao longo de Contribuindo para um projeto, vemos vários exemplos de uso do git push para compartilhar o trabalho em ramificações por meio de vários controles remotos.

Machine Translated by Google

Vemos como usá-lo para compartilhar tags que você criou com a opção --tags em Compartilhamento de tags.

Em Publishing Submodule Changes, usamos a opção --recurse-submodules para verificar se todo o trabalho de nossos submódulos foi publicado antes de enviar o superprojeto, o que pode ser muito útil ao usar submódulos.

Em Outros ganchos de cliente, falamos brevemente sobre o gancho pré-push , que é um script que podemos configurar para ser executado antes que um push seja concluído para verificar se ele deve ter permissão para enviar.

Por fim, em Pushing Refspecs , analisamos o push com uma refspec completa em vez dos atalhos gerais que normalmente são usados. Isso pode ajudá-lo a ser muito específico sobre o trabalho que deseja compartilhar.

git remote

O comando git remote é uma ferramenta de gerenciamento para seu registro de repositórios remotos. Ele permite que você salve URLs longos como identificadores curtos, como “origem”, para que você não precise digitá-los o tempo todo. Você pode ter vários deles e o comando git remote é usado para adicioná-los, alterá-los e excluí-los.

Este comando é abordado em detalhes em Trabalhando com controles remotos, incluindo listagem, adição, remoção e renomeação deles.

Ele também é usado em quase todos os capítulos subseqüentes do livro, mas sempre no formato git remote add <nome> <url> padrão.

arquivo git

O comando git archive é usado para criar um arquivo de um instantâneo específico do projeto.

Usamos o arquivo git para criar um tarball de um projeto para compartilhamento em Preparando um lançamento.

submódulo git

O comando git submodule é usado para gerenciar repositórios externos dentro de repositórios normais.
Isso pode ser para bibliotecas ou outros tipos de recursos compartilhados. O comando submodule possui vários subcomandos (adicionar, atualizar, sincronizar, etc) para gerenciar esses recursos.

Este comando é apenas mencionado e totalmente abordado em Submódulos.

Inspeção e Comparação

git show

O comando git show pode mostrar um objeto Git de forma simples e legível. Normalmente você usaria isso para mostrar as informações sobre uma tag ou um commit.

Nós o usamos primeiro para mostrar informações de tags anotadas em Tags anotadas.

Mais tarde, nós o usamos bastante na Seleção de revisão para mostrar os commits para os quais nossas várias seleções de revisão resolvem.

Machine Translated by Google

Uma das coisas mais interessantes que fazemos com o git show é a recombinação manual de arquivos para extrair conteúdos de arquivos específicos de vários estágios durante um conflito de mesclagem.

git shortlog

O comando git shortlog é usado para resumir a saída do git log. Serão necessárias muitas das mesmas opções que o comando git log , mas em vez de listar todos os commits, ele apresentará um resumo dos commits agrupados por autor.

Mostramos como usá-lo para criar um bom changelog no The Shortlog.

git descreva

O comando git describe é usado para pegar qualquer coisa que resolva para um commit e produza uma string que seja legível por humanos e não mude. É uma maneira de obter uma descrição de um commit tão inequívoca quanto um commit SHA-1, mas mais compreensível.

Usamos git describe em Gerando um número de compilação e preparando uma versão para obter uma string para nomear nosso arquivo de versão.

Depurando

O Git tem alguns comandos que são usados para ajudar a depurar um problema em seu código. Isso varia desde descobrir onde algo foi introduzido até descobrir quem o introduziu.

git bisect

A ferramenta git bisect é uma ferramenta de depuração incrivelmente útil usada para descobrir qual commit específico foi o primeiro a introduzir um bug ou problema fazendo uma busca binária automática.

É totalmente coberto na Pesquisa Binária e é mencionado apenas nessa seção.

git culpa

O comando git culpe anota as linhas de qualquer arquivo com o qual o commit foi o último a introduzir uma alteração em cada linha do arquivo e qual pessoa criou esse commit. Isso é útil para encontrar a pessoa para solicitar mais informações sobre uma seção específica do seu código.

Ele é abordado em Anotação de arquivo e é mencionado apenas nessa seção.

git grep

O comando git grep pode ajudá-lo a encontrar qualquer string ou expressão regular em qualquer um dos arquivos em seu código-fonte, mesmo em versões mais antigas de seu projeto.

Ele é abordado no Git Grep e é mencionado apenas nessa seção.

Machine Translated by Google

Correção

Alguns comandos no Git são centrados no conceito de pensar em commits em termos das mudanças que eles introduzem, como se a série de commits fosse uma série de patches. Esses comandos ajudam você a gerenciar suas ramificações dessa maneira.

git cherry-pick

O comando git cherry-pick é usado para pegar a mudança introduzida em um único commit do Git e tentar reintroduzi-lo como um novo commit no branch em que você está atualmente. Isso pode ser útil para obter apenas um ou dois commits de uma ramificação individualmente, em vez de mesclar na ramificação que recebe todas as alterações.

Cherry picking é descrito e demonstrado em Rebasing e Cherry-Picking Workflows.

git fox

O comando git rebase é basicamente uma escolha automatizada. Ele determina uma série de commits e então os seleciona um por um na mesma ordem em outro lugar.

O rebase é abordado em detalhes em Rebase, incluindo a cobertura dos problemas colaborativos envolvidos com o rebase de ramificações que já são públicas.

Nós o usamos na prática durante um exemplo de divisão de seu histórico em dois repositórios separados no Replace, usando o sinalizador --onto também.

Passamos por um conflito de mesclagem durante o rebase em Rerere.

Também o usamos em um modo de script interativo com a opção -i em Alterar várias mensagens de confirmação.

git reverter

O comando git revert é essencialmente uma seleção reversa do git cherry. Ele cria um novo commit que aplica exatamente o oposto da mudança introduzida no commit que você está almejando, basicamente desfazendo ou revertendo.

Usamos isso em inverter o commit para desfazer um commit de mesclagem.

E-mail

Muitos projetos Git, incluindo o próprio Git, são inteiramente mantidos em listas de discussão. O Git tem várias ferramentas incorporadas que ajudam a tornar esse processo mais fácil, desde a geração de patches que você pode facilmente enviar por e-mail até a aplicação desses patches em uma caixa de e-mail.

git apply

O comando git apply aplica um patch criado com o comando git diff ou mesmo GNU diff. É semelhante ao que o comando patch pode fazer com algumas pequenas diferenças.

Machine Translated by Google

Demonstramos como usá-lo e as circunstâncias em que você pode fazê-lo em Aplicando patches de e-mail.

vá de manhã

O comando git am é usado para aplicar patches de uma caixa de entrada de e-mail, especificamente uma que esteja formatada em mbox. Isso é útil para receber patches por e-mail e aplicá-los facilmente ao seu projeto.

Abordamos o uso e o fluxo de trabalho do git am em Aplicar um patch com am , incluindo o uso das opções --resolved, -i e -3 .

Há também vários ganchos que você pode usar para ajudar com o fluxo de trabalho em torno do git am e todos eles são abordados em Ganchos de fluxo de trabalho de email.

Também o usamos para aplicar alterações de solicitação de pull do GitHub com formato de patch nas notificações por email.

git format-patch

O comando git format-patch é usado para gerar uma série de patches no formato mbox que você pode usar para enviar para uma lista de discussão devidamente formatada.

Passamos por um exemplo de contribuição para um projeto usando a ferramenta git format-patch em Public Project over Email.

git imap-send

O comando git imap-send carrega uma caixa de correio gerada com git format-patch em uma pasta de rascunhos IMAP.

Passamos por um exemplo de contribuição para um projeto enviando patches com a ferramenta git imap-send em Public Project over Email.

git enviar e-mail

O comando git send-email é usado para enviar patches que são gerados com git format-patch por e-mail.

Passamos por um exemplo de contribuição para um projeto enviando patches com a ferramenta git send-email em Public Project over Email.

git request-pull

O comando git request-pull é simplesmente usado para gerar um corpo de mensagem de exemplo para enviar por e-mail a alguém. Se você tiver uma ramificação em um servidor público e quiser que alguém saiba como integrar essas alterações sem enviar os patches por e-mail, execute este comando e envie a saída para a pessoa para quem deseja obter as alterações.

Demonstramos como usar o git request-pull para gerar uma mensagem pull no Forked Public Project.

Machine Translated by Google

Sistemas Externos

O Git vem com alguns comandos para integração com outros sistemas de controle de versão.

git svn

O comando git svn é usado para se comunicar com o sistema de controle de versão do Subversion como um cliente. Isso significa que você pode usar o Git para fazer check-out e se comprometer com um servidor Subversion.

Este comando é abordado em profundidade no Git e no Subversion.

git fast-import

Para outros sistemas de controle de versão ou importação de praticamente qualquer formato, você pode usar git fast import para mapear rapidamente o outro formato para algo que o Git possa gravar facilmente.

Esse comando é abordado em profundidade em Um importador personalizado.

Administração

Se você estiver administrando um repositório Git ou precisar corrigir algo em grande escala, o Git fornece vários comandos administrativos para ajudá-lo.

git gc

O comando git gc executa a “coleta de lixo” em seu repositório, removendo arquivos desnecessários em seu banco de dados e compactando os arquivos restantes em um formato mais eficiente.

Este comando normalmente é executado em segundo plano para você, embora você possa executá-lo manualmente, se desejar. Examinamos alguns exemplos disso em Manutenção.

git fsck

O comando git fsck é usado para verificar se há problemas ou inconsistências no banco de dados interno.

Só usamos isso rapidamente uma vez na Recuperação de Dados para procurar objetos pendurados.

git reflog

O comando git reflog passa por um registro de onde todos os cabeçalhos de suas ramificações estiveram enquanto você trabalha para encontrar confirmações que você pode ter perdido ao reescrever históricos.

Abordamos esse comando principalmente em RefLog Shortnames, onde mostramos o uso normal e como usar git log -g para visualizar as mesmas informações com a saída git log .

Também passamos por um exemplo prático de recuperação de tal ramo perdido na Recuperação de Dados.

Machine Translated by Google

git filter-branch

O comando git filter-branch é usado para reescrever cargas de commits de acordo com certos padrões, como remover um arquivo em qualquer lugar ou filtrar todo o repositório em um único subdiretório para extrair um projeto.

Em Removendo um arquivo de cada confirmação, explicamos o comando e exploramos várias opções diferentes, como --commit-filter, --subdirectory-filter e --tree-filter.

No Git-p4 nós o usamos para consertar repositórios externos importados.

Comandos de encanamento

Também havia vários comandos de encanamento de nível inferior que encontramos no livro.

O primeiro que encontramos é ls-remote em Pull Request Refs , que usamos para examinar as referências brutas no servidor.

Usamos ls-files em Manual File Re-merging, Rerere e The Index para dar uma olhada mais crua na aparência da sua área de teste.

Também mencionamos rev-parse em Branch References para pegar praticamente qualquer string e transformá-la em um objeto SHA-1.

No entanto, a maioria dos comandos de encanamento de baixo nível que abordamos estão no Git Internals, que é mais ou menos o foco do capítulo. Tentamos evitar o uso deles durante a maior parte do livro.

Machine Translated by Google

Índice

@

$EDITOR, 336
$VISUAL
    veja $EDITOR,
336 .gitignore, 337 .NET,
474 @{upstream}, 93
@{u}, 93

A

pseudônimos, 60
Apache, 117
Apple, 474
arquivamento,
351 atributos,
345 autocorreção, 338

B

golpe, 464
Bazar, 406
arquivos binários, 345
BitKeeper, 14
branches, 63
    workflow básico, 70
    criação, 65 exclusão
    remota, 94 diffing, 155
    execução longa, 82
    gerenciamento, 79
    fusão, 75 remota, 85,
    154 comutação, 66
    tópico, 83, 151
    rastreamento, 92
    upstream, 92 números
    de construção, 163

C

C, 470
C#, 474
Cocoa, 474
cores, 338
modelos de confirmação,
336 contribuições, 129
    equipes gerenciadas privadas, 138
    equipe pequena privada, 131
    grande projeto público, 147
    pequeno projeto público, 143
credenciais, 329 crlf, 342

CVS, 11

D

difftool, 339
git distribuído, 126
Dulwich, 480

E

editor
    alterando o padrão, 36
emails, 149 aplicando
    patches de, 151 exclusões, 337,
419

F

arquivos em
    movimento, 39
removendo, 38 bifurcando, 128, 171

G

Git como cliente, 366
comandos git add, 29,
    30, 30 am, 152
    apply, 151 archive,
    164 branch, 65,
    79 checkout, 66
    cherry-pick, 160
    clone, 27 bare,
    110 commit, 36, 63
    config, 21, 23, 36,
        61, 149, 335
    credencial, 329
    daemon, 116 descrever, 163 diff,
    33 verificar, 130 importação rápida,
    410

Machine Translated by Google

fetch, 52
fetch-pack, 443 filter-
branch, 410 format-
patch, 147 gitk, 457 gui,
457 help, 24, 116 http-
backend, 117 init, 27, 30

bare, 111, 114
instaweb, 119 log, 40
merge, 73 squash, 146
mergetool, 78 p4, 394,
409 pull, 53 push,
53, 58, 91 rebase, 96
receive-pack, 441
remote, 51, 52, 53 , 54
request-pull, 144 rerere,
162 send-pack, 441
shortlog, 164 show, 57
show-ref, 369 status, 28, 36
svn, 366 tag, 55, 56, 58
upload-pack, 443 git-svn, 366

GitHub, 166
API, 212
Flow, 172
organizações, 205
solicitações pull, 175
contas de usuário, 166
GitHub para macOS, 459
GitHub para Windows, 459 gitk,
457
GitLab, 120
GitWeb, 119
Vai, 478
vai-git, 478
GPG, 337
Ferramentas gráficas, 457
GUIs, 457

H

ganchos, 353
pós-atualização, 107

eu

ignorando arquivos, 32
Importando
do Bazaar, 406 do
Mercurial, 403 de outros,
410 do Perforce, 409 do
Subversion, 401

integrando o trabalho, 156
Interoperação com outros VCSs
Mercúrio, 376
Forçosamente, 386
Subversão, 366
IRC, 24

J

Java, 475
jgit, 475

k

expansão de palavras-chave, 348

eu

libgit2, 470
finais de linha, 342
Linus Torvalds, 14
Linux, 14
instalação, 18
filtragem de log, 45
formatação de log, 42

M

Mac OS
instalação, 19
manutenção de um projeto, 150
master, 64 Mercurial, 376, 403
mergetool, 339 merge, 75 conflitos,
76 estratégias, 353 vs. rebase, 103
Migrating to Git, 401 Mono, 474

Machine Translated by Google

O

Objective-C, 474
origem, 86

P

pager, 337
Forforce, 11, 14, 386, 409
Git Fusion, 387
exemplo de política, 356
posh-git, 467
PowerShell, 467
protocolos HTTP
burro, 107 git, 109
local, 105 HTTP
inteligente, 107

SSH, 108
puxando, 94
empurrando, 91
Python, 475, 480

R

rebase, 95
perigos de, 100
vs. fusão, 103
referências
remoto, 85
liberação, 164 voo,
162
Rubi, 471

S

servindo repositórios, 105
protocolo git, 116
GitLab, 120
GitWeb, 119
HTTP, 117
SSH, 111
SHA-1, 16
shell solicita bash,
464
PowerShell, 467 zsh,
465
Chaves SSH, 112
com GitHub, 167 saltos
de área de preparação, 37

Subversão, 11, 14, 127, 366, 401

T

bash de conclusão de
tabulação, 464
PowerShell, 467 zsh,
465 tags, 55, 162
anotados, 56 leves, 57
assinaturas, 162

EM

controle de versão, 10
centralizado, 11
distribuído, 12 local,
10
Estúdio Visual, 462

EM

espaço em branco, 342
janelas
instalação, 19
fluxos de trabalho,
126 centralizado, 126
ditador e tenentes, 128 gerente de
integração, 127 fusão, 156 fusão
(grande), 159 rebase e seleção seletiva,
160

X

Código X, 19

COM

zsh, 465